Université **PARIS 7**

OVERLOADING, SUBTYPING AND LATE BINDING: FUNCTIONAL FOUNDATION OF OBJECT-ORIENTED PROGRAMMING

PhD dissertation

January 1994

Advisor:
Giuseppe Longo

Referees:
Kim Bruce
Luca Cardelli
Didier Rémy

Chairperson:
Guy Cousineau

Committee Members:
Serge Abiteboul
Jean-Pierre Jouannaud
Giuseppe Longo
Didier Rémy
Patrick Sallé

Giuseppe Castagna

Laboratoire d'Informatique
de l'Ecole Normale Supérieure

Université **PARIS 7**

SURCHARGE, SOUS-TYPAGE ET LIAISON TARDIVE : FONDEMENTS FONCTIONNELS DE LA PROGRAMMATION ORIENTÉE OBJETS

Thèse de doctorat

janvier 1994

Directeur de thèse :
Giuseppe Longo

Rapporteurs :
Kim Bruce
Luca Cardelli
Didier Rémy

Président du Jury:
Guy Cousineau

Examinateurs :
Serge Abiteboul
Jean-Pierre Jouannaud
Giuseppe Longo
Didier Rémy
Patrick Sallé

Giuseppe Castagna

Laboratoire d'Informatique
de l'Ecole Normale Supérieure

Quest tesi è dedicata a
mio nonno 'Gin
che mi mostró la dolcezza, la bontá e la tolleranza,
mio nonno Beppe
che mi mostró la coerenza e la perseveranza
Ilaria e `\ldots`
per i momenti felici passati assieme

Acknowledgments

Giuseppe Longo really deserves the first position in this list. It is not the task of a simple PhD. student to praise the scientific qualities of Prof. Longo; but since I had the chance to work with him and to share some of his time, I can witness of his exceptional human qualities. He was always ready to listen to me, to help me in the difficult moments, to calm down with patience my enthusiasms when they were too strong and to tolerate my faults. His attitude was the one of a permanent "learner": he listens to you to learn from you. Thus I want to thank more the man than the scholar, because if it is true that I owe much to the latter, I am indebted for a more important example to the former. And I am happy to express here all my appreciation for him.

All my thanks also to Giorgio Ghelli. This thesis owes him a lot: the incipient intuition, some ideas and results presented in this thesis come from him. The "anschauung" he transmitted me was very helpful to perceive the general setting behind every particular result.

Benli Pierce is a great researcher and a friend; thanks to him this thesis has one chapter written in *real* english (apart from the modifications I made)

Luca Cardelli deserves a triple acknowledgment: for having started the type theoretic research on object-oriented programming, for his suggestions and the various discussions I had with him —thanks to him the chapter 8 of this thesis exists—, and for the incredible courage demonstrated in accepting to be the referee of this thesis. The same holds for Kim Bruce, which has my gratitude also for his advice and for the warm hospitality he and his wife Fatima offered me in Williamstown. Didier Rémy completes the list of the audacious who accepted the challenge of judging my thesis; I thank him for this and for the interest he always demonstrated in my work.

Thanks a lot to Guy Cousineau, for his help, his amiability and his always ready advice: I owe him a lot. With him I also want to thank the Laboratoire d'Informatique de l'École Normale Supérieure, who offered me a pleasant and stimulating place to develop this work. The other department I want warmly to thank is the DISI of University of Genoa, in the person of Eugenio Moggi: his advice and his comprehension helped me a lot and I really regretted to be obliged to abandon Genoa to end this thesis. Thanks also to the University of Pisa for all the years I spent there and the Consiglio Nazionale delle Ricerche, Comitato Nazionale per le Scienze Matematiche for its financial support.

I am also grateful to Martín Abadi whose suggestions the results of section 8.5 come from; to Véronique Benzaken who introduced me to O_2; to John Lamping who suggested me the study of one of the systems in chapter 4; to John Mitchell for his advice and his hospitality

at Stanford; to Hideki Tsuiki who pointed me out an error in the semantics for $\lambda\&$; to Allyn Dimock, Maribel Fernández, and some anonymous referees (I will never say which ones!) for their comments on the drafts of some papers; to Dinesh Katiyar for his courage in assisting four times to the same talk I gave in different places; to Kathleen Milsted for her help in writing the introduction of this thesis; to Bob Muller and his wife Susan for their hospitality during my visit to the Apple-Eastern Research and Technology Lab. and also to the Dylan group for the many stimulating discussions; to Maria Virginia Apónte, François Bouladoux, Lucky Chillan, Pierre Cregut, Adriana Compagnoni, Pierre-Louis Curien, Roberto Di Cosmo, Furio Honsell, Delia Kesner, Simone Martini, Chet Murthy, Pino Rosolini for the discussions I had with them.

I also want to thank the friends of these years that are not cited above: Alejandro, Alessandra, Antonio, Carola, Cristina (J. P.-A.), Ilaria, Inés, José, Kiki, Liliane, Mario, Michel, Nadine, Pompeo, Roberto, Tiziana, Tomás(!), Yiyo just because they were there … which means a lot.

Let me end with the persons who are the demiurges of this thesis: Franca and Nico, my parents. I want to thank them for their constant support; for having trusted in me; for having taught me to take my responsibilities and to respect every other person; for the example they were for me ... in a word: for their love. And if this thesis cannot certainly recompense all they gave to me, could be a comfort to our separation to know that I do love them.

Preface

E poi che la sua mano alla mia pose
con lieto volto, ond'io mi confortai,
mi mise dentro alle segrete cose.

Dante Alighieri
Inferno; iii, 19-21

Many of the results in this thesis have already been published in review or conference proceedings. More precisely chapters 2 and 3 are based on an article to appear in *Information and Computation* [CGL92b] whose extended abstract can be found in the proceedings of the *1992 ACM Conference on LISP and Functional Programming* [CGL92a]. The first and the fifth chapters are partially based on a paper whose (very) preliminary version appeared as a Technical Report of LIENS [Cas92], and whose extended abstract has been published in the *13th Conference on Foundation of Software Technology and Theoretical Computer Science* [Cas93b]. The extended abstract of the sixth chapter appears in the proceedings of the *International Conference on Typed Lambda Calculi and Applications* [CGL93]. Chapter 9 and part of chapter 10 are contained in a paper actually under submission, whose extended abstract appears in the proceedings of the *4th International Workshop on Data Base Programming Languages* [Cas93a]. The extended abstract of chapter 8 will be presented at the *21st Annual Symposium on Principle Of Programming Languages*. Finally for the second and the seventh chapter we also used part of the course notes for a summer school in Nice [CL91b].

Most of these papers are coauthored: without the essential contributions of Giorgio Ghelli, Giuseppe Longo and Benjamin Pierce this thesis certainly could not be but much poorer than what it is. We will recall the other authors at the beginning of each chapter whose results are not due only to this thesis's author.

Contents

Présentation de la thèse 13
Programmation orientée objets . . . 18
Le $\lambda\&$-calcul . . . 19
Normalisation Forte . . . 26
Trois variations sur le thème . . . 27
Un méta-langage de $\lambda\&$ . . . 31
Sémantique . . . 34
Second ordre . . . 37
Vers une quantification bornée décidable . . . 38
Quantification bornée avec surcharge . . . 42
Surcharge de second ordre et programmation orientée objets . . . 45
Conclusion . . . 48

Introduction 53

Background and notation 61
Term rewriting systems . . . 61
Logic . . . 62

I Simple typing 63

1 Object-oriented programming 65
1.1 A kernel functional object-oriented language . . . 65
1.1.1 Objects . . . 65
1.1.2 Messages . . . 66
1.1.3 Methods and functions . . . 66
1.1.4 Classes . . . 68
1.1.5 Inheritance . . . 70
1.1.6 Multiple inheritance . . . 72
1.1.7 Extending classes . . . 74
1.1.8 Super, self and the use of coercions . . . 75
1.1.9 Multiple dispatch . . . 76
1.1.10 Messages as first-class values: adding overloading . . . 77

1.2 Type checking 78
1.2.1 The types 78
1.2.2 Intuitive typing rules 79

2 The $\lambda\&$-calculus 83
2.1 Informal presentation 83
2.1.1 Subtyping, run-time types and late binding 84
2.2 The syntax of the $\lambda\&$-calculus 86
2.2.1 Subtyping rules. 86
2.2.2 Types 87
2.2.3 Terms 88
2.2.4 Type checking 89
2.2.5 Reduction Rules 91
2.3 The Generalized Subject Reduction Theorem 94
2.4 Church-Rosser 97
2.5 Basic encodings 99
2.5.1 Surjective pairings 100
2.5.2 Simple records 100
2.5.3 Updatable records 101
2.6 $\lambda\&$ and object-oriented programming 102
2.6.1 The "objects as records" analogy 104
2.6.2 Binary methods and multiple dispatch 107
2.6.3 Covariance vs. contravariance 108
2.6.4 Abstract classes 109

3 Strong Normalization 113
3.1 The full calculus is not normalizing 113
3.2 Fixed point combinators 114
3.3 The reasons for non normalization 115
3.4 Typed-inductive properties 117
3.5 Strong Normalization is typed-inductive 120

4 Three variations on the theme 123
4.1 More freedom to the system: $\lambda\&^+$ 123
4.1.1 Modifying the good formation of types 124
4.1.2 Modifying the formation of the terms 125
4.1.3 Modifying the notion of reduction 126
4.1.4 Conservativity 128
4.1.5 Subject Reduction 128
4.1.6 Church-Rosser 129
4.1.7 Strong Normalization 129
4.2 Adding explicit coercions 130
4.2.1 Subject Reduction 131
4.2.2 Church Rosser 131
4.2.3 Strong Normalization 132

4.2.4 More on updatable records 133
4.3 Unifying overloading and λ-abstraction: $\lambda^{\{\}}$ 134
4.3.1 Subject Reduction 135
4.3.2 Church-Rosser 136
4.4 Reference to other work 139

5 A meta-language from $\lambda\&$ 141
5.1 The formal presentation of the toy language 142
5.1.1 The terms of the language 142
5.1.2 The types of the language 144
5.2 λ_object 152
5.2.1 The type system 156
5.2.2 Some results 158
5.3 Translation 160
5.3.1 Simple methods without recursion 161
5.3.2 With multi-methods 164
5.3.3 With recursive methods 166
5.3.4 Correctness of the type-checking 167
5.4 λ_object and $\lambda\&$ 167
5.4.1 The encoding of the types 168
5.4.2 The encoding of the terms 170

6 Semantics 173
6.1 Introduction 173
6.2 The completion of overloaded types 174
6.3 Early Binding 177
6.4 Semantics 179
6.4.1 PER as a model 179
6.4.2 Overloaded types as Products 182
6.4.3 The semantics of terms 186
6.5 Summary of the semantics 191

II Second order 195

7 Introduction to part II 197
7.1 The loss of information in the record-based models: a short history 198
7.1.1 Implicit Polymorphism 198
7.1.2 Explicit Polymorphism 199
7.1.3 $F_{\leq}$ 200

8 A roadmap to decidable bounded quantification 203
8.1 Introduction 203
8.2 Syntax 206
8.3 Expressiveness 207

8.4 Basic Properties . . . 208
8.4.1 Subtyping algorithm . . . 209
8.4.2 Meets and joins . . . 212
8.5 Semantics . . . 214
8.5.1 The language **TARGET** . . . 216
8.5.2 Translation . . . 218
8.6 Conservativity of Recursive Types . . . 220
8.7 The typing relation . . . 222
8.8 Conclusions . . . 223

9 Bounded quantification with overloading 225
9.1 The loss of information in the overloading-based model . . . 225
9.1.1 Type dependency . . . 227
9.2 Type system . . . 229
9.2.1 Some useful results . . . 230
9.2.2 Transitivity elimination . . . 232
9.2.3 Subtyping algorithm and coherence of the system . . . 237
9.3 Terms . . . 240
9.4 Reduction . . . 242
9.4.1 The encoding of records . . . 243
9.4.2 Generalized Subject Reduction . . . 244
9.4.3 Church-Rosser . . . 257
9.5 Decidable subtyping . . . 260
9.5.1 Subtyping algorithm . . . 260
9.5.2 Termination . . . 262
9.5.3 Terms and reduction . . . 264

10 Second order overloading and object-oriented programming 267
10.1 Object-oriented programming . . . 267
10.1.1 Extending classes . . . 270
10.1.2 First class messages, super and coerce . . . 270
10.1.3 Typing rules for the toy language . . . 271
10.1.4 Multiple dispatch . . . 272
10.1.5 Advanced features . . . 274
10.2 Future work . . . 274

11 Conclusion 277
11.1 Proof Theory . . . 277
11.2 Object-oriented programming . . . 279
11.2.1 Inheritance . . . 282
11.2.2 Higher-order bounds . . . 284
11.2.3 Beyond object-oriented programming . . . 284

III Appendixes **285**

A Implementation of λ_object **287**
A.1 The language . . . 288
A.2 The module . . . 291

B Type system of λ_object **309**
B.1 Types . . . 309
B.2 Typing rules . . . 309

C Specification of the toy language **311**
C.1 Terms . . . 311
C.2 Subtyping . . . 312
C.2.1 Auxiliary Notation . . . 312
C.3 Typing Rules . . . 313

D Proof of theorem 5.3.8 **315**

E Original $\mathbf{F}_{\leq}$ rules **321**
E.1 Subtyping . . . 321
E.2 Typing . . . 321
E.3 Typing algorithm . . . 322

F Translation of $\mathbf{F}_{\leq}^{\top}$ into explicit coercions **323**

Présentation de la thèse

L'écriture qui semble devoir fixer la langue, est précisément ce qui l'altère ; elle ne change pas les mots, mais la génie ; elle substitue l'exactitude à l'expression. L'on rend ses sentiments quand on parle et ses idées quand on écrit.

Jean-Jacques Rousseau
Essai sur l'origine des langues

Durant ces deux dernières décennies une distinction importante a été largement utilisée en Théorie des Langages entre polymorphisme paramétrique et polymorphisme "ad hoc" [Str67] (voir aussi [CW85]). Le polymorphisme paramétrique offre la possibilité de définir des fonctions dont le même code peut être exécuté sur des types différents, tandis que le polymorphisme "ad hoc" permet de définir des fonctions exécutant un code différent pour chaque type. Tant la Théorie de la Démonstration que la Sémantique de la première forme de polymorphisme ont été largement étudiées par de nombreux auteurs, sur la base de travaux initiaux de Hindley, Girard, Milner et Reynolds ; cela a conduit à de solides pratiques de programmation. En revanche, la deuxième forme de polymorphisme, habituellement appelée "surcharge" (overloading) n'a reçu que peu d'intérêt théorique (sauf quelques exceptions comme [MOM90], [WB89] ou [Rou90]). Ainsi, la mise en œuvre actuelle de cette forme de polymorphisme, bien que très répandue, n'a pas subi à ce jour, une influence comparable à celle exercée par la théorie du polymorphisme explicite et/ou implicite sur la pratique de la programmation.

Très probablement cela vient du fait que les langages de programmation traditionnels n'offrent qu'une forme très limitée de surcharge : dans la plupart d'entre eux seules des fonctions pré-définies (essentiellement des opérateurs arithmétiques ou d'entrée/sortie) sont surchargées, et les rares langages offrant au programmeur la possibilité de définir ses propres fonctions surchargées décident toujours du sens de celles-ci lors de la compilation. Cette forme de surcharge peut être vue comme une simple abréviation syntaxique qui n'affecte pas de façon significative le langage sous-jacent.

En fait, nous pensons que la surcharge offre un gain réel de puissance dès lors que l'on "calcule avec les types" : afin d'exploiter toutes les potentialités de la surcharge, les types doivent être calculés pendant l'exécution du programme et le résultat de ce calcul doit affecter le résultat final de l'exécution globale. La résolution de la surcharge quand elle est opérée à la compilation n'effectue aucun calcul sur les types : la sélection du code à exécuter se réduit

à l'expansion d'une macro. Dans les langages munis d'une discipline de types "classique", retarder à l'exécution le choix du code n'aurait aucun effet puisque les types ne changent pas pendant le calcul et donc le choix serait toujours le même. Cependant, il existe une large classe de langages de programmation dans lesquels les types évoluent pendant l'exécution. Ceux-la sont les langages qui utilisent des hiérarchies de sous-typage : dans ce cas, les types changent pendant l'exécution, notamment ils décroissent. C'est en ce sens qu'on "calcule avec les types" ; ce calcul ne correspond pas à la réduction d'un terme distingué[1], mais il est intrinsèque à l'exécution du programme. Néanmoins nous pouvons l'utiliser pour affecter le résultat final du programme, simplement en basant la sélection du code d'une fonction surchargée sur le type à un moment donné de l'exécution.

Ainsi, dans les langages qui utilisent une relation de sous-typage on peut déterminer au moins deux disciplines pour la sélection du code d'une fonction surchargée :

1. La sélection basée sur la moindre information de type : les types des arguments à la compilation sont utilisés. Nous appelons cette discipline *liaison précoce* (early binding).

2. La sélection basée sur la meilleure information de type : les types des formes normales des arguments sont utilisés. Nous appelons cette discipline *liaison tardive* (late binding).

Nous avons déjà remarqué que l'introduction de la surcharge avec *liaison précoce* n'affecte pas de manière considérable le langage sous-jacent. Cependant, la possibilité de définir des fonctions surchargées, dès qu'elle est associée avec le sous-typage et la *liaison tardive*, augmente sensiblement les potentialités d'un langage, car elle permet un haut degré de réutilisation du code et donc une programmation de type incrémentale. L'idée intuitive est qu'on peut appliquer une fonction surchargée aux paramètres formels d'une fonction (ordinaire) externe et laisser au système la tâche de sélectionner le code adéquat selon le type des paramètres actuels de la fonction externe. Ce choix doit être effectué pendant l'exécution ; plus précisément après la substitution des paramètres formels par les paramètres actuels. Sans la liaison tardive on serait obligé de définir aussi la fonction extérieure comme surchargée et son corps devrait être dupliqué dans chaque branche[2], tandis que grâce à la liaison tardive ce même code est partagé. Par exemple, considérons trois types différents, A, B et C, avec $B, C \leq A$, et une fonction surchargée f, composée de trois branches f_A, f_B et f_C, une pour chaque type. Imaginons que nous ayons défini une fonction g avec un paramètre formel x de type A, et que dans le corps de g la fonction f soit appliquée à x. En utilisant les contextes du λ-calcul (c-à-d des λ-termes avec un "trou") cela correspond à

$$g = \lambda x\colon A.\mathcal{C}[f(x)] \tag{0.1}$$

où $\mathcal{C}[\ ]$ dénote un contexte. Si l'on utilise la liaison précoce alors le code f_A est toujours utilisé car $x\colon A$; c'est à dire la fonction (0.1) est équivalent à

$$\lambda x\colon A.\mathcal{C}[f_A(x)]$$

Grâce au sous-typage g peut être appliquée aussi à des arguments de type B ou C ; avec la liaison précoce la seule façon d'utiliser le code de f défini pour le type du paramètre actuel

[1] Au moins dans la plupart des langages

[2] Nous appelons *branche* chaque code distinct composant une fonction surchargée

de g est de définir g comme une fonction surchargée de trois branches

$$\begin{array}{rcl} g_A & = & \lambda x.\mathcal{C}[f_A(x)] \\ g_B & = & \lambda x.\mathcal{C}[f_B(x)] \\ g_C & = & \lambda x.\mathcal{C}[f_C(x)] \end{array} \tag{0.2}$$

Si l'on utilise la liaison tardive alors le choix de la branche pour f est accompli quand x a été remplacé par le paramètre actuel. Par conséquent la définition de g dans (0.1) est équivalente à celle de (0.2). Autrement dit, par liaison tardive la fonction g dans (0.1) est implicitement une fonction surchargée avec trois branches ; et grâce à la liaison tardive ces branches virtuelles partagent le code $\mathcal{C}[\,]$ (soit, les branches virtuelles pour B et C réutilisent le code défini pour A).

Dans cette thèse nous proposons une première analyse théorique (donc uniforme et générale) de cette forme plus riche de surcharge. Cependant nous ne présentons pas un traitement exhaustif des fonctions surchargées ; nous développons de façon détaillée une approche purement fonctionnelle centrée sur l'étude de certains mécanismes propres à la programmation orientée objets, tels que l'envoi de messages et le sous-typage, dans le contexte d'un calcul véritablement dépendant des types. Toutefois, l'intérêt de cette étude ne se limite pas aux langages orientés objets. En effet, la surcharge combinée à liaison tardive permet, comme nous venons de le montrer, la réutilisation du code ; ainsi son étude devient intéressante en vue d'une intégration dans d'autres formalismes et/ou contextes (au moment de la rédaction de cette thèse nous étudions son intégration dans le système de modules de SML, dans les langages de programmation pour bases de données et dans ML). En outre, la dépendance de types particulière à la surcharge, alliée au sous-typage revêt un intérêt théorique remarquable.

En fait, cette "dépendance *par* les types" (le fait que le résultat du calcul puisse dépendre des types) ainsi que le rôle joué par la distinction entre type-à-la-compilation et type-à-l'exécution constituent le fil rouge qui lie les différents calculs présentés dans cette thèse. Les différents calculs (d'ordre supérieur) comme le Système F ou ses extensions, permettent d'abstraire par rapport aux types et d'appliquer des termes à ces derniers ; mais la "valeur" de cette application ne dépend pas véritablement du type passé comme argument et, plus généralement, la sémantique d'une expression ne dépend pas des types qu'elle contient. Cette "généricité" ou propriété d'"effacement des types" (type erasure) joue un rôle crucial dans la propriété fondamentale de ces calculs : le théorème d'élimination des coupures. Dans les interprétations sémantiques cette indépendance intrinsèque du calcul par rapport aux types est comprise comme le fait que le sens d'une fonction polymorphe est donné essentiellement par des fonctions constantes.

En revanche, les fonctions surchargées expriment des calculs qui dépendent véritablement des types puisque différentes "branches" de code peuvent être appliquées en fonction des types en entrée. Ainsi, nous sommes en présence d'une nouvelle forme de polymorphisme : la paramétricité caractérise un même code qui opère sur différents types ; la surcharge caractérise un ensemble de codes, un pour chaque type différent. La nouveauté de cette approche est clairement ressentie lorsqu'on se plonge dans l'étude de la sémantique : les modèles existants ne sont plus adéquats et le mélange de la surcharge, de la liaison tardive et du sous-typage ouvre de nouveaux enjeux mathématiques.

Toutefois la motivation principale de cette thèse réside dans le fait de considérer la surcharge comme une façon d'interpréter l'envoi de message dans la programmation orientée objets.

Dans la programmation orienté objets deux façons distinctes de considérer l'envoi de message coexistent :

La première approche considère les objets comme des tableaux qui associent une méthode à chaque message. Lorsque le message `m` est passé à l'objet `obj`, la méthode associée à `m` dans l'objet `obj` est recherchée. Une telle approche est décrite dans la Figure a.

object	
internal_state	
`message_1`	method_1
⋮	⋮
`message_n`	method_n

Figure a.
Objets comme enregistrements.

message_i	
`class_name_1`	method_1
⋮	⋮
`class_name_n`	method_n

Figure b.
Messages comme fonctions surchargées.

Ce premier point de vue a été largement étudié et correspond à l'analogie "objets comme enregistrements" introduite dans [Car88] ; dans ce contexte les objets sont des enregistrements (bien sûr!) dont les étiquettes sont les messages et dont les champs contiennent les méthodes correspondantes. L'envoi de message correspond alors à l'extraction de champ.

La seconde approche considère les messages comme des identificateurs de fonctions particulières et l'envoi de message comme leur application. Si, dans le contexte des langages typés, nous supposons que le type d'un objet est (le nom de) sa classe alors les messages sont des identificateurs de fonctions surchargées : la méthode est choisie selon la classe (ou, plus généralement, le type) de l'objet auquel le message est passé (voir Figure b). Ainsi nous renversons, dans un certain sens, la situation précédente : au lieu d'envoyer des messages aux objets nous envoyons des objets aux messages.

D'emblée, cette deuxième approche semble posséder certains avantages par rapport à la première, au moins sur le plan d'une étude théorique du cas typé. Ceci est vrai en particulier pour les *multi-méthodes*, le *dispatch multiple* ou pour l'*indépendance logique* des donnés persistantes comme dans les langages de programmation pour les bases de données[3]. En outre, elle clarifie le rôle de la *covariance* et de la *contra-variance* dans la règle de sous-typage pour les méthodes.

Par ailleurs, d'autres problèmes surgissent dès que l'on utilise les fonctions surchargées pour modéliser les méthodes. Particulièrement, ceci se produit quand on souhaite intégrer la redéfinition dynamique de nouvelles classes et un haut niveau d'"encapsulation" ; ce dernier point par exemple rend ce formalisme inapte à la modélisation des objets dans les systèmes distribués à grand échelle (WADS) pour lesquels les objets doivent encapsuler les méthodes (pour des raisons évidentes de sécurité et d'efficacité).

Un regard plus attentif au modèle basé sur la surcharge nous persuade que le style de

[3]Dans le sens qu'il est possible d'ajouter des nouvelles méthodes pour les objets d'une classe donnée sans perturber la définition de leurs types et donc le bon typage des applications écrites dans l'ancien schéma

programmation qu'il modélise est tout à fait diffèrent de celui engendré par le modèle basé sur les enregistrements. Le problème est que le terme "orienté objets" regroupe sous un même chapeau de nombreuses techniques différentes. En effet, sous ce terme cohabitent différents styles de programmation dont l'affinité minimale est capturée par les trois termes : "objet", "envoi de message" et "héritage". Vouloir pousser plus loin la similarité en incluant d'autres "mots magiques" tels que "encapsulation" ou "modularité" exclurait des classes significatives de langages (e.g. CLOS pour la modularité et Simula pour l'encapsulation). Ces "mots magiques" partitionnent l'ensemble des langages objets en différents styles le composant.

La recherche dans le domaine de la théorie des types s'est jusqu'à présent préoccupée de la partition caractérisée par le mot clef "encapsulation des méthodes" et modélisée par les enregistrements. À partir de [Car88] (déjà paru en 1984) toutes les études théoriques dans le domaine se fondèrent sur l'hypothèse que les méthodes d'un objet étaient encapsulées dans celui-ci. Ceci excluait des mécanismes tels que les multi-méthodes et le dispatch multiple, présents dans certains langages orientés objets mais pour lesquels ce type de modèles était inadéquat.

Au début de notre travail nous pensions que les modèles existants n'était pas assez puissants pour capturer ces mécanismes. C'est pourquoi nous avons commencé à chercher un modèle véritablement neuf. À partir des idées de [Ghe91], nous avons établi la base de ce modèle en définissant le $\lambda\&$-calcul [CGL92b]. Mais portant un regard plus attentif aux mécanismes que nous avions modélisés, nous nous sommes aperçu que nous avions décrit un style de programmation complètement différent de celui exposé par les enregistrements. Les modèles par les enregistrements n'étaient pas mis en défaut par celui que nous avions défini mais plus simplement indépendants de celui-ci : à différents mécanismes différents modèles.

Le "nouveau" style de programmation orienté objets que nous avons modélisé correspondait à celui des fonctions génériques. Il est intéressant de constater qu'à partir d'une approche purement théorique nous avons obtenu un modèle de programmation déjà existant. En fait, nous nous sommes bientôt rendu compte qu'à la relation

enregistrement	↔	objet
champ	↔	méthode
étiquette	↔	message

de l'approche "objets comme enregistrements" correspond la relation

fonction surchargée	↔	fonction générique
branche	↔	méthode

de notre approche. Dans les deux cas le passage de la théorie à la pratique a permi d'obtenir une discipline de typage (dont la correction peut être formellement prouvée!). Mais comme pour le modèle par enregistrements les bénéfices résultant de la définition d'un modèle typé ne se réduisent pas à l'obtention d'une discipline de typage : l'étude du modèle nous suggère d'introduire de nouveaux mécanismes dans les langages orientés objets (par exemple les messages de première classe) ou de généraliser ou redéfinir les mécanismes existants (par exemple les coercitions explicites).

Cette thèse est une étude exhaustive de la surcharge combinée à la liaison tardive dans la

perspective particulière de définir ce nouveau modèle, et s'attache également à présenter l'impact pratique qu'un tel modèle peut avoir sur la définition des langages orientés objets et leurs disciplines de types.

La thèse est composée de deux parties principales : la première se concentre sur la surcharge pour laquelle la dépendance de types est *implicite*, dans le sens où la sélection de la branche est déterminée par le type *de* l'argument de la fonction surchargée. La second partie est consacrée à l'étude de la dépendance de type *explicite* de la surcharge, dans le sens où la sélection de la branche est déterminée par le type qui *est* l'argument de la fonction surchargée.

Nous détaillons dans les sections suivantes le contenu des chapitres de la thèse.

Programmation orientée objets

Un programme orienté objet est construit à partir d'*objets*. Un objet est une unité de programmation qui associe des données avec les opérations qui peuvent utiliser ou modifier ces données. Ces opérations sont appelés *méthodes* ; les données sur lesquelles elles opèrent sont les *variables d'instance* des objets. Les variables d'instance d'un objet sont privées, leur emploi est limité à l'objet même : on ne peut y accéder que par les méthodes de l'objet. Un objet est seulement capable de répondre à des *messages* qui lui sont *envoyés* ou *passés*. Un message est le nom d'une méthode définie pour l'objet en question.

Le passage de message est le mécanisme de base de la programmation orientée objet. En fait, un programme orienté objets consiste en un ensemble d'objets qui interagissent en s'échangeant des messages. Chaque langage possède sa propre syntaxe pour le passage de message. Nous utilisons la notation suivante :

[*destinataire message*]

Le *destinataire* est un objet ou une expression calculant un objet ; lors de l'envoi d'un message, le système sélectionne entre les méthodes définies pour l'objet en question, celle dont le nom correspond au message ; l'existence de cette méthode doit être vérifiée statiquement (c'est-à-dire lors de la compilation) par un programme de vérification des types. Nous avons déjà remarqué qu'une manière de comprendre le passage de message est de le considérer comme la sélection d'un champ d'un enregistrement.

Dans cette thèse, au contraire, on considère le passage de message comme l'application d'une fonction, où le message est (l'identificateur de) la fonction et le destinataire son argument (cette technique est utilisée par les langages CLOS [DG87] et Dylan [App92]). Toutefois les fonctions ordinaires ne suffisent pas à formaliser cette approche. Le fait qu'une méthode *appartient* à un objet spécifique implique que la sémantique du passage de message est tout à fait différente de celle de l'application ordinaire. Deux caractéristiques différencient les messages des fonctions :

1. *Surcharge* : Deux objets peuvent répondre d'une manière différente au même message. Toutefois tous les objets d'une même classe répondent à un message de la même façon.[4]

[4]Cela n'est pas vrai dans les langages objets basés sur la délégation (delegation based).

Sous l'hypothèse que le type d'un objet est sa classe, cela revient à dire que *les messages dénotent des fonctions surchargées*, du moment que le code à exécuter est choisi sur la base du type de l'argument. Chaque méthode associée à un message m constitue une branche de la fonction surchargée dénotée par m.

2. *Liaison tardive* : La deuxième différence entre l'application d'une fonction et le passage d'un message est que la fonction est liée à son exécutable au moment de la compilation, tandis qu'un message est lié à la méthode à exécuter seulement pendant l'exécution, lorsque le destinataire est complètement connu. Cette caractéristique, appelée liaison tardive, est un des traits saillants de la programmation orientée objets. Dans notre approche elle naît de la combinaison de la surcharge et du sous-typage. On peut reformuler l'exemple de la section précédente : supposons que les classes *Cercle* et *Carré* soient sous-types de la classe *Figure* et que les trois classes aient une méthode pour le message *dessine*. Si l'on utilise la liaison précoce, le passage du message suivant

$$\lambda x^{Figure}.(\dots\ [x\ dessine]\ \dots)$$

est toujours effectué en utilisant la méthode définie pour les figures. En revanche, par liaison tardive la méthode est choisie après que la fonction a été appliquée, selon que x est lié à un cercle, à un carré ou à une figure.

Pour commencer une étude formelle à partir de cette intuition, nous définissons une extension du λ-calcul simplement typé capable de modéliser ces deux mécanismes : surcharge et liaison tardive.

Le $\lambda\&$-calcul

Une fonction surchargée est formée par un ensemble de fonctions ordinaires (i.e. des λ-abstractions), chacune constituant une branche différente. Pour relier ces fonctions nous avons choisi le symbole & (d'où le nom du calcul) ; donc nous enrichissons les termes du λ-calcul simplement typé par le terme suivant :

$$(M\&N)$$

qui, intuitivement, dénote une fonction surchargée avec deux branches, M et N, qui seront sélectionnées selon le type de l'argument.

On doit distinguer l'application ordinaire de l'application d'une fonction surchargée, car elles représentent deux mécanismes différents[5]. Ainsi nous utilisons "$\bullet$" pour dénoter une "application surchargée" et "$\cdot$" pour une application ordinaire.

Nous construisons les fonctions surchargées comme des listes, c'est-à-dire en partant d'une fonction surchargée *vide* dénotée par ε, et en concaténant de nouvelles branches par &. Donc dans le terme précédent M est une fonction surchargée et N une fonction ordinaire (une branche). Ainsi, le terme

$$((\dots((\varepsilon\&M_1)\&M_2)\dots)\&M_n)$$

[5] À la première est associée une substitution, à la deuxième une sélection.

dénote une fonction surchargée avec n branches $M_1, M_2, \ldots, M_n$.

Le type d'une fonction surchargée est l'ensemble des types de ses branches. Donc nous ajoutons aux types du λ-calcul simplement typé des ensembles de flèches. Ainsi, si $M_i\colon U_i \to V_i$ alors la fonction surchargée ci-dessus a le type

$$\{U_1 \to V_1, U_2 \to V_2, \ldots, U_n \to V_n\}$$

et si l'on applique cette fonction à un argument de type U_j on sélectionnera la branche M_j, soit

$$(\varepsilon \& M_1 \& \ldots \& M_n) \bullet N \quad \rhd^* \; M_j \cdot N \tag{0.3}$$

où $\rhd^*$ signifie "réécrit en zéro ou plusieurs pas".

Nous définissons sur les types une relation de sous-typage. Intuitivement $U \leq V$ si tout terme de type U peut être utilisé "type safely" là où un terme de type V est requis. Donc un calcul ne produira pas d'erreurs de type tant qu'il maintiendra ou réduira les types des termes. La relation de sous-typage pour les types flèches est bien connue (covariance à droite et contra-variance à gauche) ; la relation pour les types surchargés est déduite du fait qu'une fonction surchargée peut en remplacer une autre si pour toute branche de la seconde il y en a une de la première capable de la remplacer.

Avec le sous-typage, le type de N dans (0.3) peut ne pas correspondre à un des U_i mais être un sous-type de l'un d'entre eux. Dans ce cas on sélectionne la branche dont le type U_i approche au mieux le type U de N, c'est-à-dire on sélectionne la branche j telle que $U_j = \min\{U_i | U \leq U_i\}$.

Dans notre système les ensembles de flèches ne sont pas tous des types surchargés. En fait un ensemble de types flèche $\{U_i \to V_i\}_{i \in I}$ est un type surchargé si et seulement si pour touts i, j dans I il satisfait les conditions suivantes

$$U_i \leq U_j \quad \Rightarrow \quad V_i \leq V_j \tag{0.4}$$

$$U_i \Downarrow U_j \quad \Rightarrow \quad \text{il existe un unique } z \in I \text{ t.q. } U_z = \inf\{U_i, U_j\} \tag{0.5}$$

où $T_1 \Downarrow T_2$ dénote l'existence d'un minorant commun pour les types T_1 et T_2.

La condition (0.4) assure que pendant l'exécution les types ne peuvent que décroître. Dans un sens elle prend en compte une certaine nécessité de covariance pour les flèches dans la pratique de la programmation. Plus précisément, considérons une fonction surchargée M de type $\{U_1 \to V_1, U_2 \to V_2\}$ où $U_2 < U_1$. Si l'on applique M à un terme N ayant à la compilation le type U_1 alors le type de $M \bullet N$ lors de la compilation sera V_1. Mais si la forme normale de N a le type U_2 (ce qui est tout à fait possible étant donné que $U_2 < U_1$) alors le type de $M \bullet N$ à l'exécution sera V_2 et donc la condition $V_2 < V_1$ doit être vérifiée.

La condition (0.5) concerne la sélection d'une branche. On rappelle que pour l'application d'une fonction de type $\{U_i \to V_i\}_{i \in I}$ à un argument de type U on sélectionne la branche de type $U_j \to V_j$ telle que $U_j = \min_{i \in I}\{U_i | U \leq U_i\}$. (0.5) est une condition suffisante pour l'existence de ce minimum.

Jusqu'à présent nous avons montré comment inclure la surcharge et le sous-typage. Il manque encore la liaison tardive. Une façon très simple de l'obtenir est d'imposer qu'une réduction comme (0.3) ne soit effectuée que si N est fermé et en forme normale.

La description formelle de $\lambda\&$ peut se résumer de la façon suivante :

Pré-types

$$V ::= A \mid V \to V \mid \{V_1' \to V_1'', \ldots, V_n' \to V_n''\}$$

Sous-typage
La relation de sous-typage est pré-définie sur les types atomiques et elle est étendue aux (pré-)types supérieurs de la façon suivante :

$$\frac{U_2 \leq U_1 \quad V_1 \leq V_2}{U_1 \to V_1 \leq U_2 \to V_2} \qquad \frac{\forall i \in I, \exists j \in J \qquad U_j' \to V_j' \leq U_i'' \to V_i''}{\{U_j' \to V_j'\}_{j \in J} \leq \{U_i'' \to V_i''\}_{i \in I}}$$

Types

1. $A \in$ **Types**
2. si $V_1, V_2 \in$ **Types** alors $V_1 \to V_2 \in$ **Types**
3. si pour tout $i, j \in I$
   (a) $(U_i, V_i \in$ **Types**$)$ et
   (b) $(U_i \leq U_j \Rightarrow V_i \leq V_j)$ et
   (c) $(U_i \Downarrow U_j \;\; \Rightarrow \;\; \exists! h \in I \;.\; U_h = \inf\{U_i, U_j\})$

   alors $\{U_i \to V_i\}_{i \in I} \in$ **Types**

Termes (où V est un type)

$$M ::= x^V \mid \lambda x^V.M \mid M \cdot M \mid \varepsilon \mid M \&^V M \mid M \bullet M$$

Typage

[TAUT] $\quad x^T : T$ $\qquad$ [TAUT$_\varepsilon$] $\quad \varepsilon : \{\}$

[$\to$INTRO] $\quad \dfrac{M : T}{\lambda x^U.M : U \to T}$ $\qquad$ [$\{\}$INTRO] $\quad \dfrac{M : W_1 \leq \{U_i \to T_i\}_{i \leq (n-1)} \quad N : W_2 \leq U_n \to T_n}{(M \&^{\{U_i \to T_i\}_{i \leq n}} N) : \{U_i \to T_i\}_{i \leq n}}$

[$\to$ELIM$_\leq$] $\quad \dfrac{M : U \to T \quad N : W \leq U}{M \cdot N : T}$ $\qquad$ [$\{\}$ELIM] $\quad \dfrac{M : \{U_i \to T_i\}_{i \in I} \quad N : U \quad U_j = \min_{i \in I}\{U_i | U \leq U_i\}}{M \bullet N : T_j}$

où tout type apparaissant dans les règles est un type bien formé (i.e. appartient à **Type**)

Réduction
La réduction $\triangleright$ est la *fermeture compatible* de la *notion de réduction* suivante :

β) $(\lambda x^T.M)N \triangleright M[x^T := N]$

$\beta_\&$) Si $N : U$ est clos et en forme normale, et $U_j = min_{i=1..n}\{U_i | U \leq U_i\}$ alors

$$(M_1 \&^{\{U_i \to T_i\}_{i=1..n}} M_2) \bullet N \triangleright \begin{cases} M_1 \bullet N & \text{for } j < n \\ M_2 \cdot N & \text{for } j = n \end{cases}$$

Théorèmes Principaux

- *Elimination de la subsumption* : Le langage admet aussi une présentation équivalente avec la règle de subsumption ($M\!:\!W$ et $W\!\leq\!U$ impliquent $M\!:\!U$)

- *Élimination de la transitivité* : L'ajout de la transitivité ne modifie pas la relation de sous-typage.

- *Unicité du type* : Chaque terme bien typé possède un seul type

- *Subject Reduction Généralisée* : Soit $M\!:\!U$. Si $M \;\triangleright^* N$ alors $N\!:\!U'$, et $U' \leq U$.

- *Confluence* : Des termes égaux possèdent un *reductum* commun.

À ce point il est intéressant de voir comment utiliser en première approximation ce calcul pour modéliser les langages objets. Tout d'abord il faut noter que dans $\lambda\&$ il est possible d'encoder, soit les paires surjectives (surjective pairings), soit les enregistrements simples (ceux de [Car88]), soit les enregistrements extensibles (voir [Wan87, Rém89, CM91]).

Les conditions (0.4) et (0.5) ont dans les langages objets une interprétation très naturelle : supposons que *mesg* soit l'identificateur d'une fonction surchargée ayant le type suivant

$$mesg\ \ : \{C_1 \to T_1, C_2 \to T_2\}$$

Selon la terminologie orientée objet *mesg* est un message qui dénote deux méthodes, l'une définie dans la *classe* C_1 et retournant le type T_1, l'autre dans la classe C_2 et retournant le type T_2. Si C_1 est une sous-classe de C_2 (plus précisément un sous-type : $C_1 \leq C_2$) alors la méthode de C_1 *masque* (overrides) celle de C_2. La condition (0.4) impose alors que $T_1 \leq T_2$, c'est-à-dire la méthode qui en *masque* une autre doit retourner un type plus petit. Si par contre C_1 et C_2 ne sont pas en relation mais qu'il existe une classe C_3 sous-classe des deux ($C_3 \leq C_1, C_2$) alors C_3 a été définie par *héritage multiple* de C_1 et C_2. La condition (0.5) impose qu'une branche soit définie dans *mesg* pour C_3, c'est-à-dire qu'en cas d'héritage multiple les méthodes définies dans plus d'un ancêtre doivent être redéfinies.

Voyons ceci sur un exemple. Considérons la classe `2DPoint` avec deux variables d'instance entières `x` et `y`, et la classe `3DPoint` sous-classe de la première qui possède en plus la variable d'instance `z`. Ceci peut s'exprimer par les définitions suivantes

```
class 2DPoint
  {
    x:Int;
    y:Int;
  }
     :
     :
```

```
class 3DPoint is 2DPoint
  {
    x:Int;
    y:Int;
    z:Int
  }
     :
     :
```

où à la place des pointillés se trouvent les définitions des méthodes. En première approximation cela peut être modélisé en $\lambda\&$ par deux types atomiques *3DPoint* et *2DPoint* avec

3DPoint$\leq$*2DPoint* et dont les types-représentation sont respectivement les types enregistrements $\langle\!\langle x : \mathrm{Int}; y : \mathrm{Int}; z : \mathrm{Int}\rangle\!\rangle$ et $\langle\!\langle x : \mathrm{Int}; y : \mathrm{Int}\rangle\!\rangle$. Il faut noter que *3DPoint*$\leq$*2DPoint* est compatible avec le sous-typage des types-représentation correspondants.

Une première méthode que l'on pourrait rencontrer dans la définition de `2DPoint` est

```
norme = sqrt(self.x^2 + self.y^2)
```

masquée (overriden) dans `3DPoint` par la méthode suivante

```
norme = sqrt(self.x^2 + self.y^2 + self.z^2)
```

Dans $\lambda\&$ cela est obtenu par une fonction surchargée avec deux branches

$$\begin{array}{rl} \mathit{norme} \equiv & (\ \lambda \mathit{self}^{\,2DPoint}.\sqrt{\mathit{self}.x^2 + \mathit{self}.y^2} \\ & \&\ \lambda \mathit{self}^{\,3DPoint}.\sqrt{\mathit{self}.x^2 + \mathit{self}.y^2 + \mathit{self}.z^2} \\ &) \end{array}$$

dont le type est $\{\mathit{2DPoint} \to \mathit{Real}\ ,\ \mathit{3DPoint} \to \mathit{Real}\}$. Il faut noter que `self`, qui dans les méthodes dénote le destinataire du message, est dans $\lambda\&$ le premier paramètre de la fonction surchargée, i.e. le paramètre dont la classe déterminera la sélection.

La covariance apparaît par exemple lorsqu'on définit une méthode qui modifie les variables d'instance. Ainsi, une méthode qui initialise les variables d'instance aura le type suivant

$$\mathit{initialise}\ \ : \{\mathit{2DPoint} \to \mathit{2DPoint}\ ,\ \mathit{3DPoint} \to \mathit{3DPoint}\}$$

Supposons que nous ayons défini une nouvelle classe *ColorPoint* par héritage multiple de *2DPoint* et *Color* et que ces deux classes définissent une méthode *efface* [6]

$$\begin{array}{rl} \mathit{efface} \equiv & (\ \lambda \mathit{self}^{\,2DPoint}.\langle \mathit{self}\ \leftarrow x = 0\rangle \\ & \ \ \&\ \lambda \mathit{self}^{\,Color}.\langle \mathit{self}\ \leftarrow c = \text{“white”}\rangle \\ &) \end{array}$$

Une telle définition n'est pas bien typée, car $\{\mathit{2DPoint} \to \mathit{2DPoint}\ ,\ \mathit{Color} \to \mathit{Color}\}$ ne satisfait pas la condition (0.5) ; en réalité en appliquant *efface* à un objet de classe *ColorPoint* on ne saurait pas quelle méthode choisir. Par conséquent la condition (0.5) impose l'ajout d'une méthode pour *ColorPoint* et donc

$$\mathit{efface} : \{\mathit{2DPoint} \to \mathit{2DPoint}\ ,\ \mathit{Color} \to \mathit{Color}\ ,\ \mathit{ColorPoint} \to \mathit{ColorPoint}\}$$

L'héritage dans ce cadre est donné par le sous-typage plus la règle de sélection des branches : par exemple si l'on applique *norme* à un objet de classe *ColorPoint*, la méthode exécutée sera celle définie pour *2DPoint*. Plus généralement, si l'on passe un message de type $\{C_i \to T_i\}_{i\in I}$ à un objet de classe C la méthode exécutée sera celle définie dans la classe $\min_{i=1..n}\{C_i | C \leq C_i\}$. Si ce minimum est exactement C cela signifie que le destinataire utilise la méthode définie dans sa classe ; si le minimum est strictement plus grand que C alors le destinataire utilise la méthode que sa classe, C, a *hérité* de ce minimum. Il faut noter que la recherche

[6] Pour simplifier les exemples nous supposons avoir des conversions implicites entres un type atomique et son type-représentation.

du minimum correspond précisément au "method look-up" de Smalltalk où l'on recherche la plus petite super-classe (de la classe du destinataire) pour laquelle une certaine méthode a été définie.

L'un des avantages de modéliser les messages par des fonctions surchargées est que, ces dernières étant des valeurs de première classe, les messages sont aussi de première classe. Il devient donc possible d'écrire des fonctions (même surchargées) qui prennent comme argument un message ou le rendent comme résultat. Par exemple on peut écrire la fonction suivante :

$$\lambda m^{\{C \to C\}}.\, \lambda x^{C}.\, (m \bullet (m \bullet x))$$

qui accepte comme argument un message m pouvant être envoyé à un objet de classe C (c'est-à-dire qui a *au moins* une branche qui puisse travailler avec des objets de classe C), un objet de classe (au moins) C et qui envoie deux fois le message m à l'objet x (bien sûr le résultat du message doit être un objet de classe plus grande que C)

Mais l'avantage le plus intéressant d'utiliser cette forme de programmation objet est la possibilité de pouvoir utiliser le dispatch multiple[7] : un des problèmes majeurs de l'approche avec enregistrements réside dans l'impossibilité de combiner de manière satisfaisante le sous-typage avec les méthodes binaires, c'est-à-dire les méthodes qui ont un paramètre de la même classe que celle du destinataire. Par exemple dans les modèles basés sur les enregistrements les points et les points colorés avec une méthode d'égalité sont modélisés par les enregistrements récursifs suivants :

$$EqPoint \equiv \langle\!\langle x\colon \text{Int};\ y\colon \text{Int};\ equal\colon EqPoint \to \text{Bool} \rangle\!\rangle$$
$$ColEqPoint \equiv \langle\!\langle x\colon \text{Int};\ y\colon \text{Int};\ c\colon \text{String};\ equal\colon ColEqPoint \to \text{Bool} \rangle\!\rangle$$

À cause de la contra-variance de la flèche le type d'*equal* dans *ColEqPoint* n'est pas inférieur à celui d'*equal* dans *EqPoint* et donc $ColEqPoint \not\leq EqPoint$. Considérons maintenant le même exemple dans λ&. Nous avons déjà rencontré les types atomiques *2DPoint* et *ColorPoint*. On peut continuer à les utiliser parce que, contrairement à ce qu'il se passe avec les enregistrements, l'ajout d'une méthode à une classe ne change pas le type des instances. En λ& une définition telle que

$$equal\colon \{2DPoint \to (2DPoint \to \text{Bool})\ ,\ ColorPoint \to (ColorPoint \to \text{Bool})\}$$

ne possède pas un type bien formé : $ColorPoint \leq 2DPoint$ donc la condition (0.4) nécessite $ColorPoint \to \text{Bool} \leq 2DPoint \to \text{Bool}$ ce qui n'est pas vrai à cause de la contra-variance de la flèche. Il faut noter qu'une telle fonction choisirait la branche sur la base du type du premier argument seulement. Or, le code de *equal* ne peut être choisi que dès que l'on connaît les types des deux arguments. C'est pourquoi on ne veut pas accepter le type ci-dessus (d'ailleurs il serait très facile d'écrire un terme engendrant une erreur). Toutefois dans λ& il est possible d'écrire une fonction qui prenne en compte les types de ses deux arguments pour effectuer la sélection. Pour *equal* ceci est obtenu de la façon suivante

$$equal\colon \{(2DPoint \times 2DPoint) \to \text{Bool}\ ,\ (ColorPoint \times ColorPoint) \to \text{Bool}\}$$

[7] C'est-à-dire la possibilité de sélectionner une méthode en tenant compte de classes autres que celle du destinataire du message

Si l'on passe à cette fonction deux objets de classe *ColorPoint* alors la deuxième branche est choisie ; lorsque l'un des deux arguments est de classe *2DPoint* (et l'autre d'une classe plus petite ou égale à *2DPoint*) la première branche sera choisie.

Une autre caractéristique intéressante de ce modèle est qu'il permet d'ajouter une méthode à une classe C déjà existante sans modifier le type de ses objets. En fait, si la méthode doit répondre au message m, il suffit d'ajouter une branche pour le type C à la fonction surchargée dénotée par m. Il est important de noter que la nouvelle méthode est immédiatement disponible pour toutes les instances de C et il est donc possible d'envoyer le message m à un objet de classe C, même si cet objet a été défini *avant* la branche de m pour C. Cet aspect est très important quand on a à gérer des données persistantes, car on peut modifier le schéma logique des donnés (en ajoutant des fonctionnalités) sans devoir modifier les applications déjà écrites.

Un des apports de ce modèle est de clarifier les rôles qui jouent la covariance et la contra-variance dans le sous-typage (*equal* en constitue un joli exemple) : la contra-variance est la règle à utiliser lorsqu'on substitue une fonction par une autre de type différent ; la covariance est la règle à utiliser lorsqu'on spécialise une branche d'une fonction surchargée par une autre. Il est important de noter que dans ce cas la nouvelle branche *ne se substitue pas* à l'ancienne branche mais plutôt la *masque* aux objets de certaines classes. En fait, notre formalisation montre très clairement que la question "contra-variance ou covariance" était un faux problème dû au mélange de deux mécanismes qui n'ont rien à voir l'un avec l'autre : la substitutivité et le masquage. La substitutivité nous indique quand il est possible d'utiliser une expression d'un certain type S *à la place* d'une expression du type T. Cette information est utilisée par l'application : soit f une fonction de type $T \to U$, on veut identifier une catégorie de types dont les valeurs peuvent être passées comme arguments à f ; il faut noter que ces arguments *se substitueront* dans le corps de la fonction, au paramètre formel qui a le type T . Pour cela nous définissons une rélation de sous-typage telle que f accepte tout argument ayant un type S plus petit que T. La catégorie en question est ainsi l'ensemble des sous-types de T. En particulier, si T est de la forme $T_1 \to T_2$ il se peut que dans le corps de f le paramètre formel de la fonction soit appliqué à une expression de type T_1 ; de ceci on déduit deux choses : le paramètre actuel doit lui aussi être une fonction (donc si $S \leq T_1 \to T_2$ alors S doit être de la forme $S_1 \to S_2$), et en plus il doit être une fonction à laquelle on puisse passer des arguments de type T_1 (et donc $T_1 \leq S_1$, eh oui!... contravariance). Evidemment, si l'on ne souhaite pas passer des fonctions comme argument, il n'y a aucun sens à définir la relation de sous-typage pour les flèches (c'est pourquoi O_2 [BDe92] fonctionne très bien même sans la contra-variance). Le masquage correspond à un tout autre phénomène : on a un identificateur m (en l'occurrence un *message*) qui identifie par exemple deux fonctions $f : A \to C$ et $g : B \to D$ (A et B incomparables) ; cet identificateur peut être appliqué à une expression e ; cette application est résolue par le passage de e à f si cette expression a un type plus petit que A (dans le sens de la substitutivité qu'on vient d'expliquer), à g si le type est plus petit que B. Supposons à présent que $B \leq A$; la résolution dans ce cas sélectionne f si e a un type compris entre A et B, g si le type de e est plus petit ou égal à B ; mais il y a un problème supplémentaire : les types peuvent diminuer pendant l'exécution ; donc il se pourrait que le type checker voit e de type A et pense que m appliquée à e retournera le type C (f est sélectionnée) ; mais si pendant l'exécution le type de e diminue jusqu'à

B, l'application a le type D ; mais alors D doit être un type qui puisse se substituer à C (dans le sens de la substitutivité plus en haut), i.e. $D \leq C$. On peut appeler ceci covariance, si l'on veut, mais il doit être clair qu'il ne s'agit pas d'une règle de sous-typage : g ne se substitue pas à f car g ne sera jamais appliquée à des arguments de type A ; g et f sont deux fonctions indépendantes ayant des tâches très précises : f travaille avec les arguments de m ayant un type compris entre A et B, g avec ceux de type plus petit ou égal à B. Il n'est pas question de définir la substitutivité, mais de donner une règle de formation pour un ensemble de fonctions dénoté par un identificateur unique, de façon à assurer la consistance des types pendant l'exécution.

D'une façon plus pratique : une méthode a des paramètres ; la classe de chaque paramètre peut ou ne pas être prise en compte pour la sélection de la méthode. Lorsqu'on masque (override) cette méthode, les paramètres dont le type est pris en compte pour la sélection doivent être masqués de manière covariante (i.e. dans la nouvelle méthode les paramètres correspondants doivent posséder un type plus petit), les autres de manière contra-variante (i.e. dans la nouvelle méthode les paramètres correspondants doivent posséder un type plus grand). Dans les modèles basés sur les enregistrements aucun argument est pris en compte pour la sélection : la méthode est déterminée par l'enregistrement dont on sélectionne le champ. Par conséquent, dans de tels modèles il ne peut y avoir que de la contra-variance et le dispatch multiple est impossible.

Pour être plus précis, le modèle par enregistrements possède une forme très limitée de covariance mais qui est cachée par le codage des objets : considérons une étiquette ℓ ; pour la règle de sous-typage des enregistrements, si l'on "envoie" cette étiquette à deux enregistrements de types S et T avec $S \leq T$ alors le type associé à ℓ dans S doit être un sous-type de celui associé à ℓ dans T. Cela correspond exactement à notre règle de covariance, mais sa forme est bien plus limitée parce qu'elle ne s'applique qu'aux types enregistrements (puisqu'il s'agit d'un "envoi" d'étiquettes), et non aux produits cartesiens (i.e. dispatch multiple) ni aux flèches.

Normalisation Forte

Le $\lambda\&$-calcul n'est pas fortement normalisant. Cela provient du fait qu'on peut y typer l'auto-application. Considérons le type suivant

$$\{\,\{\,\} \rightarrow T\,\}$$

où T est n'importe quel type. Ceci est le type d'une fonction surchargée qui accepte tout argument ayant un type plus petit ou égal à $\{\}$ (le type surchargé vide). Mais remarquez que tout type surchargé est plus petit que $\{\}$, donc en particulier $\{\,\{\,\} \rightarrow T\,\} \leq \{\}$. Dès lors une fonction de type $\{\,\{\,\} \rightarrow T\,\}$ accepte comme argument des termes du même type que le sien et donc soi même ; par conséquent

$$\lambda x^{\{\,\{\,\}\rightarrow T\,\}}.x\bullet x$$

a pour type $\{\,\{\,\} \rightarrow T\} \rightarrow T$. Il est donc facile de définir un terme sans forme normale, en imitant les constructions classiques du λ-calcul ; soit

$$\omega_T \;=\; (\mathcal{E}_T \,\&^{\{\,\{\}\rightarrow T\,,\,\{\,\{\}\rightarrow T\,\}\rightarrow T\,\}}\,(\lambda x^{\{\{\}\rightarrow T\}}.x\bullet x))$$

où $\mathcal{E}_T$ est n'importe quel terme clos de type $\{\{\} \to T\}$; le terme $\Omega_T \equiv \omega_T \bullet \omega_T$ qui a type T ne possède pas de forme normale. On peut également définir pour tout type T un opérateur de point fixe $\mathbf{Y}_T : (T \to T) \to T$ de la façon suivante

$$\mathbf{Y}_T \equiv \lambda f^{T \to T}.((\mathcal{E}_T \&^{\{\{\} \to T, \{\{\} \to T\} \to T\}} \lambda x^{\{\{\} \to T\}}.f(x \bullet x)) \bullet (\mathcal{E}_T \&^{\{\{\} \to T, \{\{\} \to T\} \to T\}} \lambda x^{\{\{\} \to T\}}.f(x \bullet x)))$$

La raison de ce phénomène doit être recherchée dans la définition particulière de la relation de sous-typage. La circularité typique de l'auto-application dérive de la possibilité de pouvoir associer deux types d'une structure syntaxique différente ; plus précisément, deux types ayant des arbres syntaxiques de profondeur différente peuvent être en relation, comme c'est le cas pour $\{\}$ et $\{\{\} \to T\}$. Il est donc possible qu'un type soit sous-type d'une de ses occurrences, d'où l'auto-application[8].

Toutefois dans notre modélisation de la programmation objet cette forme de circularité n'est jamais utilisée. Il peut donc être intéressant d'étudier des sous-systèmes de $\lambda\&$ qui soient à la fois fortement normalisant et assez expressifs pour modéliser les langages orientés objet. Dans ce but on démontre le théorème suivant :

Théorème 1 *Soit $\lambda\&^-$ n'importe quel sous-système de $\lambda\&$ fermé par réduction ; soit* rank *une fonction associant des entiers aux types de $\lambda\&^-$ telle que si T est une occurrence syntaxique de U alors $rank(T) \leq rank(U)$. Si dans $\lambda\&^-$ pour toute application $M^T N^U$ bien typée $rank(T) \leq rank(U)$ alors $\lambda\&^-$ est fortement normalisant.*

Nous qualifions comme *stratifiés* les systèmes qui satisfont les hypothèses du théorème. Un exemple de système stratifié est obtenu en émondant la relation de sous-typage définie sur les types bien formés de $\lambda\&$ des couples formés par des types de profondeur différente (rank(T) est dans ce cas la profondeur de l'arbre syntaxique de T).

Ces sous-systèmes sont utilisés pour l'étude de la sémantique.

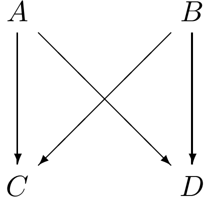
Trois variations sur le thème

On peut à ce point étudier comment modifier, étendre ou reformuler $\lambda\&$ pour l'adapter à des exigences spécifiques.

Première variante : $\lambda\&^+$

La première variante consiste à donner plus de liberté à $\lambda\&$ en introduisant certaines modifications qui ont été suggérées en essayant de traduire des langages orientés objets dans $\lambda\&$.

[8] Dans les enregistrements on compare aussi des types d'arbre syntaxique différents, mais les sous-arbres de profondeurs differentes sont toujour séparés par des étiquettes différentes



Tout d'abord la condition (0.5) est trop restrictive pour l'héritage multiple : considérons quatre classes A, B, C et D telles que C et D soient toutes deux sous-types de A et B (comme dans la figure ci-contre) ; soit m un message défini dans les quatre classes, i.e. $m\colon \{A \to ..., B \to ..., C \to ..., D \to ...\}$. Le domaine de m ne respecte pas la condition (0.5) parce que A et B n'ont pas de plus petite borne inférieure. Cependant il n'y aura pas d'ambiguïté dans le choix de la branche. Par conséquent, (0.5) est trop restrictive pour modéliser l'héritage multiple tel qu'il est utilisé dans la programmation objets.

La condition (0.5) a été introduite pour assurer l'existence d'un minimum pendant la sélection. Toutefois (0.5) est une condition suffisante mais pas nécessaire à l'existence d'un tel minimum. Elle peut donc être améliorée. C'est pourquoi en $\lambda\&^+$ nous remplaçons (0.5) par :

$$\forall U_i, U_j \in \{U_i\}_{i\in I}\,.\ \forall U \text{ élément maximal de } LB(U_i, U_j)\ \ \exists! h \in I.\, U_h = U \tag{0.6}$$

où $LB(S,T)$ est l'ensemble des bornes inférieures de S et T. Cette condition correspond à la gestion de l'héritage multiple, par exemple d'Eiffel. Le théorème suivant démontre qu'il s'agit d'une condition nécessaire et suffisante à l'existence du minimum :

Théorème 2 *Soit* $(Y, \leqslant)$ *un ensemble partiellement ordonné et,* $X \subseteq Y$. *Définissons :*
(1) $\forall a, b \in X. \forall c \in Y.$ (c *élément maximal de* $LB(a, b) \Rightarrow c \in X$)
(2) $\forall c \in Y.\ \{x \in X | c \leqslant x\}$ *est soit vide soit possède un plus petit élément*
alors (1) $\iff$ (2)

En outre, dans $\lambda\&^+$ on veut avoir la possibilité de remplacer une branche d'une fonction surchargée. Plus exactement si N possède le type $U \to T$ et M est une fonction surchargée possédant une branche pour le type U nous souhaitons qu'en écrivant $(M\&N)$ la branche N remplace l'ancienne branche pour U (pourvu que le type obtenu soit bien formé). Il y a plusieurs manières d'obtenir cela. La plus simple est de ne plus imposer dans

$$[\{\}\text{INTRO}] \qquad \frac{M\colon W_1 \leq \{U_i \to T_i\}_{i\leq(n-1)} \quad N\colon W_2 \leq U_n \to T_n}{(M\&^{\{U_i\to T_i\}_{i\leq n}}N)\colon \{U_i \to T_i\}_{i\leq n}}$$

que $\{U_i \to T_i\}_{i\leq(n-1)}$ soit un type bien formé. De cette façon il devient possible de typer la fonction suivante

$$(M^{\{U_1\to T_1, U_2\to T_2, U_3\to T_3\}}\ \ \&^{\{U_2\to T_2, U_3\to T_3, U_1\to T_4\}}\ \ N^{U_1\to T_4})$$

même si $U_1 \leq U_2, U_3$ (et donc $\{U_2 \to T_2, U_3 \to T_3\}$ ne peut pas être bien formé). Nous avons déjà vu qu'en $\lambda\&$ il est possible d'ajouter une nouvelle méthode à une classe donnée. Grâce à cette petite modification il devient aussi possible de *redéfinir* la méthode d'une classe.

Enfin, en $\lambda\&$ on effectue la sélection de la branche seulement après que l'argument ait atteint une forme normale fermée. En $\lambda\&^+$ nous voulons pouvoir effectuer la réduction plus tôt, dès que l'on est sûr que toute évaluation ultérieure de l'argument ne changera pas le résultat de la sélection. Dans ce but nous remplaçons dans $\lambda\&^+$ la réduction $(\beta_\&)$ par la réduction suivante

$\beta^+_\&$) Soit $U_j = min\{U_i | U \leq U_i\}$; si $N{:}U$ est fermé et en forme normale ou $\{U_i | U_i \leq U_j\} = \{U_j\}$ alors

$$((M_1 \&^{\{U_i \to V_i\}_{i=1..n}} M_2) \bullet N) \rhd \begin{cases} M_1 \bullet N & \text{for } j < n \\ M_2 \cdot N & \text{for } j = n \end{cases}$$

Autrement dit, dès que l'on a un terme de la forme $(M_1 \& M_2) \bullet N$ on contrôle s'il y a des branches avec un domaine plus petit que celui de la branche sélectionnée. Si ce n'est pas le cas alors la sélection ne peut plus changer et donc on peut effectuer la sélection, même si N n'est pas clos ou en forme normale. Cette modification est intéressante surtout pour l'optimisation des langages objets à la compilation. Une règle qui permet de résoudre la sélection d'une méthode à la compilation est indispensable pour produire du code efficace. En fait des études préliminaires sur des maquettes du compilateur pour Dylan ont montré qu'en moyenne environ 30% du temps d'exécution d'un code non optimisé est utilisé pour la sélection des méthodes. Résoudre la sélection à la compilation est donc une des nécessités les plus pressantes pour réaliser un langage utilisant des fonctions génériques. Dans $\lambda\&^+$ on peut donc utiliser la règle ($\beta^+_\&$) pour détecter pendant la compilation les envois de message qui peuvent déjà être liés à leur méthode.

Pour $\lambda\&^+$ nous avons démontré les théorèmes suivants :

- *Conservativité de la théorie de sous-typage* : Si S et T sont deux types bien formés dans $\lambda\&$ alors $\lambda\& \vdash S \leq T$ si et seulement si $\lambda\&^+ \vdash S \leq T$
- *Subject Reduction Généralisée* : Soit $M{:}U$. Si $M \rhd^* N$ alors $N{:}U'$, et $U' \leq U$.
- *Confluence* : Des termes égaux possèdent un réductum commun.
- *Normalisation forte* : Les sous-systèmes stratifiés sont fortement normalisants.

Deuxième variante : $\lambda\&$+*coerce*

La deuxième variante que nous étudions est l'extension de $\lambda\&$ par des coercitions explicites. De manière informelle on peut définir une coercition explicite comme un terme qui change le type de son argument. Par exemple le terme $\mathbf{coerce}^T(M)$ change le type de M en T, tout en en conservant les fonctionnalités. Cette construction est une extension cruciale dans $\lambda\&$ où les types déterminent l'exécution : le fait de pouvoir changer les types implique un plus grand contrôle sur l'exécution. En particulier il est possible de diriger la sélection sur une branche donnée en appliquant une coercition explicite à l'argument d'une fonction surchargée. De cette façon il devient donc possible de modéliser des commandes telles que `as` dans Dylan et `change-class` dans CLOS, dont la fonction est de changer la classe d'un objet.

Plus formellement, l'extension par coercitions de $\lambda\&$ est obtenue en ajoutant parmi les termes de $\lambda\&$ le terme suivant

$$\mathbf{coerce}^T(M)$$

parmi les règles de typage la règle suivante

$$[\textsc{Coerce}] \qquad \frac{\vdash M{:}S \leq T}{\vdash \mathbf{coerce}^T(M){:}T}$$

et parmi les notions de réduction

(coerce) $\quad \mathbf{coerce}^T(M) \circ N \ \triangleright M \circ N$

Il est possible de démontrer que cette extension possède les propriétés suivantes : la conservativité par rapport à la théorie équationelle (ou de réduction) des termes ; la subject-reduction généralisée ; la confluence ; la normalisation des sous-systèmes stratifiés.

Troisième variante :$\lambda^{\{\}}$

Nous voulons définir un système pur de fonctions surchargées, où la lambda abstraction soit obtenue comme cas particulier de fonction surchargée avec une seule branche. Il y a donc un seul opérateur d'abstraction et un seul opérateur d'application. La réduction effectuera une sélection et une substitution à la fois. Nous utilisons une discipline de sélection semblable à $(\beta^+_\&)$ de façon à ne forcer l'appel par valeur que si nécessaire (ainsi le λ-calcul simplement typé est un sous-calcul de $\lambda^{\{\}}$). Tout ceci peut être obtenu de la manière suivante :

$$\begin{array}{lcll} T & ::= & A \mid \{S_1 \to T_1, \cdots, S_n \to T_n\} & n \geq 1 \\ M & ::= & x \mid \lambda x(M_1{:}\, S_1 \Rightarrow T_1, \cdots, M_n{:}\, S_n \Rightarrow T_n) \mid MM & n \geq 1 \end{array}$$

Sous-typage

(subtype) $$\frac{\forall i \in I, \exists j \in J \ \ U''_i \leq U'_j \ \text{ et } \ V'_j \leq V''_i}{\{U'_j \to V'_j\}_{j \in J} \leq \{U''_i \to V''_i\}_{i \in I}}$$

Types

Tout type atomique appartient à **Types**. Si pour tout $i, j \in I$

a. $(U_i, V_i \in \mathbf{Types})$

b. $(U_i \leq U_j \Rightarrow V_i \leq V_j)$

c. $(U_i \Downarrow U_j \Rightarrow$ il existe un seul $h \in I$ tel que $U_h = \inf\{U_i, U_j\})$

alors $\{U_i \to V_i\}_{i \in I} \in \mathbf{Types}$

Typage

[Taut] $$\Gamma \vdash x{:}\, \Gamma(x)$$

[Intro] $$\frac{\forall i \in I \qquad \Gamma, (x{:}\, S_i) \vdash M_i{:}\, U_i \leq T_i}{\Gamma \vdash \lambda x(M_i : S_i \Rightarrow T_i)_{i \in I}{:}\, \{S_i \to T_i\}_{i \in I}}$$

[Elim] $$\frac{\Gamma \vdash M{:}\, \{S_i \to T_i\}_{i \in I} \ \ \Gamma \vdash N{:}\, S}{\Gamma \vdash MN{:}\, T_j} \qquad S_j = \min_{i \in I}\{S_i | S \leq S_i\}$$

Réduction

ζ) Soit $S_j = \min_{i \in I}\{S_i | U \leq S_i\}$ et $\Gamma \vdash N{:}\, U$; si N est fermé et en forme normale ou $\{S_i | S_i \leq S_j\} = \{S_j\}$ alors

$$\lambda x(M_i : S_i \Rightarrow U_i)_{i \in I} N \ \ \triangleright_\Gamma \ M_j[x{:=}\, N]$$

Pour cette variante nous avons démontré la propriété de subject réduction généralisée et la confluence.

L'intérêt de cette variante est principalement théorique. Toutefois elle n'est pas dépourvue d'intérêt pratique, car elle constitue un premier noyau de calcul basé uniquement sur des fonctions génériques (cf. Dylan).

Un méta-langage de $\lambda\&$

Dans la section consacrée à $\lambda\&$ nous avons montré la façon intuitive d'utiliser ce calcul pour modéliser la programmation orientée objets. Toutefois $\lambda\&$ n'est pas adéquat pour une étude des propriétés des langages objets ; il ne possède pas assez de structure pour pouvoir y traduire des "vrais" langages : il manque les commandes pour définir de nouveaux types atomiques, pour travailler sur leur représentation et pour définir la relation de sous-typage ainsi qu'un vrai concept d'objet. C'est pourquoi, nous définissons λ_object, un langage minimal directement dérivé de $\lambda\&$, qui possède ces propriétés, et davantage. En tant que langage, λ_object est caractérisé par une sémantique opérationnelle à reduction : cette sémantique est définie pour les termes non typés (dans le sens où nous définissons également la sémantique pour les termes qui ne seront pas bien typés) ; ceci nous permet de définir formellement les erreurs de type (en particulier nous distinguons parmi eux l'erreur "message not understood"). Nous définissons ensuite un algorithme de typage et nous en démontrons la correction par rapport à la sémantique opérationelle et à la définition d'erreur de type (dans le sens que l'exécution de tout programme bien typé ne cause pas d'erreur de type).

L'un des choix critiques dans la définition de λ_object est celui de la modélisation des objets. Un objet dans λ_object est un terme "taggué" (tagged term). On affecte le "tag" A au terme M par la construction $in^A(M)$; l'intuition est que A est la classe de l'objet $in^A(M)$ et M son état interne. Ainsi par exemple $in^{2DPoint}(\langle x = 0; y = 0\rangle)$ est un objet de la classe *2DPoint* dont les variables d'instance x et y ont la valeur 0. Les tags (i.e les classes) doivent être déclarés en les associant au type de leur état interne (ceci sert pour le typage). Par exemple, pour la classe *2DPoint* ceci est obtenu par :

$$\textbf{let}\ \mathit{2DPoint}\ \textbf{hide}\ \langle\!\langle x : int, y : int\rangle\!\rangle\ \textbf{in}\ \ldots$$

Dans un certain sens une telle déclaration correspond à la définition dans ML d'un "datatype" de la forme : $\textbf{datatype}\ \mathit{2DPoint} = in^{2DPoint}\ \textbf{of}\ \{x : int, y : int\}$

Plutôt que d'utiliser le filtrage comme dans ML nous utilisons, pour accéder à l'état interne d'un objet, une fonction *out* qui composée avec *in* donne l'identité. Il faut aussi déclarer un ordre sur les tags. Cet ordre, qui correspond à l'ordre de "sous-classe", est à la fois utilisé pour la résolution de la surcharge et pour le typage :

$$\textbf{let}\ \mathit{ColorPoint} \leq\ \mathit{2DPoint},\ \mathit{Color}\ \ \textbf{in}\ \ldots$$

Les fonctions surchargées ne peuvent avoir comme argument que des objets ou des produits d'objets (dispatch multiple). La description des types, des expressions et des programmes du langage est donnée par les productions suivantes, où A dénote un tag :

$$
\begin{array}{rcl}
T & ::= & A \;\mid\; T\times T \;\mid\; T\to T \\
 & \mid & \{(A_1\times\ldots\times A_{m_1})\to T_1,\ldots,(A_1\times\ldots\times A_{m_n})\to T_n\} \qquad {\scriptstyle n,m_i\geq 1} \\
M & ::= & x^T \;\mid\; \lambda x^T.M \;\mid\; M\cdot M \;\mid\; \varepsilon \;\mid\; M\&^T M \;\mid\; M\bullet M \\
 & \mid & <M\,,M> \;\mid\; \pi_1(M) \;\mid\; \pi_2(M) \;\mid\; \mu x^T.M \\
 & \mid & \mathbf{coerce}^A(M) \;\mid\; \mathbf{super}^A(M) \;\mid\; in^A(M) \;\mid\; out^A(M) \\
P & ::= & M \;\mid\; \mathbf{let}\ A\leq A_1,\ldots,A_n\ \mathbf{in}\ P \;\mid\; \mathbf{let}\ A\ \mathbf{hide}\ T\ \mathbf{in}\ P
\end{array}
$$

Sémantique opérationnelle

Un autre aspect important de λ_object est la définition des *valeurs tagguées* : une valeur tagguée est tout terme qui, passé à une fonction surchargée, permet immédiatement la sélection de la branche. Une bonne définition des valeurs tagguées est très importante parce que, couplée avec la règle $(\beta^+_\&)$ (cf. page 28), elle est le meilleur moyen pour effectuer des optimisations à la compilation. Nous utilisons G^D pour dénoter une valeur tagguée par le tag D.[9]

$$G^D ::= \; in^D(M) \mid \mathbf{coerce}^D(M) \mid \mathbf{super}^D(M) \mid <G_1^{A_1}, G_2^{A_2},\ldots,G_n^{A_n}>$$

Avant de donner la sémantique opérationelle nous devons encore définir la notion de valeur dans λ_object, c'est-à-dire distinguer les termes qui peuvent être considerés comme des résultats. Nous utilisons la méta-variable G pour dénoter les valeurs

$$G \;::=\; x \mid (\lambda x^T.M) \mid \varepsilon \mid (M_1\&^T M_2) \mid <G_1\,,G_2> \mid \mathbf{coerce}^A(M) \mid \mathbf{super}^A(M) \mid in^A(M)$$

La sémantique opérationnelle est donc définie par les règles suivantes (nous utilisons $\circ$ pour dénoter soit $\cdot$ soit $\bullet$ et $\overline{D}$ pour le $\min_{i=1..n}\{D_i | C\vdash D\leq D_i\}$) :

Axiomes

$(C\,,\ \pi_i(<G_1\,,G_2>)) \Rightarrow (C\,,\ G_i)$ $\quad$ i=1,2
$(C\,,\ out^{A_1}(in^{A_2}(M))) \Rightarrow (C\,,\ M)$
$(C\,,\ out^{A_1}(\mathbf{coerce}^{A_2}(M))) \Rightarrow (C\,,\ out^{A_1}(M))$
$(C\,,\ out^{A_1}(\mathbf{super}^{A_2}(M))) \Rightarrow (C\,,\ out^{A_1}(M))$
$(C\,,\ \mu x.M) \Rightarrow (C\,,\ M[x:=\mu x.M])$
$(C\,,\ (\lambda x.M)\cdot N) \Rightarrow (C\,,\ M[x:=N])$
$(C\,,\ (M_1\&^{\{D_1\to T_1,\ldots,D_n\to T_n\}}M_2)\bullet G^D) \Rightarrow (C\,,\ M_1\bullet G^D)$ $\quad$ if $D_n\neq\overline{D}$
$(C\,,\ (M_1\&^{\{D_1\to T_1,\ldots,D_n\to T_n\}}M_2)\bullet G^D) \Rightarrow (C\,,\ M_2\cdot G^D)$ $\quad$ if $D_n=\overline{D}$ and $G^D\not\equiv\mathbf{super}^D(M)$
$(C\,,\ (M_1\&^{\{D_1\to T_1,\ldots,D_n\to T_n\}}M_2)\bullet G^D) \Rightarrow (C\,,\ M_2\cdot M)$ $\quad$ if $D_n=\overline{D}$ and $G^D\equiv\mathbf{super}^D(M)$
$(C\,,\ \mathbf{let}\ A\leq A_1\ldots A_n\ \mathbf{in}\ P) \Rightarrow (C\cup(A\leq A_1)\cup\ldots\cup(A\leq A_n)\,,\ P)$
$(C\,,\ \mathbf{let}\ A\ \mathbf{hide}\ T\ \mathbf{in}\ P) \Rightarrow (C\,,\ P)$

[9] Dans la dernière production $D\equiv(A_1\times\ldots\times A_n)$; donc un tag peut être aussi le produit de plusieurs types atomiques. Nous utilisons le non terminal D pour les produits de types atomiques.

Règles de contexte

$$\frac{(\,C\ ,\ M\,) \Rightarrow (\,C\ ,\ M'\,)}{(\,C\ ,\ <M\,,N>\,) \Rightarrow (\,C\ ,\ <M'\,,N>\,)} \qquad \frac{(\,C\ ,\ M\,) \Rightarrow (\,C\ ,\ M'\,)}{(\,C\ ,\ <G\,,M>\,) \Rightarrow (\,C\ ,\ <G\,,M'>\,)}$$

$$\frac{(\,C\ ,\ M\,) \Rightarrow (\,C\ ,\ M'\,)}{(\,C\ ,\ \pi_i(M)\,) \Rightarrow (\,C\ ,\ \pi_i(M')\,)} \qquad \frac{(\,C\ ,\ M\,) \Rightarrow (\,C\ ,\ M'\,)}{(\,C\ ,\ out^A(M)\,) \Rightarrow (\,C\ ,\ out^A(M')\,)}$$

$$\frac{(\,C\ ,\ M\,) \Rightarrow (\,C\ ,\ M'\,)}{(\,C\ ,\ M \circ N\,) \Rightarrow (\,C\ ,\ M' \circ N\,)} \qquad \frac{(\,C\ ,\ M\,) \Rightarrow (\,C\ ,\ M'\,)}{(\,C\ ,\ (N_1\&N_2)\bullet M\,) \Rightarrow (\,C\ ,\ (N_1\&N_2)\bullet M'\,)}$$

L'application ordinaire est implémentée par un appel-par-nom, tandis que l'application d'une fonction surchargée est réalisée par un "appel-par-valeur-tagguée".

Système de types

Nous sommes à present capables de donner la définition d'erreur de type. Un programme produit une erreur de type s'il se réduit à une forme normale qui n'est pas une valeur. En particulier si l'exécution est bloquée sur un terme de la forme $((M_1\&^T M_2)\bullet G^D)$ nous disons que l'erreur produite est "message not understood". Il faut noter que dans ce dernier cas la réécriture a échoué parce que $\overline{D}$ (i.e. $\min_{i=1..n}\{D_i|C \vdash D \leq D_i\}$) n'est pas défini : ceci peut dériver soit du fait que l'ensemble $\{D_i|D \leq D_i\ ,\ i = 1..n\}$ est vide (ceci signifie que nous avons envoyé un message au mauvais destinataire) soit du fait qu'il n'y a pas de plus petit élément (ceci signifie que la condition sur l'héritage multiple n'a pas été respectée).

Il nous reste à définir un système de types qui assure qu'un programme bien typé ne produit pas d'erreurs de type. Le système de type en question est très semblable à celui de $\lambda\&^+$ (parce que nous voulons pouvoir modéliser l'héritage dans sa forme la plus générale et l'extension/redéfinition de méthode). La différence la plus importante est qu'il doit prendre en compte les déclarations pour les tags introduites par (**let** $A \leq \ldots$ **in**) et (**let** A **hide** $\ldots$ **in**). Ceci est fait par les règles suivantes :

[NewType] $$\frac{C\ ,\ S[A \leftarrow T] \vdash P\colon U}{C, S \vdash \textbf{let}\ A\ \textbf{hide}\ T\ \textbf{in}\ P\colon U}$$ $A \notin dom(S), T \in_{C,S}$ **Types** and T not atomic

[Constraint] $$\frac{C \cup (A \leq A_i)_{1=1..n}, S \vdash P\colon T}{C, S \vdash \textbf{let}\ A \leq A_1, \ldots, A_n\ \textbf{in}\ P\colon T}$$ if $C \vdash S(A) \leq S(A_i)$ and A do not appear in C

[Coerce] $$\frac{C, S \vdash M\colon B}{C, S \vdash \textbf{coerce}^A(M)\colon A}$$ $C \vdash B \leq A$ and $A \in_{C,S}$ **Types**

[Super] $$\frac{C, S \vdash M\colon B}{C, S \vdash \textbf{super}^A(M)\colon A}$$ $C \vdash B \leq A$ and $A \in_{C,S}$ **Types**

[In] $$\frac{C, S \vdash M\colon T}{C, S \vdash in^A(M)\colon A}$$ $C \vdash T \leq S(A)$ and $A \in_{C,S}$ **Types**

[Out] $$\frac{C, S \vdash M\colon B}{C, S \vdash out^A(M)\colon S(A)}$$ $C \vdash B \leq A$ and $A \in_{C,S}$ **Types**

On peut donc prouver que si un programme est bien typé alors il ne produit pas d'erreurs de type. Le langage λ_object ainsi défini peut alors être utilisé pour prouver des propriétés des langages objets. Dans le chapitre de la thèse dédié à λ_object nous définissons un langage fonctionnel objets simple où l'on retrouve les constructions usuelles (`class`, `super`, `self`, `new`, héritage multiple etc.) et d'autres moins usuelles (messages de première classe, extension/redéfinition de méthodes, multi-méthodes mutuellement récursives)[10] ; nous définissons aussi une discipline de types pour ce langage. Puis nous donnons une traduction de ce langage dans λ_object et en définissons ainsi la sémantique. Enfin nous démontrons que la traduction d'un programme bien typé est un terme bien typé dans λ_object et obtenons, de cette façon, une preuve de correction de la discipline de types définie pour ce langage objets simple .

Pour terminer nous montrons comment il est possible, sous certaines hypothèses, de coder λ_object dans $\lambda\&^+$.

L'implémentation en CAML LIGHT d'un interpreteur pour λ_object est décrite en Appendix A.

Sémantique

La définition d'une sémantique pour λ& présente quatre problèmes principaux: pré-ordre, exécution dépendant des types, liaison tardive, imprédicativité. Plus précisément :

- *Pré-ordre* : La relation de sous-typage dans λ& est un pré-ordre. En fait soit $U \leq V$, alors par les règles de sous-typage de λ& on obtient que $\{U \to T, V \to T\} \leq \{V \to T\}$ et $\{V \to T\} \leq \{U \to T, V \to T\}$. Toutefois ces deux types sont complètement interchangeables : on peut utiliser l'un ou l'autre dans un programme sans que rien ne change. Nous voulons par conséquent qu'ils aient la même interprétation, et donc que "$\leq$" soit interprété par une relation d'ordre entre les types sémantiques.

- *Exécution dépendant des types* : les types des termes déterminent le résultat de l'exécution. Par conséquent l'interprétation d'une fonction surchargée doit prendre en compte l'interprétation du type de son argument.

- *Liaison tardive* : L'interprétation d'une application surchargée doit être déterminée par le type de l'argument à l'exécution, lorsque cet argument aura atteint une forme normale fermée.

- *Imprédicativité* : nous avons déjà montré dans la section sur la normalisation forte que la relation de sous-typage ne respecte pas la taille des types ; c'est pourquoi il est possible d'appliquer une fonction de type $\{\{\} \to T\}$ à elle même. Nous venons de dire que l'interprétation d'une fonction doit prendre en compte l'interprétation du type de son argument. Par conséquent l'interprétation du type $\{\{\} \to T\}$ fait appel à l'interprétation du type lui même, d'où l'imprédicativité!

Pour traiter le problème du pré-ordre nous utilisons la construction syntaxique dite de "complétion". Un type surchargé $\{U \to T\}$ possède en effet un nombre (potentiellement infini)

[10] voir Appendix C.1

de branches virtuelles, une pour chaque sous-type de U. La complétion d'un type surchargé est son expansion par ces branches virtuelles. Donc par exemple la complétion de $\{U \to T\}$ sera $\{U \to T, U_1 \to T, U_2 \to T, \dots\}$ où $U_1, U_2, \dots$ forment l'ensemble des sous-types de U. Plus formellement, on change la notation des types surchargés de manière à pouvoir écrire des types infinis. Nous écrivons $\Downarrow H$, si l'ensemble de types H possède une borne inférieure.

Définition 1 *Un* type surchargé généralisé *(t.s.g.) est un couple* (K, out) *où* K *est un ensemble de types et out est une fonction de* K *vers* **Types** *telle que*

1. *si* $H \subseteq K$ *et* $\Downarrow H$ *alors il existe* $V \in K$ *tel que* $V \in \inf H$.[11]
2. *out est monotone par rapport au pré-ordre de sous-typage.*

Parfois on utilisera $\{U \to out(U)\}_{U \in K}$ *pour dénoter le t.s.g.* (K, out)

Il faut noter que tout type surchargé est un t.s.g. mais pas nécessairement le contraire.

Nous pouvons maintenant définir la complétion. Nous complétons un t.s.g. en élargissant son domaine par les sous-types des types dans le domaine et en étendant la fonction *out* au domaine ainsi élargi. En d'autres termes, la complétion de (K, out) est le t.s.g. $(\widehat{K}, \widehat{out})$ où $\widehat{K} = \{U' | \exists U \in K \;\; U' \leq U\}$ et $\widehat{out}(U') = out(\min\{U \in K | U' \leq U\})$.

L'interprétation d'un type surchargé sera l'interprétation de sa complétion, et ainsi nous obtiendrons que deux types équivalents (c'est-à-dire tels que l'un soit plus petit que l'autre et vice-versa) aient la même interprétation, puisqu'il est possible de prouver qu'ils ont la même complétions.

La sémantique est donnée dans un modèle $(\mathcal{D}, \cdot)$ du λ-calcul sans type : les termes sont interprétés par des (classes d'équivalence d') éléments du modèle et les types par des relations partielles d'équivalence sur le modèle même. En outre, pour tenir compte du fait que l'exécution dépend des types, nous codons l'ensemble des types syntaxiques, **Types** par un sous-ensemble [**Types**]$\subseteq \mathcal{D}$ du modèle[12]. Il y a donc une dualité dans l'interprétation syntaxique d'un type : type comme ensemble de valeurs qui est interprété par une relation partielle d'équivalence, et type comme valeur déterminant l'exécution qui est interprété comme élément du modèle dans [**Types**]. La dépendance est alors obtenue en interprétant un type surchargé par un produit indexé sur les codes des types syntaxiques. Plus précisément, si l'on écrit $\mathcal{T}_d$ pour le type syntaxique associé à l'élément d dans [**Type**], la sémantique d'un type surchargé est donnée par

$$[\![\{U \to out(U)\}_{U \in K}]\!] = \prod_{n \in [\widehat{K}]} [\![\mathcal{T}_n \to \widehat{out}(\mathcal{T}_n)]\!]$$

où $[\widehat{K}] = \{d \,|\, \mathcal{T}_d \in \widehat{K}\}$. À ce point on cerne mieux d'où surgit le problème de l'imprédicativité : considérons une fois de plus le type $\{\,\{\} \to T\,\}$ et soit $\overline{d}$ son codage dans [**Types**] ; son interprétation sera un produit indexé sur les codes des sous-types de $\{\}$, i.e. sur les codes de tous les types surchargés. Par conséquent, parmi les index du produit il y aura $\overline{d}$ lui même. Donc l'interprétation du type ci-dessus, c'est-à-dire $[\![\mathcal{T}_{\overline{d}}]\!]$ est donné en termes de $[\![\mathcal{T}_{\overline{d}}]\!] \to [\![T]\!]$. Cette

[11] La plus grande borne inférieure dans le pré-ordre est une classe d'équivalence

[12] La seule restriction que l'on impose est que la topologie induite sur [**Type**] soit la topologie discrète, afin que toute fonction dans [**Types**] $\to \mathcal{D}$ puisse être étendue à une fonction continue dans $\mathcal{D} \to \mathcal{D}$

forme d'impédicativité est similaire à celle que l'on trouve dans le Système F : la sémantique de $\forall X.T$ est le produit indexé sur la sémantique de tous les types et, donc, sur $\forall X.T$ lui même. Dans les relations partielles d'équivalence, par exemple, le problème est résolu en indexant le produit sur l'ensemble PER de toutes les relations partielles d'équivalence :

$$[\![\forall X.T]\!]_E = \prod_{C \in \mathrm{PER}} [\![T]\!]_{E[X:=C]}$$

Cette définition est bien fondée car PER existe indépendamment des types qui y sont interprétés. De façon semblable dans $F_{\leq}$ on interprète $\forall X \leq S.T$ par le produit indexé sur les *sous-relations* de l'interprétation de S :

$$[\![\forall X{\leq}S.T]\!]_E = \prod_{C \subseteq [\![S]\!]_E} [\![T]\!]_{E[X:=C]}$$

Dans notre système nous ne sommes pas capables d'imiter ces constructions parce que nous sommes forcés d'indexer les produits sur un "codage" des types, qui donc n'existe pas de manière indépendante des types mêmes. Il nous est ainsi impossible de définir dans le modèle un ordre des codes qui respecte l'ordre de sous-typage. Autrement dit, dans les modèles de $F_{\leq}$ il est possible de définir un ordre $\subseteq$ tel que si $S \leq T$ alors $[\![S]\!] \subseteq [\![T]\!]$. Ici nous ne sommes pas capables de trouver *dans le modèle* un ordre $\preccurlyeq$ sur les éléments de $\mathcal{D}$ tel que si $\mathcal{T}_n \leq \mathcal{T}_m$ alors $n \preccurlyeq m$. Donc nous ne sommes pas capable de donner la sémantique du système complet. C'est pourquoi, les définitions de cette section s'appliquent uniquement aux sous-systèmes stratifiés, pour lesquels on ne rencontre pas ce problème d'imprédicativité.

Le fait que le calcul soit dépendant des types se manifeste dans la définition de la sémantique d'une fonction surchargée. Soit $(M_1\&M_2)$ de type $\{U \to out(U)\}_{U\in K}$; son interprétation doit être un élément d'un produit indexé sur le codage des types. Plus précisément $[\![(M_1\&M_2)]\!]$ est la fonction f qui pour tout code d d'un type dans $\widehat{K}$ et $U_j = \min\{U \in K | \mathcal{T}_d \leq U\}$ est définie de la façon suivante :

$$f(d) = \begin{cases} [\![M_2]\!] & \text{si } U_j \text{ sélectionne la deuxième branche} \\ [\![M_1]\!](d) & \text{sinon} \end{cases}$$

L'interprétation d'une fonction surchargée est un élément d'un produit indexé ; elle est donc une fonction dans le modèle qui prend comme argument le code d'un type et restitue un élément (une classe d'équivalence) dans le type sémantique approprié. Ainsi l'interprétation de l'application d'une fonction surchargée est donnée par

$$[\![M \bullet N]\!] = ([\![M]\!](d))[\![N]\!] \tag{0.7}$$

Avec la liaison tardive d doit être le codage du type de la forme normale fermée de N. Or, il est impossible connaître ce type sans exécuter N et le contexte dans lequel cette expression se trouve. D'autre part, il n'est pas possible d'effectuer cette exécution pour donner l'interprétation car on perdrait la compositionalité de la définition. On pourrait envisager d'utiliser à la place de d (le codage d') une variable de type, mais ainsi on entrerait dans le domaine d'un calcul du second ordre que nous ne maîtrisons pas encore. Donc nous

nous limitons au cas de liaison précoce. Dans ce but nous modifions la définition des termes de $\lambda\&$ de la manière suivante :

$$M ::= x^V \mid \lambda x^V.M \mid M \cdot M \mid \mathbf{coerce}^V(M) \mid \varepsilon \mid M\&M \mid M\bullet\mathbf{coerce}^V(M)$$

ainsi nous sommes sûrs que le type de l'argument d'une fonction surchargée ne pourra plus changer pendant l'exécution, car il est *contraint* (coerced) à un type donné. Par conséquent "*d*" dans (0.7) est le code associé au type de N.

Pour cette sémantique on peut démontrer les théorèmes suivants :

- *Correction par rapport au sous-typage* : Si $U \leq V$ est dérivable alors $[\![U]\!] \sqsubseteq [\![V]\!]$ ($\sqsubseteq$ inclusion entre relations partielles d'équivalence comme ensembles de couples)
- *Correction par rapport au typage* : Si $N\!:\!U$ est dérivable alors $[\![N]\!] \in [\![U]\!]$. (relation comme ensemble de classes d'équivalence)
- *Correction par rapport à la réduction* : Si $M \rhd N$ alors $[\![M]\!] \supseteq [\![N]\!]$. (classe d'équivalence comme ensemble d'éléments de $\mathcal{D}$)

Second ordre

Dans la section précédente nous évoquions la possibilité d'étudier un formalisme du second ordre afin de mieux cerner la signification mathématique de la liaison tardive. Une telle étude couvrirait un thème qui, jusqu'à présent, a été négligé par la recherche, soit, l'étude du polymorphisme dans un calcul où l'exécution dépendrait directement des types. Le Système F de Girard [Gir72] est un calcul où l'on peut définir des fonctions qui prennent des types comme argument ; néanmoins ces fonctions dépendent de leurs arguments d'une façon très limitée : des types différents comme argument n'affectent que le type du résultat, pas sa valeur. La contrepartie pratique de cette propriété est que les types peuvent être effacés pour l'exécution, qui est ainsi effectuée sur les *effacements* (erasures) des termes. $F_{\leq}$ est une extension conservative de F qui permet de plus de spécifier des bornes pour les paramètres de type. Cependant il possède encore la même forme de dépendance des types que F, car les types peuvent être effacés pour l'exécution. En revanche, nous somme intéressés en une dépendance de types qui affecte l'exécution. Nous voulons pouvoir décrire des fonctions qui exécutent (dispatch) des codes différents en fonction du type passé comme argument. Cette recherche se situe donc dans un cadre plus général : elle vise la formalisation du polymorphisme explicite dit "ad hoc" [Str67] ainsi que son intégration avec sa contrepartie : le polymorphisme paramétrique.

Toutefois il y a d'autres motivations qui amènent à s'intéresser au second ordre, et dont la retombée pratique immédiate est bien plus importante. Il s'agit de résoudre le problème qui dans la recherche théorique sur les systèmes de types pour les langages à objets est dénommé "le problème de perte d'information". Ce problème a été présenté dans [Car88] sur une observation de Antonio Albano et peut être illustré de la manière suivante. Considérons la règle d'application du λ-calcul simplement typé avec sous-typage:

$$[\to\textsc{Elim}_{\leq}] \qquad \frac{M\!:\!T \to U \qquad N\!:\!S \leq T}{MN\!:\!U}$$

et deux types S et T tels que $S \leq T$; par exemple soit S un type enregistrement comme T mais avec des champs supplémentaires. Prenons la fonction d'identité pour T, $\lambda x^T.x$ qui possède donc le type $T \to T$. Soit M un terme de type S. Par la règle ci-dessus $(\lambda x^T.x)M$ a le type T (plutôt que S). Nous avons ainsi perdu de l'information; dans l'exemple des enregistrements on a perdu l'information liée aux champs supplémentaires de S car, après avoir appliqué M à l'identité, il n'est plus possible d'avoir accès à ces champs.

La solution à ce problème consiste à passer à un formalisme du second ordre, et a été originairement décrite dans [CW85]. L'idée en elle même est très simple : l'identité ci-dessus n'est plus une fonction qui accepte tout argument de type plus petit ou égal à T et rend un résultat de type T, mais elle devient une fonction qui accepte tout argument de type plus petit ou égal à T et rend un résultat *du même type que* celui de l'argument. Autrement dit, la fonction identité possède le type $\forall X{\leq}T.X \to X$. [13] On rencontre un problème analogue dans $\lambda\&$. Soit m un message qui modifie l'état interne des objets de la classe C_1. Puisque nous sommes dans un cadre fonctionnel la méthode dans C_1 rend un *nouvel* objet de classe C_1. Par conséquent $m{:}\,\{..., C_1 \to C_1, ...\}$. Soit C_2 une sous-classe de C_1 qui hérite la méthode en question. Si l'on passe le message m à un objet de classe C_2 la branche définie dans C_1 va être sélectionnée. Mais cette branche possède le type $C_1 \to C_1$, par conséquent le résultat du passage du message aura le type C_1 au lieu de C_2.

Nous utilisons pour $\lambda\&$ la même solution que pour le problème de [Car88] et passons à l'étude de formalismes du second ordre avec sous-typage. Plus particulièrement, notre recherche se concentre sur $F_{\leq}$ qui est considéré comme la formalisation standard de la quantification bornée. Nous agissons tout d'abord sur deux plans : nous essayons d'améliorer $F_{\leq}$ et, parallèlement, de l'étendre par la surcharge et la liaison tardive. Après avoir réuni ces deux études, nous montrons comment le polymorphisme "ad hoc" explicite peut être utilisé pour modéliser la programmation objets.

Vers une quantification bornée décidable

Les systèmes avec quantification bornée sont obtenus en enrichissant le Système F par une relation de sous-typage et un type Top (le type plus grand que tous les types) et en spécifiant une borne pour toute variable quantifiée. Les règles d'introduction et d'élimination des types polymorphes sont étendues de manière à prendre en compte ces bornes.

La formulation standard de la quantification bornée, $F_{\leq}$, a été définie par Curien et Ghelli [CG92]. Parmi les règles de sous-typage celle qui permet de comparer les types polymorphes possède la forme suivante :

$$(\forall\text{-orig}) \qquad \frac{C \vdash T_1 \leq S_1 \qquad C \cup \{X \leq T_1\} \vdash S_2 \leq T_2}{C \vdash \forall(X{\leq}S_1)S_2 \leq \forall(X{\leq}T_1)T_2} \qquad X \notin dom(C)$$

Cette règle peut être lue de la façon suivante. Un type T de la forme $\forall(X{\leq}T_1)T_2$ décrit une collection de termes polymorphes (fonctions qui prennent un type et rendent un terme) qui

[13] Ce genre de polymorphisme peut être soit de forme explicite (la quantification est linguistique : X est substitué par le type explicitement passé à la fonction) soit de forme implicite (la quantification est méta-linguistique : X est substitué par le type trouvé par unification par le système de typage).

associent des sous-types de T_1 à des termes dans T_2. Si T_1 est un sous type de S_1, alors le domaine de T est plus petit que celui de $\forall(X{\leq}S_1)T_2$; ainsi ce dernier a une contrainte plus faible et décrit une plus grande collection de termes polymorphes. En outre, si l'on considère un type $S = \forall(X{\leq}S_1)S_2$ tel que pour tout U qui soit un argument recevable par les fonctions des deux collections (i.e. par celles avec la contrainte plus restrictive : $U \leq T_1$), la U-instance de S_2 est un sous-type de la U-instance de T_2 alors T est "par points" une contrainte plus faible que S et il décrit donc une plus grande collection de termes polymorphes.

Bien que sémantiquement très naturelle, cette règle est à l'origine de la perte de maintes propriétés syntaxiques désirables. Tout d'abord elle cause l'indécidabilité de la relation de sous-typage [Pie93, Ghe93a], ce qui entraîne l'indécidabilité du contrôle des types. Mais pire encore, $F_{\leq}$ ne possède pas de plus grande borne inférieure pour les ensembles bornés de types, ce qui empêche certains raisonnements par induction ou, plus simplement, le typage d'un "if_then_else" possédant deux branches dont les types soient incomparables (mais avec un minorant commun).

Le cœur du problème réside dans la borne pour la variable X de S_2 : cette borne est S_1 dans la conclusion de la règle, et devient T_1 dans la prémisse droite. Ce "re-liage" des variables, en lui même assez bizarre, invalide toute une classe de raisonnements par induction structurelle sur les types, où le cas pour une variable utilise normalement l'hypothèse d'induction sur la borne.

C'est pourquoi plusieurs variantes de cette règle ont été proposé dans la littérature. La règle originaire proposée par Cardelli et Wegner dans [CW85] pour leur calcul Fun (voir (∀-Fun) Figure 0.1) évite ce reliage en limitant la comparaison à des types ayant la même borne ;

$$(\forall\text{-Fun})\quad \frac{C \cup \{X{\leq}U\} \vdash S_2 \leq T_2}{C \vdash \forall(X{\leq}U)S_2 \leq \forall(X{\leq}U)T_2} \qquad (\forall\text{-local})\quad \frac{C \vdash T_1 \leq S_1 \qquad C \cup \{X{\leq}S_1\} \vdash S_2 \leq T_2}{C \vdash \forall(X{\leq}S_1)S_2 \leq \forall(X{\leq}T_1)T_2}$$

Figure 0.1: Règles alternatives

quoique décidable cette restriction ne possède aucune motivation sémantique apparente et elle empêche la modélisation des types partiellement abstraits (voir [CP94]).

Katiyar et Shankar [KS92] proposent une restriction dans laquelle les bornes ne peuvent par contenir Top. De cette façon on obtient la décidabilité mais aux frais de l'expressivité du calcul car les enregistrements (dont le codage dans $F_{\leq}$ utilise Top) ne peuvent plus être utilisés comme bornes. Une autre variation, (∀-local), utilise la borne la plus grande pour comparer les corps des quantifications (voir Figure 0.1). Mais Giorgio Ghelli a montré récemment qu'elle est impraticable d'un point de vu algorithmique, tout en laissant ouvert le problème de la décidabilité.

Nous proposons une alternative très simple : une règle de sous-typage pour la quantification universelle

$$(\forall\text{-new})\qquad \frac{C \vdash T_1 \leq S_1 \qquad C \cup \{X{\leq}\mathsf{Top}\} \vdash S_2 \leq T_2}{C \vdash \forall(X{\leq}S_1)S_2 \leq \forall(X{\leq}T_1)T_2} \qquad X \notin dom(C)$$

où les corps des quantifications doivent être reliés ($S_2 \leq T_2$), mais sous *aucune* hypothèse

(refl)	$C \vdash T \leq T$	(Top)	$C \vdash T \leq \mathsf{Top}$
(trans)	$\dfrac{C \vdash T_1 \leq T_2 \qquad C \vdash T_2 \leq T_3}{C \vdash T_1 \leq T_3}$	($\to$)	$\dfrac{C \vdash T_1 \leq S_1 \qquad C \vdash S_2 \leq T_2}{C \vdash S_1 \to S_2 \leq T_1 \to T_2}$
(taut)	$C \vdash X \leq C(X)$	($\forall$)	$\dfrac{C \vdash T_1 \leq S_1 \qquad C \cup \{X{\leq}\mathsf{Top}\} \vdash S_2 \leq T_2}{C \vdash \forall(X{\leq}S_1)S_2 \leq \forall(X{\leq}T_1)T_2}$

Figure 0.2: Relation de sous-typage pour $F_{\leq}^{\top}$

préalable ($X{\leq}\mathsf{Top}$). Cette règle est d'un point de vu sémantique plus naturelle que les autres variantes puisqu'elle concrétise une notion de sous-typage "par points" que l'on retrouve dans d'autres travaux [Mit90b, BM92, Bru93, PT93] : elle affirme, tout simplement, que $\forall F \leq \forall G$ (où F, G : **Type**$\to$**Type**) si et seulement si $dom(G) \subseteq dom(F)$ et F est "par points" plus petit que G.

Évidement, cette règle est plus faible que ($\forall$-orig). Le système obtenu de $F_{\leq}$ en remplaçant ($\forall$-orig) par ($\forall$-new), et que nous appelons $F_{\leq}^{\top}$(voir Figure 0.2), ne peut pas prouver une un jugement tel que $\vdash \forall(X \leq S_1)X \leq \forall(X \leq S_2)S_2$ qui est au contraire prouvable dans $F_{\leq}$. Toutefois cette différence d'expressivité ne joue aucun rôle dans les cas d'application pratique de la quantification bornée. En fait, nous avons testé cette règle avec le modèle de programmation orientée objets présenté dans [PT93], qui utilise l'extension de $F_{\leq}$ par le polymorphisme d'ordre supérieur du Système F^{ω} [Gir72] : tout terme de ce modèle possède dans $F_{\leq}^{\top}$ le même type que dans $F_{\leq}$. De même, Luca Cardelli nous a confirmé que tout programme écrit pour son implémentation de $F_{\leq}$ [Car93], est aussi typable dans $F_{\leq}^{\top}$. Une autre application de la quantification bornée est donnée par la traduction due à Cardelli d'un calcul d'enregistrements extensibles dans $F_{\leq}$ [Car92] : une fois de plus, $F_{\leq}^{\top}$ peut être utilisé à la place de $F_{\leq}$.

$F_{\leq}^{\top}$ possède plusieurs propriétés syntaxiques qui font défaut à $F_{\leq}$. En premier lieu le sous-typage est décidable. En fait il est possible de définir un algorithme de sous-typage qui est valide (sound) et complet (complete) par rapport à $F_{\leq}^{\top}$; ceci est obtenu en démontrant pour $F_{\leq}^{\top}$ les propriétés d'élimination de la réflexivité et de la transitivité, et la terminaison de l'algorithme qui en dérive[14].

La décidabilité du sous-typage est une propriété assez souhaitable. Cependant, d'un point de vu pratique, son absence cause moins de troubles que la non existence du plus grand minorant et du plus petit majorant. Giorgio Ghelli [Ghe90] a montré que $F_{\leq}$ présente une telle carence. Par contre dans $F_{\leq}^{\top}$ toute paire de types possède un plus petit majorant et tout paire de type avec un minorant commun possède un plus grand minorant.

La simplicité syntaxique accrue de la formalisation de $F_{\leq}^{\top}$ entraîne une remarquable simplification des preuves par rapport à $F_{\leq}$. Cette simplification, nous la rencontrons tant dans les preuves de validité et complétude de l'algorithme de sous-typage qu'en étudiant la sémantique de $F_{\leq}^{\top}$. Pour donner une sémantique à $F_{\leq}^{\top}$ il est possible d'utiliser la technique

[14]Un récent résultat non encore publié de Giorgio Ghelli prouverait que cet algorithme peut être implémenté de manière polynômiale.

de [BTCGS91], et ainsi traduire $F_{\leq}^{\top}$ dans un calcul avec des coercitions explicites pour lequel plusieurs modèles sont déjà définis. Grâce à l'existence des plus grandes bornes inférieures et des plus petites bornes supérieures, il est possible d'utiliser cette même technique pour donner une sémantique à l'extension de $F_{\leq}^{\top}$ qui comprend les types enregistrements, les types récursifs et les types énumérés (variant types), technique qui, dans le cas de $F_{\leq}$, échoue.

Si nous étendons le langage des types de $F_{\leq}$ par les types récursifs ($\mu X.T$) et ajoutons parmi les règles de sous-typage

$$\text{(Unfold-L)}\ \frac{C \vdash S[X := \mu X.S] \leq T}{C \vdash \mu X.S \leq T} \qquad \text{(Unfold-R)}\ \frac{C \vdash T \leq S[X := \mu X.S]}{C \vdash T \leq \mu X.S}$$

la théorie de sous-typage ainsi obtenue n'est pas une extension conservative de la théorie pour $F_{\leq}$. Autrement dit, il existe un jugement de sous-typage dans $F_{\leq}$ qui n'est pas prouvable avec le système originel mais qui est prouvable par la théorie étendue [Ghe93b]. Donc l'algorithme de contrôle des types obtenu en ajoutant à l'algorithme pour $F_{\leq}$ les deux règles ci-dessus, n'est pas correct. Par contre si nous étendons $F_{\leq}^{\top}$ de la même manière —appellons ce système $F_{\leq}^{\top\mu}$— nous obtenons une extension conservative de la théorie de sous-typage. Ceci découle très simplement de la démonstration d'élimination de la transitivité pour $F_{\leq}^{\top\mu}$ (propriété qui n'est pas satisfaite dans l'extension de $F_{\leq}$).[15] Par conséquent, nous pouvons étendre l'algorithme de sous-typage pour $F_{\leq}^{\top}$ par les deux règles ci-dessus ; nous obtenons ainsi un algorithme de sous-typage pour $F_{\leq}^{\top\mu}$ qui est valide, complet et qui termine.

Comme nous l'avons montré, le passage de la règle ($\forall$-orig) à la règle ($\forall$-new) apporte de nombreux bénéfices au système de sous-typage, qui dès lors satisfait plusieurs propriétés souhaitables. Malheureusement, ce même passage a un effet néfaste sur la relation de typage. En fait, comme Giorgio Ghelli l'a fait remarquer, $F_{\leq}^{\top}$ ne possède pas la propriété d'existence d'un type minimal, dans le sens où l'ensemble des types prouvés par un terme donné peut ne pas posséder de plus petit élément. Ceci est illustré par l'exemple suivant: considérons le terme $M \equiv \Lambda X{\leq}Y.\lambda x^X.x$. En utilisant la règle de subsumption, il est possible de prouver que ce terme est typé tant par le type $\forall(X{\leq}Y)X \to X$ que par type $\forall(X{\leq}Y)X \to Y$. Ces deux types sont tous les deux minimaux dans l'ensemble des types de M. En conséquence, si dans l'algorithme de typage habituel pour $F_{\leq}$ nous utilisons l'algorithme de sous-typage pour $F_{\leq}^{\top}$, nous obtenons un algorithme de typage qui est valide pour $F_{\leq}^{\top}$, mais qui n'est pas complet. Par exemple, l'algorithme de typage déduira pour M ci-dessus le premier type mais pas le second. Le problème de trouver un algorithme de typage complet dans ce cas demeure ouvert, comme demeure ouvert le problème de la décidabilité de la relation de typage.

C'est l'interaction entre la règle de subsumption, et la règle ($\forall$-new) qui est responsable de la perte de cette propriété. En fait, en utilisant la subsumption nous pouvons faire remonter dans la hiérarchie de sous-typage, le type du corps d'une Λ-abstraction, alors que la règle ($\forall$-new) empêche l'algorithme de sous-typage de faire la même chose. Toutefois, il faut noter que les tests que nous avons effectués sur les interpréteurs existants de $F_{\leq}$, ont consisté à

[15] En fait, si T_1 et T_2 sont deux types de $F_{\leq}$, et si $T_1 \leq T_2$ est faux dans $F_{\leq}$ mais est prouvable dans l'extension par les types récursifs, alors il existe un type récursif T tel que $T_1 \leq T \leq T_2$ soit prouvable dans l'extension. Ceci n'est pas possible dans $F_{\leq}^{\top\mu}$: si c'était le cas, par l'élimination de la transitivité on obtiendrait une preuve de $T_1 \leq T_2$ sans utiliser la transitivité. Mais la preuve ainsi obtenue serait aussi une preuve de $T_1 \leq T_2$ dans $F_{\leq}^{\top}$: ce qui est impossible.

remplacer dans l'algorithme de sous-typage la règle ($\forall$-orig) par la règle ($\forall$-new), mais tout en continuant à utiliser l'algorithme de typage traditionnel. Ainsi, aucune utilisation de la règle de subsumption n'a été faite. On pourrait être tenté de continuer à adopter la philosophie consistant à éliminer des jugements inutiles (sous tendant l'entière définition de $F_{\leq}^{\top}$) et, donc, considérer comme relation de typage celle définie par la composition de l'algorithme de typage de $F_{\leq}$, et la nouvelle relation de sous-typage. Cependant un autre problème surgit: le système résultant ne satisfait pas la propriété de subject reduction. Considérons le terme $\Lambda X \leq Y.\lambda y^Y.(\lambda x^X.x)y$. Le type déterminé pour ce terme par l'algorithme de typage est $\forall(X \leq Y)Y \rightarrow X$. Le terme en question se réduit en un pas à $\Lambda X \leq Y.\lambda y^Y.y$ pour lequel l'algorithme de typage rend $\forall(X \leq Y)Y \rightarrow Y$, qui est incomparable (pour la règle ($\forall$-new)) avec le type précédent.

Maintenant, considérons ce même système dans lequel les réductions à l'intérieur des Λ-abstractions ne sont pas utilisées; nous éliminons ainsi non seulement d'"inutiles" jugements de typage et de sous-typage mais encore d'"inutiles" réductions. Pour un tel système, dont le typage est décidable (évident, car tout terme bien type possède un seul type, celui déterminé pas l'algorithme) et dont la relation de sous-typage possède toutes les bonnes propriétés établies dans cette section, nous conjecturons la propriété de subject-reduction. Notons que c'est ce système que nous avons implicitement testé en utilisant les implementations de Cardelli et de Pierce et Turner de l'algorithme de typage de $F_{\leq}$ avec la relation de sous-typage modifiée

Quantification bornée avec surcharge

Nous avons déjà montré qu'une méthode qui modifie l'état interne d'un objet de classe C_1

$$m\colon \{..., C_1 \rightarrow C_1, ...\}$$

possède le même problème de perte d'information que la fonction identité de type $C_1 \rightarrow C_1$: dans les deux cas en passant un argument de type strictement plus petit que C_1, on obtient un résultat de type C_1. La solution adoptée pour la fonction identité était de passer à un formalisme du second ordre et d'utiliser ainsi le type $\forall X \leq C_1.X \rightarrow X$. Comme nous l'avons déjà anticipé, nous adoptons pour $\lambda\&$ la même solution. Donc l'idée est de définir un système de types où le message m ci-dessus puisse avoir un type de la forme

$$m\colon \{..., \forall X \leq C_1.X \rightarrow X, ...\}$$

C'est pourquoi, nous définissons $F_{\leq}^{\&}$ où cette dépendance des types est traitée de façon explicite.

Dans un langage de programmation une fonction qui prend comme argument un type et qui pour chaque type exécute un code différent, aurait très probablement la syntaxe suivante :

```
Fun(X:*) = if X≤T1 then exp_1 else
           if X≤T2 then exp_2 else
               ⋮
           if X≤Tn then exp_n
```

Cette fonction exécute l'expression exp_1 si on lui passe un type plus petit ou égal à T_1, exp_2 si le type passé est plus petit ou égal à T_2 et ainsi de suite. S'il y a plus d'un candidat possible on sélectionne parmi eux la branche avec la borne la plus petite.

Dans $F^{\&}_{\leq}$ nous dénotons cette fonction par :

$$(\Lambda X{\leq}T_1.exp_1 \;\&\; \Lambda X{\leq}T_2.exp_2 \;\&\ldots\&\; \Lambda X{\leq}T_n.exp_n)$$

Pour des raisons techniques nous quantifions un type surchargé à l'extérieure de ses parenthèses ; ainsi le type de la fonction ci-dessus sera :

$$\forall X\{T_1.S_1, T_2.S_2, \ldots, T_n.S_n\}$$

où $exp_i\colon S_i$. Comme pour $\lambda\&$ nous avons deux conditions pour sélectionner les types bien formés. La première est une condition de cohérence qui assure que le type diminue toujours, même si au cours de l'exécution on change la branche sélectionnée. Ceci était donné dans $\lambda\&$ par la condition de covariance (0.4). La condition correspondante dans le second ordre impose pour un type surchargé $\forall X\{T_i.S_i\}_{i\in I}$ que pour tout $i, j \in I$ si $C \vdash T_i \leq T_j$ alors $C \cup \{X{\leq}T_i\} \vdash S_i{\leq}S_j$. La deuxième condition doit assurer pour la sélection l'existence d'une branche avec une borne minimum (entre les branches compatibles avec l'argument). Pour le second ordre il ne suffit plus d'imposer que pour tout couple de bornes compatibles T_i et T_j dans $\forall X\{T_i.S_i\}_{i\in I}$ il existe une borne qui en est le plus grand majorant. Le problème est que les bornes peuvent contenir des variables libres. Donc il faut contrôler les conflits pour toute valeur admissible pour ces variables. Pour assurer l'existence de la branche sélectionnée nous imposons deux restrictions : tout d'abord les bornes ne peuvent qu'être soit des types atomiques (des noms de classe) soit des variables bornées par des types atomiques. Ceci nous permet d'éviter tout problème dû à la contra-variance des types flèche. En outre, on demande que les bornes d'un type surchargé satisfassent la condition dite de "$\cap$-closure" (meet-closure) qui est définie comme suit : soit A un type atomique ou une variable bornée par un type atomique; dénotons par $\mathcal{B}(A)$ le type A lui-même dans le premier cas et la borne de la variable A dans le second. La $\cap$-closure impose que pour tout A_i et A_j dans $\forall X\{A_i.S_i\}_{i\in I}$ si $\mathcal{B}(A_i){\Downarrow}\mathcal{B}(A_j)$ alors il existe h dans I tel que A_h est la plus grande borne inférieure de A_i et A_j. Si $\{A_i\}_{i\in I}$ satisfait cette condition (assez restrictive) alors pour tout type T et pour toute substitution σ compatible avec les bornes des variables libres dans $\{A_i\}_{i\in I}$ l'ensemble $\{\sigma(A_i) \mid T \leq \sigma(A_i), i \in I\}$ est soit vide soit possède un plus petit élément.

On définit donc le système de types de $F^{\&}_{\leq}$ en ajoutant aux productions des types de $F_{\leq}$ la production suivante

$$T ::= \forall X\{A_i.T_i\}_{i\in I}$$

en ajoutant aux règles de sous-typage de $F_{\leq}$ la règle suivante:

$$(\{\,\})\qquad \frac{\text{pour tout } i \in I \text{ il existe } j \in J \text{ t.q. } C \vdash A'_i{\leq}A_j \quad C \cup \{X{\leq}A'_i\} \vdash T_j{\leq}T'_i}{C \vdash \forall X\{A_j.T_j\}_{j\in J}{\leq}\forall X\{A'_i.T'_i\}_{i\in I}} \qquad X \notin dom(C)$$

Ensuite nous utilisons la relation de sous-typage ainsi définie pour sélectionner les types qui sont bien formés, c'est-à-dire les (pré-)types qui satisfont la covariance, la $\cap$-closure et dont les bornes ne prennent comme valeurs que des types atomiques.

Nous démontrons pour ce système la cohérence du sous-typage, ainsi que les propriétés d'élimination de la réflexivité et de la transitivité. De ce dernier résultat découle une présentation algorithmique de la relation de sous-typage. L'algorithme ainsi obtenus est une procédure de semi-décision : $F_{\leq}^{\&}$, hérite de $F_{\leq}$ l'indécidabilité du sous-typage.

Les termes de $F_{\leq}^{\&}$ sont obtenu en ajoutant aux productions des termes de $F_{\leq}$ les productions suivantes :

$$a ::= \varepsilon \mid (a \&^{I} a) \mid a[A]$$

où $a[A]$ indique l'application d'une fonction surchargée (pour la différencier de l'application d'une fonction polymorphe que nous notons par $a(T)$) et I un index de la forme $[A_1.T_1 \parallel \ldots \parallel A_n.T_n]$. Les index sont utilisés pour la sélection et le typage. Les nouveaux termes sont typés de la façon suivante :

$[\varepsilon]$ $$C \vdash \varepsilon{:} \forall X\{\}$$

$[\{\}\text{Intro}]$ $$\frac{C \vdash a{:}\, T_1 {\leq} \forall X\{A_i.T_i\}_{i\leq n} \quad C \vdash b{:}\, T_2 {\leq} \forall (X {\leq} A)T}{C \vdash (a \&^{[A_1.T_1 \parallel \ldots \parallel A_n.T_n \parallel A.T]} b){:}\, \forall X(\{A_i.T_i\}_{i\leq n} \cup \{A.T\})}$$

$[\{\}\text{Elim}]$ $$\frac{C \vdash a{:}\, T \qquad C \vdash A_j = min_{i\in I}\{A_i | C \vdash A {\leq} A_i\}}{C \vdash a[A]{:}\, T_j[X := A]} \qquad \mathcal{B}(T) = \forall X\{A_i.T_i\}_{i\in I}$$

Ces règles vont s'ajouter à celles du typage pour les termes de $F_{\leq}$. La cohérence du typage dérive directement de la cohérence du sous-typage. Enfin nous augmentons les notions de réduction de $F_{\leq}$ par la règle suivante :

$(\beta_{\{\}})$ Si $A, A_1 \ldots, A_n$ sont fermés alors

$$C \vdash (a \&^{[A_1.T_1 \parallel \ldots \parallel A_n.T_n]} b)[A] \;\triangleright \begin{cases} b(A) & \text{si } A_n = min_{1\leq i\leq n}\{A_i | C \vdash A {\leq} A_i\} \\ a[A] & \text{sinon} \end{cases}$$

La condition de fermeture des types utilisés pour la sélection correspond à l'utilisation de la liaison tardive pour ce calcul. Le calcul ainsi défini satisfait les propriétés de subject-réduction généralisée et de confluence. Nous conjecturons que $F_{\leq}^{\&}$ est aussi fortement normalisant.

Nous avons vu qu'en remplaçant dans $F_{\leq}$ la règle de sous-typage pour la quantification universelle par la règle ($\forall$-new) nous obtenons un système de sous-typage décidable. Il est donc légitime de se poser la question de savoir si en effectuant le même changement dans $F_{\leq}^{\&}$ on obtient un calcul qui à la fois possède un sous-typage décidable et soit assez expressif pour modéliser la programmation orientée objets. La réponse est affirmative : en fait on ne rencontre aucun problème particulier à reparcourir les preuves de $F_{\leq}^{\top}$ de terminaison du sous-typage et d'élimination de la transitivité pour le système ainsi obtenu.

Il est intéressant de noter que la décidabilité est obtenue par la seule modification de la règle ($\forall$), tandis que la règle de sous-typage ($\{\}$) pour les types surchargés demeure inchangée. En effet, on pourrait s'attendre à ce que la borne dans la prémisse de droite de la règle ($\{\}$)

doive être changée de A_i' à Top. Ceci n'est pas nécessaire pour la décidabilité : en fait, les bornes utilisées dans la quantification surchargée sont de loin moins générales que celles de la quantification universelle, car dans ces dernières Top peut y apparaître, d'où l'indécidabilité (dans un certain sens les types surchargés de ce système—que nous notons $F_{\leq}^{\&\top}$—forment un système qui satisfait la restriction de [KS92]).

Nous conjecturons que $F_{\leq}^{\top}$ hérite de toutes les bonnes propriétés de la relation de sous-typage de $F_{\leq}^{\top}$, en particulier l'existance d'une plus grand borne inférieure et d'une (unique) plus petite borne supérieure. Ce qui est certain c'est que ce système possède toutes les mauvaises propriétés de $F_{\leq}^{\top}$, notamment il ne satisfait pas la proprieté d'existance d'un type minimal et le système sans la règle de subsumption, ne satisfait pas la propriété de "subject-reduction". Cependant, si la conjecture posée à la fin de la section précédente est correcte elle impliquera automatiquement la même propriété pour $F_{\leq}^{\&\top}$, c'est-à-dire que le système, sans la règle de subsumption et sans réductions impliquant des variables libres de type, satifera non seulment la décidabilité mais encore la propriété de subject-reduction.

Surcharge de second ordre et programmation orientée objets

Nous allons maintenant montrer comment utiliser le formalismes du second ordre que nous avons défini dans la séction précédente, pour la modélisation des langages objets. Tout les examples de cette section sont typables tant dans $F_{\leq}^{\&}$ que dans $F_{\leq}^{\&\top}$ (sans ou avec subsumption).

Comme dans la section sur $\lambda\&$ nous utilisons des types atomiques pour les noms des classes. À chaque type atomique est associé un type-représentation qui décrit les variables d'instance de la classe. Une relation de sous-typage peut être définie sur ces types atomiques, pourvu qu'elle respecte le sous-typage des types-représentation.

Une fois de plus, un message est un identificateur de fonction surchargée ; mais cette fois la méthode à exécuter est choisie sur la base du type passé *comme argument*, qui sera la classe de l'objet auquel le message est envoyé. Ainsi l'envoi du message *mesg* à un objet *a* de classe *A* est modélisé par $(mesg[A])a$.

Une fois de plus nous considérons les types atomiques *3DPoint* et *2DPoint* avec *3DPoint* $\leq$ *2DPoint* (les types-représentation sont toujours les mêmes). Le message `norme` devient donc

$$\begin{array}{rl} \mathit{norme} \equiv & (\ \Lambda \mathit{Mytype}\!\leq \mathit{2DPoint}\ .\lambda \mathit{self}^{\ \mathit{Mytype}}.\sqrt{\mathit{self.x}^2 + \mathit{self.y}^2} \\ & \&\Lambda \mathit{Mytype}\!\leq \mathit{3DPoint}\ .\lambda \mathit{self}^{\ \mathit{Mytype}}.\sqrt{\mathit{self.x}^2 + \mathit{self.y}^2 + \mathit{self.z}^2} \\ &) \end{array}$$

dont le type est

$$\forall \mathit{Mytype}.\{\mathit{2DPoint.Mytype} \to \mathrm{Real}, \mathit{2DColorPoint.Mytype} \to \mathrm{Real}\}$$

Nous avons utilisé la variable *self* pour dénoter, à l'intérieur d'une méthode, le destinataire du message et, en conformité avec la notation de [Bru92], la variable *Mytype* pour dénoter le type du destinataire. Il faut noter que contrairement à [Bru92] nous n'avons pas besoin de récursion.

Le message d'initialisation aura dans $F_{\leq}^{\&}$ le type suivant :

$$initialise : \forall Mytype\{2DPoint.Mytype \rightarrow Mytype, 3DPoint.Mytype \rightarrow Mytype\}$$

qui est bien formé (il satisfait de manière triviale les conditions de bonne formation). Mais même si nous avions défini *initialise* de façon qu'il aie le type

$$initialise : \forall Mytype\{2DPoint.Mytype \rightarrow 2DPoint, 3DPoint.Mytype \rightarrow Mytype\}$$

nous aurions encore obtenu un type bien formé. En fait, la condition de covariance demande que

$$\{Mytype{\leq}3DPoint\} \vdash Mytype \rightarrow Mytype \leq Mytype \rightarrow 2DPoint$$

ce qui est vrai dans ce cas. Plus généralement, si le message m a été défini dans les classes B_i (pour $i \in I$) alors le type de m a la forme $\forall Mytype\{B_i.Mytype \rightarrow T_i\}_{i\in I}$. Si $B_h \leq B_k$ alors la méthode définie pour m dans la classe B_h a *masqué* (overridden) celle définie dans B_k. Étant donné que *Mytype* est le même dans les deux branches (c'est ici que se justifie le fait d'utiliser un unique quantificateur pour toutes les branches), la condition de covariance se réduit à prouver que

$$\{Mytype{\leq}B_h\} \vdash T_h \leq T_k$$

et donc que la méthode masquante rend un type plus petit ou égal à celui de la méthode masquée. L'exemple avec *initialise* montre pourquoi il est important de vérifier cette condition sous l'hypothèse que *Mytype* soit plus petit que la borne la plus petite. Au contraire si une méthode rend un résultat de type *Mytype* alors une méthode la masquant doit aussi retourner un résultat de type *Mytype* (et pas le nom de la classe où la méthode a été définie parce que par héritage ceci pourrait être un type plus grand que la valeur actuelle de *Mytype*).

L'intérêt du second ordre réside dans l'élimination du problème de perte d'information : ceci avait lieu pour le message *efface* de la page 23. En fait si l'on envoyait ce message à un objet de classe $3DPoint$ le résultat de l'envoi avait le type $2DPoint$. Par contre dans le second ordre la branche pour *2DPoint* dans *efface* aura la définition suivante :

$$efface \equiv (\ \Lambda Mytype{\leq}2DPoint.\lambda self^{MyType}.\langle self \leftarrow x = 0\rangle\ \& \ldots)$$

dont le type est

$$\forall Mytype.\{2DPoint.Mytype \rightarrow Mytype, \ldots\}$$

L'envoi du message *efface* à un objet b de type *3DPoint* est traduit en $(efface\,[3DPoint])(b)$. Pour les règles [{}Elim] et [→Elim] cette expression possède le type *3DPoint* ; nous n'avons donc pas de perte d'information.

Comme pour λ& la redéfinition ou l'ajout d'une méthode correspondent à la concaténation d'une nouvelle branche à une fonction surchargée. De même, les messages sont des valeurs de première classe ; un exemple trivial d'utilisation de cette propriété est donné par la définition d'une fonction qui envoie un message à self en commençant la recherche de la méthode de la classe C (ceci ressemble à la fonction super) :

let super_C $= \lambda m^{\forall X\{C.T\}}.m[C]self$

Cette fonction accepte comme argument tout message pouvant être envoyé à un objet de

classe C (l'expression est bien typée pourvu que *Mytype* $\leq C$). Il faut noter que la possibilité d'appliquer les termes aux types, donne un plus grande contrôle sur le calcul, tout comme l'introduction des coercitions explicites l'avait donné dans $\lambda\&$.

Le calcul que nous venons de décrire est assez simple, surtout à cause des restrictions sur les bornes d'une fonction surchargée. Toutefois il n'y a aucune difficulté à modifier la définition de $\cap$-closure pour permettre comme bornes des produits cartésiens de classes (pour modéliser le dispatch multiple) et des structures d'ordre autres que le demi-treillis (pour modéliser l'héritage multiple dans toute sa généralité : voir la condition (0.6) page 28). Il est intéressant toutefois de noter qu'en cas de dispatch multiple la condition de covariance a une signification différente qu'en $\lambda\&$. Nous rappelons qu'en $\lambda\&$ le message *equal* ne pouvait pas être typé par $\{2DPoint \to 2DPoint \to Bool, 2DColorPoint \to 2DColorPoint \to Bool\}$ car ce dernier n'est pas un type bien formé. C'est pourquoi, dans $\lambda\&$ nous définissions

$$equal : \{(2DPoint \times 2DPoint) \to Bool, (2DColorPoint \times 2DColorPoint) \to Bool\}$$

Par contre, en passant au second ordre le type $\forall X\{2DPoint.X \to X \to Bool, 2DColorPoint.X \to X \to Bool\}$ est bien formé. Toutefois pour sélectionner la branche correcte il faut passer entre les types des deux arguments celui qui est le plus grand. Ceci n'est pas ce qu'on voudrait : on aimerait passer à la fonction les deux types des arguments et laisser au système la tâche de sélectionner la branche correcte. Ceci peut être obtenu par des multi-méthodes, en définissant *equal* de sorte qu'elle possède le type suivant :

$$\forall X\{2DPoint \times 2DPoint.X \to Bool, 2DColorPoint \times 2DColorPoint.X \to Bool\}$$

$F^{\&}_{\leq}$ ne capture pas la puissance de tous les langages orientés objets ; cependant il possède certaines caractéristiques qui leur manquent. Donc on peut envisager d'enrichir les langages existants par ces caractéristiques ; par exemple on pourrait définir un nouveau langage à objets qui fournisse soit la paramétricité soit la surcharge, où les classes puissent être passées comme argument aux fonctions ; ces fonctions pourraient alors exécuter des codes différents selon la classe reçue comme argument. On peut imaginer plusieurs applications d'un tel mécanisme : par exemple, imaginons de devoir écrire une routine générale d'installation pour des logiciels qui travaillent sur des machines différentes. Imaginons de classifier les logiciels en mathématiques et graphiques, et les machines en machines noir et blanc et machines couleur. Très probablement cela correspondrait aux classes suivantes : *GraphSW, MathSW* $\leq$ *Software* et *Color, B&W* $\leq$ *Machine* et la routine d'installation aura le type

$$install : \forall(M \leq Machine)\forall(S \leq Software)(M \times S) \to \ldots$$

Le corps de cette routine inclura certaines parties communes à toutes les machines et à tous les logiciels, et certaines parties spécialisées selon l'espèce des arguments. Cette spécialisation sera obtenue en utilisant des fonctions de type :

$$\begin{aligned}\forall X\{&Software \times Machine. ...,\\ &GraphSW \times B\&W. ...,\\ &GraphSW \times Color. ...,\\ &MathSW \times B\&W. ...,\\ &MathSW \times Color. ...\}\end{aligned}$$

Parmi les limites de modélisation du second ordre ou, plus précisément, de la $\cap$-closure il faut citer l'impossibilité de modéliser les classes génériques du langage Eiffel [Mey88]. Une classe est générique quand elle est paramétrée par une variable de type. Par exemple, si X est une variable de type, alors nous aimerions pouvoir définir une classe $Stack[X]$ avec deux méthodes $pop\colon X$ et $push\colon X \to ()$, et puis obtenir une pile d'entiers en instantiant X de la manière suivante : `new(Stack[Int])`. Il ne semble pas être trop difficile d'affaiblir la définition de $\cap$-closure pour permettre parmi les bornes des fonctions surchargées des constructeurs monotones de types. Mais nous n'avons pas d'idée simple pour traiter les constructeurs non-monotones. Toutfois ceci n'est pas une limite de l'approche, mais seulment de sa formalisation actuelle.

Conclusion

Nous concluons cette présentation en situant notre travail dans un contexte plus général et en suggérant les perspectives de notre recherche future.

Théorie de la Preuve

Au début de ce chapitre nous avons rappelé la classification introduite par [Str67], subdivisant le polymorphisme entre "ad hoc" et paramétrique. Ce dernier est à son tour réparti en polymorphisme (paramétrique) *implicite*, où la quantification des types est une opération méta-linguistique qui exprime l'utilisation de schémas de types dans la théorie de la preuve correspondante [Hin69, Mil78], et en polymorphisme (paramétrique) *explicite* où cette même quantification est exprimée par un opérateur linguistique et où la multiplicité des instances "prouvées" par un terme est linguistiquement exprimée par un unique type syntaxique [Gir72, Rey83].

Nous introduisons pour le polymorphisme "ad hoc" la même classification, mais avec un sens légèrement différent : nous définissons comme *implicite* le polymorphisme "ad hoc" où la sélection de la branche est basée sur le type *de* l'argument ; nous définissons comme *explicite* le polymorphisme "ad hoc" où la sélection de la branche est basée sur le type *passé en* argument.

L'emploi de la même terminologie que pour le polymorphisme paramétrique est justifié par le fait que le polymorphisme ad hoc *implicite* peut être caractérisé par une quantification méta-linguistique sur les (sous-types des) "types d'entrée" d'une fonction surchargée, tandis que dans le polymorphisme ad hoc *explicite* cette quantification est promue au niveau linguistique. Ceci devient encore plus évident si l'on utilise la notation des types surchargés généralisés introduite à la page 35 ; le type surchargé $\{U_i \to out(U_i)\}_{i\in I}$ peut être considéré comme une notation alternative du schéma de type suivant :

$$\forall\alpha \in \widehat{\{U_i\}}_{i\in I}.\alpha \to \widehat{out}(\alpha) \tag{0.8}$$

Ainsi un terme de type $\{U_i \to out(U_i)\}_{i\in I}$ est implicitement polymorphe dans le sens où il possède plusieurs types obtenus en instanciant le schéma (0.8). Pour typer l'application d'une fonction $M : \forall\alpha \in \widehat{\{U_i\}}_{i\in I}.\alpha \to \widehat{out}(\alpha)$ on utilisera une instance particulière de son schéma :

$$[\{\}\text{ELIM}] \qquad \frac{M{:}\, U \to \widehat{out}(U) \qquad N{:}\, U}{M{\bullet}N{:}\, \widehat{out}(U)}$$

Il faut noter que la variable générique α peut varier sur l'ensemble des sous-types des types d'entrée. Ceci se manifeste au niveau linguistique dans le traitement de la sémantique par la notion de *complétion* : la quantification méta-linguistique est transformée en l'expansion de ses instances, afin d'être interprétée comme un produit indexé sur les types.

Nous pouvons donc classifier les différents calculs sur la base de la présence et du type éventuel des différentes formes de polymorphisme. Nous nous trouvons donc en présence de neuf classes de langages, dont certaines (dénotées par (1) et (2)) sont vides :

	polymorphisme	
	paramétrique	**ad hoc**
λ-calc. simplement typé	absent	absent
λ&-calcul	absent	implicite
(1)	absent	explicite
ML	implicite	absent
ML+ fonctions génériques	implicite	implicite
(2)	implicite	explicite
Système $F_{(\leq)}$	explicite	absent
$\lambda\&+F_{(\leq)}$	explicite	implicite
$F_{\leq}^{\&}$	explicite	explicite

Il faut noter que les deux formes de polymorphisme sont syntaxiquement distinguées car elles sont implémentées par des termes différents; ainsi dans la table ci-dessus chaque colonne se réfère à la caractéristique du calcul pur des termes correspondant (car dès que l'on introduit le polymorphisme ad hoc, le système ne satisfait plus la paramétricité dans le sens de [MR91, Rey83]).

Enfin, le lecteur ne devrait pas être étonné que la table ci-dessus mélange calculs avec et sans sous-typage : en fait, tous les calculs que nous avons étudiés possèdent le sous-typage (car cela implique un gain remarquable en puissance expressive). Toutefois cette classification est également valable pour les langages sans sous-typage ; et pour adapter cette étude à ces langages il suffit d'utiliser l'égalité syntaxique pour le sous-typage (i.e. $S \leq T \Leftrightarrow S \equiv T$), avec toutes les simplifications que cela entraîne.

Programmation orientée objets

Nous avons déjà remarqué tout au long de cette présentation que notre modèle donne des fondements à un style de programmation orientée objets différent de celui modélisé par les enregistrements. C'est pourquoi ce modèle est capable de capturer aisément des mécanismes différents et d'en suggérer d'autres totalement nouveaux.

Nous avons aussi montré que le fait d'étudier "le revers de la médaille" nous a permis de mieux cerner le fonctionnement des modèles à enregistrements (tel est le cas pour la covariance/contra-variance) ainsi que leurs limites.

Nous voulons conclure ce point par quelques mots sur l'héritage, car dans cette thèse nous avons limité notre étude uniquement à la forme d'héritage qui dérive de l'utilisation du sous-typage. En effet, dans la programmation objets le sous-typage et l'héritage caractérisent deux hiérarchies différentes, bien que très liées l'une à l'autre. Le sous-typage concerne l'exécution des expressions, puisqu'il établit dans quel cas les expressions d'un type donné peuvent être utilisées là où une expression d'un type différent est attendue. L'héritage se rapporte à la définition des opérations pour les expressions d'un type donné car il établit dans quel cas les expressions d'un type donné peuvent utiliser certaines opérations initialement définies pour un type différent.

Selon cette définition le sous-typage implique l'héritage car si les expressions d'un type donné peuvent se substituer à celles d'un autre type alors ils peuvent aussi utiliser toute opération définie pour ce type. Toutefois la réciproque n'est pas vérifiée car dans certains cas on peut avoir la réutilisation du code mais pas la substitutivité. Ce dernier point est illustré simplement par la méthode qui effectue la copie du destinataire [16] :

```
class C
  {...}
   copy = self
  [[ copy: Mytype ]]
```

Évidement le code de `copy` peut être utilisé par n'importe quelle classe, mais ceci n'implique pas que toute classe puisse être un sous-type de `C`. Si l'on ne dispose que du sous-typage, et que l'on souhaite définir une nouvelle classe `C'` entièrement différente (pour l'état interne) de `C` mais possédant une méthode `copy`, alors la seule solution possible est de redéfinir à nouveau `copy` dans `C'`. Si, au contraire, l'on possède un mécanisme général d'héritage alors il devient possible d'exprimer que `C'` hérite (de tous les codes) des méthodes de `C`. Ceci peut s'écrire de la façon suivante :

```
class C' is ... inherits from C
  {...}
    :
    :
```

où `C'` hérite de `C` sans en être sous-type. Ceci peut intuitivement être modélisé dans $F^{\&}_{\leq}$ en admettant parmi les bornes d'une fonction surchargée les types union.

Par exemple la méthode `copy` ci-dessus pourrait être modélisée par l'expression suivante :

$$\mathit{copy} \equiv (\varepsilon \& \Lambda \mathit{Mytype} \leq C \cup C'.\lambda \mathit{self}^{\mathit{Mytype}}.\mathit{self}) : \forall \mathit{Mytype}\{C \cup C'.\mathit{Mytype} \rightarrow \mathit{Mytype}\}$$

Le type union indique que la branche en question est partagée par C et C'. Cette branche sera sélectionnée chaque fois que, lors de l'application de *copy* à un type A, l'un de ces deux types (C ou C'), se trouvera être le plus petit des types supérieurs apparaissant dans une borne.

Il est nécessaire donc de vérifier séparément la branche pour chaque type, i.e.

$$\frac{X \leq C \vdash a : T \qquad X \leq C' \vdash a : T}{\vdash \Lambda X \leq C \cup C'.a : \forall (X \leq C \cup C')T}$$

[16] Pour une explication precise de la notatio utilisée, voir le chapitre 1

Ainsi, $\cup$ dénote un "ou-exclusif" plutôt que l'union. Il est important de noter que dans ce cas l'héritage est un mécanisme spécifique à une opération dont le code doit être partagé ; ceci n'implique pas le partage de toute opération définie pour le type concerné. Donc dans notre formalisme nous pouvons introduire une nouvelle forme d'héritage, partiel, dont la caractéristique est qu'un type hérite seulement de *certaines* opérations définies pour un autre type. L'écriture

```
class C' is ... inherits from C
```

en haut indiquait que `C'` héritait de toute opération définie pour `C`. Mais par héritage partiel nous pouvons spécifier que `C'` n'hérite de `C` que de la méthode `copy` (et de rien d'autre). Une syntaxe possible pourrait être:

```
class C' is ...
  {...}
   copy = inherited from C
  [[ .. .]]
```

Dans ce cas la définition de *copy* dans $F_{\leq}^{\&}$ donnée ci-dessus est encore valable. Il faut remarquer que cet héritage partiel est impossible dans les modèles par enregistrements où toutes les méthodes d'une classe doivent obligatoirement être héritées.

Cette nouvelle vue de l'héritage nous suggère une autre généralisation de ce mécanisme. Jusqu'à présent l'héritage a été restreint aux classes (i.e. aux types atomiques). Or, il n'y a aucune raison apparente de ne limiter les unions qu'aux types atomiques. Par exemple en $F_{\leq}^{\&}$ nous pourrions envisager d'admettre des bornes formées par l'union de produits de types atomiques, obtenant d'emblée l'héritage pour les multi-méthodes. Mais des unions de types encore plus générales pourraient être étudiées.

Pour compléter cette étude on pourrait envisager d'introduire des types intersection à l'intérieur des bornes. Ceci permettrait de définir de manière aisée une condition de $\cap$-closure pour types d'ordre supérieur, ce qui nous permettrait de modéliser les classes génériques.

Au delà de la programmation objets

Dès le début nous avons affirmé que la combinaison de la surcharge et de la liaison tardive permet un très haut niveau de programmation incrémentale et de réutilisation du code. Ces caractéristiques ne sont pas, et ne doivent pas être, une exclusivité de la programmation orientée objets. C'est pourquoi il devient très intéressant d'essayer de les exporter à d'autres paradigmes, moyennant l'introduction de la surcharge et de la liaison tardive. Au moment de la rédaction de cette thèse nous avons déjà commencé à appliquer ces mécanismes aux langages des modules ; plus précisément nous sommes en train d'étudier une extension du système de modules de SML par des foncteurs surchargés combinés avec la liaison tardive. Toutefois il n'y a pas de limite apparente à l'application de ces techniques, et une extension à d'autres paradigmes (systèmes concurrents ou programmation logique, par exemple) peut également être envisagée.

Introduction

a quien leyere: *Si las paginas de este libro consienten algún verso feliz, perdóneme el lector la descortesía de haberlo usurpado yo, previamente. Nuestras nadas poco difieren; es trivial y fortuita la circustancia de que seas tú el lector de estos ejercicios, y yo su redactor*

Jorge Luis Borges
Fervor de Buenos Aires (1923)

An important distinction has been extensively used in language theory for the last two decades, between parametric (or universal) polymorphism and "ad hoc" polymorphism [Str67] (see also [CW85]). Parametric polymorphism allows one to write a function whose code can work on different types, while using "ad hoc" polymorphism it is possible to write a function which executes different code for each type. Both the Proof Theory and the semantics of the first kind of polymorphism have been widely investigated by many authors, on the grounds of early work of Hindley, Girard, Milner and Reynolds, and developed into robust programming practice. The second kind, usually known as "overloading", has had little theoretical attention, with the notable exception of [WB89], [MOM90] and [Rou90]; consequently, its wide use has been little affected by any influence comparable to the one exerted by implicit and explicit polymorphism in programming.

This is due, probably, to the fact that the traditional languages offer a very limited form of overloading: in most of them only predefined functions (essentially arithmetic operators defined on integers and reals and input/output operators) are overloaded, while in the relatively few languages where the programmer can define overloaded functions their actual meaning is always decided at compile time. This form of overloading can be easily understood as a form of syntactic abbreviation which does not significantly affect the underlying language.

Indeed we understand that the real gain of power with overloading happens only when one computes with types: to exploit the whole potentiality of overloading, types must be computed during the execution of the program and the result of this computation must affect the final "value" of the whole execution. Overloading resolution performed at compile time does not operate any computation on the types; the selection of the code to execute is nearly reduced to a macro expansion. It is true that in languages with a "classic" type discipline, delaying to run time the choice of the code to execute would not have any effect, since types do not change during the computation and thus the choice would be always the same; indeed,

these languages lack any notion of computation on the types. However, there exists a wide class of programming languages in which types evolve during the computation of a program. These are the languages that use subtyping hierarchies: in this case, types change during the computation, notably they decrease. In this sense types are computed during the execution of the program, and the computation does not correspond to the calculation of a distinguished term[17], but it is intrinsic to the stepwise reduction of the program. Nevertheless we can use it to affect the final value of the execution of the program, by performing the selection of the code of an overloaded function on the types at a given moment of the execution.

Thus in the languages that use a subtyping relation one can differentiate at least two distinct disciplines for the selection of the code:

1. The selection is based on the least type information: the types of the arguments at compile time are used. We call this discipline *early binding*.

2. The selection is based on the maximal type information: the type of the normal forms of the arguments are used. We call this discipline *late binding*.

As we said before, the introduction of overloading with early binding does not affect significantly the underlying language. Though, the ability to define new when combined with subtyping and with late binding, highly increase the possibilities of a language, since it essentially allows a high level of code reusability and an incremental style of programming. The intuitive idea is that one can apply an overloaded function to the formal parameters of an outer (standard) function and leave to the system the task of deciding which code to apply, according to the type of the actual parameters of the outer function. This must be performed at run time, more precisely at least after the substitution of the formal parameters by the actual ones. Without late binding, it would be necessary to define also the outer function as an overloaded one, and its body should have been duplicated in every branch[18], while by late binding it is shared. For example, suppose we have three different types A, B and C with $B, C \leq A$, and an overloaded function f, composed of three different codes f_A, f_B and f_C, one for each type. Then imagine we define a function g in whose body f is applied to the formal parameter x of type A; using the contexts of λ-calculus (i.e. λ-terms with a "hole") this corresponds to the following definition

$$g = \lambda x{:}\, A.\mathcal{C}[f(x)] \tag{0.9}$$

where $\mathcal{C}[\,]$ denotes a context. If early binding is used then, since $x{:}\,A$, the code of f for A is always executed; that is, the function (0.9) is equivalent to

$$\lambda x{:}\, A.\mathcal{C}[f_A(x)]$$

But by subtyping g accepts also arguments of type B or C; with early binding, the only way to use the code of f defined for the type of the actual parameter of g, would be to define g as an overloaded function of three branches:

$$\begin{array}{rcl} g_A & = & \lambda x.\mathcal{C}[f_A(x)] \\ g_B & = & \lambda x.\mathcal{C}[f_B(x)] \\ g_C & = & \lambda x.\mathcal{C}[f_C(x)] \end{array} \tag{0.10}$$

[17]...at least in most languages

[18]We call a *branch* every distinct piece of code composing an overloaded function.

If late binding is used then the code of f is chosen only when the formal parameter x has been substituted by the actual parameter. Thus with late binding the definition of g in (0.9) is equivalent to the definition in (0.10). In other terms, by late binding the function g in (0.9) is implicitly an overloaded function with three branches; thanks to late binding these virtual branches share the code $\mathcal{C}[\,]$ (or, if you prefer, the virtual branches for B and C reuse the code $\mathcal{C}[\,]$ defined for A).

In this thesis we begin a theoretical analysis, and thus a "uniform and general" one, of this richer kind of overloading. However we do not present a general treatment for overloaded functions, but we develop to a great extent a purely functional approach focussed on the study of some features of object-orientedness, namely message-passing and subtyping, in the setting of a truly type dependent calculus. However the interest of this study does not stop with object-oriented languages. Overloading with late binding can be integrated into different formalisms in order to enrich them with the properties of code reusing that characterize it (at the moment of the editing of this thesis we are studying its integration in the module system of SML, in database programming languages and in ML). Also it turns out that the peculiar "type dependency" of overloading, and its blend with subtyping possesses a remarkable theoretical interest.

Indeed, "type dependency" (the fact that terms and values may depend on types) and the role played by the distinction between run-time and compile-time types are the peculiar properties of the various calculi of this thesis. The multifarious (higher order) calculi, such as Girard's System F and its extensions, allow abstraction w.r.t. type variables and the application of terms to types; but the "value" of this application does not truly depend on the argument type, and more generally the semantics of an expression does not depend on the types which appear in it. Indeed, this "parametricity" or "type-erasure" property plays a crucial role in the basic proof-theoretic property of these calculi: the normalization (cut-elimination) theorem. In the semantic interpretations, this essential type independence of computations is understood by the fact that the meaning of polymorphic functions is given by essentially constant functions (we will say more about this in the introduction of chapter 9 and in the conclusion of this thesis. See also [Lon93]).

On the other hand, it is clear that overloaded functions express computations which truly depend on types, as different branches of code (i.e. possibly unrelated terms) may be applied on the basis of input types. Thus we are in presence of a new kind of polymorphism: parametricity characterizes the code that works on many different types; overloading the specialization with a different code for each different type. This novelty is clearly felt when one tries to study the semantics: existing models no longer work and the special *mélange* of overloading, late binding and subtyping poses new mathematical challenges (see chapter 6).

However the main motivation of this thesis comes from considering overloading as a way to interpret message-passing in object-oriented programming. Let us be more specific. In object-oriented languages the computation evolves on objects. Objects are programming

object	
internal_state	
`message_1`	method_1
⋮	⋮
`message_n`	method_n

Figure a.
Objects as records.

message_i	
`class_name_1`	method_1
⋮	⋮
`class_name_n`	method_n

Figure b.
Messages as overloaded functions.

items grouped in *classes* and possessing an internal state that may be accessed and modified by sending *messages* to the object. When an object receives a message it invokes the method (i.e. the code) associated with that message. The association between methods and messages is described by the class the object belongs to.

There are two possible ways to see message-passing: the first approach consists in considering objects as arrays that associate a method with each message. Therefore when a message `m` is passed to an object `obj` then the method associated with `m` in the object `obj` is looked for. In this approach, an object has the form shown in Figure a. This first point of view has been extensively studied and corresponds to the "objects as records" analogy [Car88]; there objects are records whose labels are the messages and whose fields contain the associated methods. Thus message passing corresponds to the field selection.

The second approach to message-passing is to consider messages as identifiers of special functions. In the context of typed languages, if one assumes that the type of an object is (the name of) its class then, as shown in Figure b, messages are identifiers of overloaded functions: depending on the class (or more generally, the type) of the object the message is passed to, a different method is chosen. In this way, in a sense, we reverse the previous situation: instead of passing messages to objects we now pass objects to messages.

At first sight this different approach seems to have some advantages w.r.t. the "objects as records" paradigm, at least in a proof-theoretical study of the typed case. This happens for *multi-methods* and multiple *dispatch*, or for the *logical independence* of the persistent data as in the database programming languages[19] . Furthermore it clarifies the role of *covariance* and *contravariance* in the subtyping rules for the methods.

On the other hand, other problems arise when overloaded functions are used to define methods: especially to model the dynamic definition of new classes and to obtain a high level of encapsulation; this last point for example renders this formalism inadequate to model objects in wide-area distributed systems, since there objects are required to encapsulate the operations that can affect them (for obvious reasons of efficiency and security).

A closer look to the model based on overloading and late binding persuades us that it accounts for a style of object-oriented programming that is rather different from the one modeled so far by records. The problem is that the term "object-oriented" groups so many different techniques under the same hat. Indeed under this adjective cohabit many different

[19]In the sense that it is possible to add new methods for the objects of a certain class without perturbating the definition of their type, and thus the well-typing of the applications written for the old schema.

programming styles whose affinity is stated by the terms “object”, “message passing”and “inheritance”. We feel that pushing the similarity further including other magic words like “encapsulation” or “modularity” would exclude significant classes of languages (e.g. CLOS for the modularity, Simula for the encapsulation); we believe that such additional words partition the set of object-oriented languages into the different styles that compose it.

The type-theoretic research so far focused on the partition characterized by the keywords “method encapsulation” and modeled by records: since 1984, when Luca Cardelli initiated the typed foundation of object-oriented programming, all the theoretic studies started from the assumption that the methods of an object are “encapsulated” inside it. This excluded some features like “multi-methods”, “multiple dispatching” present in some object-oriented languages but which did not fit the model. The efforts of modifying the existing models to include these and other features led to uneasy extensions of the record based models.

In the beginning of our research we believed that the existing models were not powerful enough to express these features. Thus we started to seek a brand new model. Starting from some ideas of [Ghe91] we laid the basis of this model by the definition of the $\lambda\&$-calculus [CGL92b]. But by a closer look to the mechanisms that we featured we realized that we had modeled a programming style completely different to the one modeled by records. The model of records was not deficient, but orthogonal to the one we had proposed: distinct models for distinct features.

The “new” style of object-oriented programming we had modeled corresponded to the one based on the *generic functions*. It is an interesting fact that starting from a purely theoretical approach we arrived to a programming model which already existed. Actually we soon realized that to the relation

$$\begin{array}{rcl} \text{record} & \leftrightarrow & \text{object} \\ \text{field} & \leftrightarrow & \text{method} \\ \text{label} & \leftrightarrow & \text{message} \end{array}$$

of the “object as records” approach corresponded the relation

$$\begin{array}{rcl} \text{overloaded function} & \leftrightarrow & \text{generic function} \\ \text{branch} & \leftrightarrow & \text{method} \end{array}$$

of our approach. In both cases in the passage one gained a typing discipline (and a provably correct one!).

As in the case of records the gains of having defined a typed model did not stop there: the study of the model suggested to us the introduction of new features in the object-oriented languages (e.g. first class messages), and the generalization or redefinition of the existing ones (e.g. the explicit coercions).

This thesis is a comprehensive study of overloading with late binding under the peculiar perspective of the definition of this new model, and the exposition of the practical impact that this model has in the definition of object-oriented languages and their type disciplines.

The thesis is composed by two main parts: in the first part we focus on overloading whose type dependence is *implicit* in the sense that the selection of the branch is based on the type *of* the argument of the overloaded function. The second part is devoted to the study of the

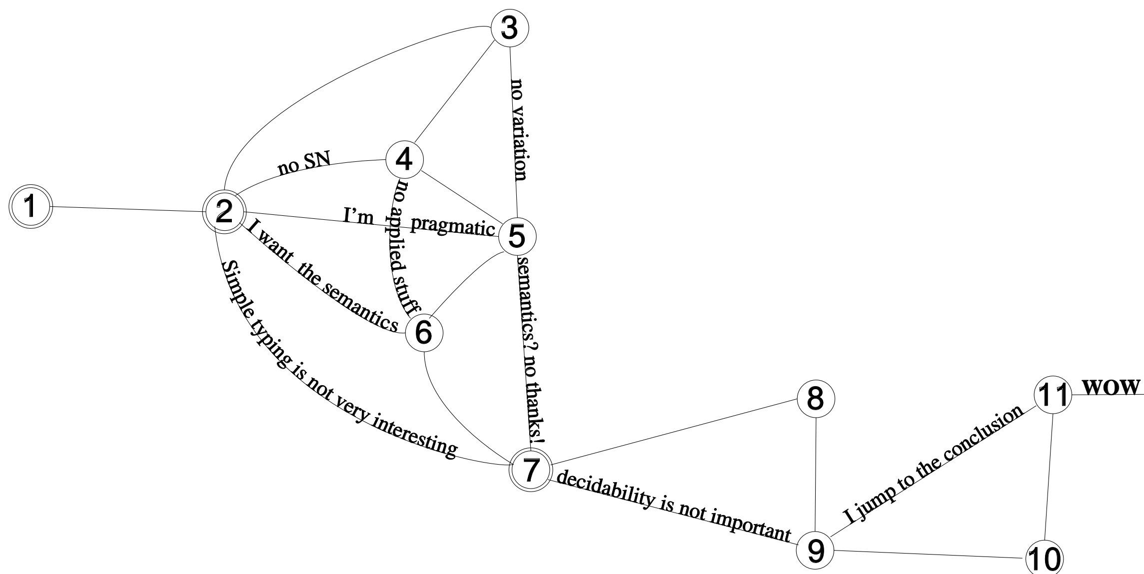


Figure 0.3: Paths of reading

explicit type dependence of the overloading in the sense that the selection of the branch is based on the type that *is* the argument of the overloaded function.

Thus we start in chapter 1 by presenting the basic ideas of object-oriented programming. This is done by giving an informal presentation of a toy object-oriented language and of its type discipline. We use this toy language also to gradually introduce the basic ideas of our model; thus it offers only a partial view of object-oriented programming. In chapter 2, we define the $\lambda\&$ calculus, an extension of the simply typed λ-calculus to model overloaded functions with late binding; we prove that it enjoys the properties of Church-Rosser and subject reduction, and we start to outline how it can be used to model object-oriented languages. In chapter 3 we study the termination of $\lambda\&$; we show first that $\lambda\&$ is not terminating and that is possible to define fixpoint operators for every type; then we describe some variants that we prove strongly normalizing and that are used in the chapter on the semantics. Chapter 4 is devoted to the study of three variation of $\lambda\&$: in the first we generalize some mechanisms of $\lambda\&$, in the second we add to it explicit coercions and in the last one we develop a proof theoretic study on the unification of the lambda-functions and overloaded functions. $\lambda\&$ is a kernel calculus and it does not possess the mechanisms to interpret an object-oriented language. Thus in chapter 5 we derive from $\lambda\&$ a language called λ_object, into which we are able to translate our toy object-oriented language; the chapter starts by the formal definition of the toy language and of its type system, followed by the definition of λ_object; finally the translation of the the toy language into λ_object and proof of the correctness of its type discipline are given. We end the first part of the thesis by tackling the problem of giving a mathematical semantics for $\lambda\&$ (chapter 6). We do not give the semantics to the whole calculus but we set the basis for it; indeed we only show how to give meaning to overloading with early binding in strongly normalizing systems. This study however opens interesting problems that pose new mathematical challenges as we describe in the conclusion of the chapter.

We begin the second part of the thesis by describing the motivation of passing to a second order formalism (chapter 7). We then deal with the problem of defining a second

order formalism by following two parallel directions.

In one direction we improve the existing second order formalisms that use subtyping, by defining a new subtyping discipline for universally quantified types (chapter 8); the resulting system enjoys many properties (foremost decidability) the existing formalisms do not. In the other direction we add overloading and late binding to existing second order formalisms that use subtyping (chapter 9) and we prove that the obtained calculus enjoys the properties of Church Rosser and subject reduction. Finally in chapter 10 we show the impact of the second order discipline on the modeling of the object-oriented languages, and we outline the modifications that one has to apport to the toy language of chapter 1 to take into account the new features of the second order. The thesis is ended by some appendixes where, among the other, we describe the implementation of an interpreter of λ_object.

This thesis is meant to have a basic conceptual unity: all the examples we use are defined in the first chapter and it is often the case that a chapter refers to the techniques and to the results of some previous chapters. Thus this thesis does not easily fit a non sequential reading. Especially for the chapters on the semantics and of the second part of the thesis, the reader not interested in the whole thesis is invited to refer to the corresponding papers; the same applies to the reader interested only in λ&. However all the single articles do not offer the global view of the model and of its underlying intuition that we hope to have given by this thesis. Yet, some topics (the whole chapter 4 and parts of the chapters 1, 2, 3, 5 and 10) are not covered by any publication.

The effort of writing a complete study had a nasty effect on the dimension of the work: we are conscious that a three hundred pages long thesis formed by interconnected chapters does not consent an easy reading. For this reason we propose to the unfortunate reader to follow some alternative paths of reading which are described by the automaton of Figure 0.1: initial states (double circled) is where you can begin your reading; then follow any ascending path of states (unlabeled transitions denote the full reading); final states are where you can stop. All the states are final ... it depends on your perseverance: have a nice reading.

[20] **In this way you'll never finish to read such a fat book!** ["Q65]

Background and notation

Term rewriting systems

- Given a (denumerable) set $\mathcal{F} = \cup_{n\geq 0}\mathcal{F}_n$ of function symbols and a (denumerable) set of variable symbols $\mathcal{X}$, the set of *terms* $\mathcal{T}(\mathcal{F},\mathcal{X})$ over $\mathcal{F}$ and $\mathcal{X}$ is the smallest set containing $\mathcal{X}$ such that $F(M_1,\ldots,M_n)$ is in $\mathcal{T}(\mathcal{F},\mathcal{X})$ whenever $F \in \mathcal{F}_n$ and $M_i \in \mathcal{T}(\mathcal{F},\mathcal{X})$ for $i = 1..n$.

- A binary relation R on $\mathcal{T}(\mathcal{F},\mathcal{X})$ is:
  reflexive $\stackrel{def}{\Longleftrightarrow}$ $(M, M) \in R$
  transitive $\stackrel{def}{\Longleftrightarrow}$ $(M, M') \in R, (M', M'') \in R \Rightarrow (M, M'') \in R$
  compatible $\stackrel{def}{\Longleftrightarrow}$ $(M, M') \in R \Rightarrow (F(\ldots M \ldots), F(\ldots M' \ldots)) \in R$
  for all $M, M', M'' \in \mathcal{T}(\mathcal{F},\mathcal{X})$ and $F \in \mathcal{F}$

- A *reduction relation* on $\mathcal{T}(\mathcal{F},\mathcal{X})$ is a binary relation on $\mathcal{T}(\mathcal{F},\mathcal{X})$ which is compatible, reflexive and transitive.

- A *notion of reduction* on $\mathcal{T}(\mathcal{F},\mathcal{X})$ is just a binary relation R on $\mathcal{T}(\mathcal{F},\mathcal{X})$.

- Let R be a notion of reduction on $\mathcal{T}(\mathcal{F},\mathcal{X})$. Then R induces the the following binary relations:

  1. The *compatible closure* of R, denoted by $\rhd_R$, and inductively defined as follows:
     $(M, N) \in R \Rightarrow M \rhd_R N$
     $M \rhd_R N \Rightarrow F(\ldots M \ldots) \rhd_R F(\ldots N \ldots)$.
  2. The *reflexive closure* of $\rhd_R$, denoted by $\rhd^{=}_R$, and defined as follows:
     $M \rhd_R N \Rightarrow M \rhd^{=}_R N$
     $M \rhd^{=}_R M$
  3. The *transitive closure* of $\rhd_R$, denoted by $\rhd^{+}_R$, and defined as follows:
     $M \rhd_R N \Rightarrow M \rhd^{+}_R N$
     $M_1 \rhd^{=}_R M_2$, $M_2 \rhd^{+}_R M_3 \Rightarrow M_1 \rhd^{+}_R M_3$
  4. The *reflexive and transitive closure* of $\rhd_R$, denoted by $\rhd^{*}_R$, and defined as the reflexive closure of $\rhd^{+}_R$
  5. The *equivalence relation* generated by $\rhd^{*}_R$, denoted by $=_R$ and inductively defined as follows:

$$M \ \rhd^*_R N \Rightarrow M =_R N$$
$$M =_R N \Rightarrow N =_R M$$

Note that $\rhd^*_R$ is a reduction on $\mathcal{T}(\mathcal{F}, \mathcal{X})$.

- An *R-redex* is a term M such that $(M, N) \in R$ for some N. In this case N is called an *R-contractum* of M.
- A term M is an $R-$*normal form* if none of its subterms is a R-redex.
- A term N is an $R-$*normal form of* M if it is an $R-$*normal form* and $M =_R N$.
- A reduction relation $\rhd^*_R$ is *weakly normalizing* if every term has an R-normal form
- A reduction relation $\rhd^*_R$ is *strongly normalizing* if there exists no infinite sequence of terms $M_1, M_2 \ldots$ such that $M_i \ \rhd M_{i+1}$
- A notion of reduction R satisfies the *diamond property* if for all M, M_1, M_2 $M \ \rhd_R M_1$ and $M \ \rhd_R M_2$ implies that there exists M_3 such that $M_1 \ \rhd_R M_3$ and $M \ \rhd_R M_3$.
- A notion of reduction R is *locally confluent* if for all M, M_1, M_2 $M \rhd_R M_1$ and $M \rhd_R M_2$ implies that there exists M_3 such that $M_1 \ \rhd^*_R M_3$ and $M \ \rhd^*_R M_3$.
- A notion of reduction R is *confluent* if for all M, M_1, M_2 $M \ \rhd^*_R M_1$ and $M \ \rhd^*_R M_2$ implies that there exists M_3 such that $M_1 \ \rhd^*_R M_3$ and $M \ \rhd^*_R M_3$ (i.e. $\rhd^*_R$ satisfies the diamond property).
- A notion of reduction R is *Church-Rosser* if for all M, N $M =_R N$ implies that there exists P such that $M \ \rhd^*_R P$ and $N \ \rhd^*_R P$.

Theorem 1 *A notion of reduction R is* Church-Rosser *if and only if it is confluent.*

Logic

- Given a language $\mathfrak{L}$ and a notion of derivability $\vdash$ on $\mathfrak{L}$, a *theory* $\mathcal{T}$ is a collection of sentences in $\mathfrak{L}$, with the property $\mathcal{T} \vdash \varphi \Rightarrow \varphi \in \mathcal{T}$ (a theory is *closed under derivability*)
- A set Σ such that $\mathcal{T} = \{\varphi | \Sigma \vdash \varphi\}$ is called an *axiom set* of the theory $\mathcal{T}$. The elements of Σ are called *axioms*.
- Let $\mathcal{T}$ and $\mathcal{T}'$ be theories in the languages $\mathfrak{L}$ and $\mathfrak{L}'$.
  1. $\mathcal{T}'$ is an *extension* of $\mathcal{T}$ if $\mathcal{T} \subseteq \mathcal{T}'$
  2. $\mathcal{T}'$ is a *conservative extension* of $\mathcal{T}$ if $\mathcal{T}' \cap \mathfrak{L} = \mathcal{T}$ (i.e. all theorems of $\mathcal{T}'$ in the language $\mathfrak{L}$ are already theorems of $\mathcal{T}$).

Part I

Simple typing

Chapter 1

Object-oriented programming

1.1 A kernel functional object-oriented language

In this section we briefly discuss (a certain kind of) object-oriented programming by gradually introducing a toy functional object-oriented language. For the functional core of this language we use the syntax of an explicitly typed version of ML. The syntax of the object-oriented components is inspired by Objective C (see [PW92] and [NeX91]) since it fits well our approach, and is general enough to represent a wide class of object-oriented languages. This does not aim to be a comprehensive presentation of object-oriented features. Far from that, it tends to present some kernel features of object-oriented programming from our particular perspective, which is the one we acquired in defining and developing the λ&-calculus, the basic calculus of this thesis. As we said in the preface, this identifies a certain class of object-oriented languages, whose best representative is perhaps CLOS, and that has been disregarded by the current type theoretic research. Therefore some features peculiar to object-oriented languages of a different class are voluntarily omitted. We will stress the differences between them in section 2.6.1 and in chapter 11. In this chapter we just give an informal presentation of the language. The formal presentation is given in chapter 5

1.1.1 Objects

Object-oriented programs are built around *objects*. An object is a programming unit that associates data with the operations that can use or affect these data. These operations are called *methods*; the data they affect are the *instance variables* of the object. In short objects are programming units formed by a data structure and a group of procedures that affect it.

Example 1.1.1 We want to define an object for two-dimensional points (`2DPoint`). A `2DPoint` object represents a point of the cartesian plane: it contains instance variables that define the position of the object; it can apply methods that return the *norm* of the point, that *erase* its x-coordinate or that *move* the position of the point. Methods may require additional parameters, as in the case of the method that moves the point, which must be told how to move it. □

The instance variables of an object are private to the object itself; they can be accessed only through the methods of the object. Moreover an object sees only the methods that were

designed for it; it cannot mistakenly perform methods that were designed for other objects. Thus object-oriented programming is characterized by a structured programming style where each single subproblem is solved by an object.

1.1.2 Messages

The only thing that an object is able to do is to respond to *messages*. A *message* is simply the name of a method that was designed for that object. In our syntax *message-expressions* are enclosed in square brackets:

[*receiver message*]

The *receiver* is an object (or more generally an expression returning an object); when it receives a message, the run-time system selects among the methods defined for that object the one whose name corresponds to the passed message; the existence of such a method is statically checked (i.e. it is verified at compile time) by a type checking algorithm.[1] .

Example 1.1.2 [continued] Suppose we have a `2DPoint` object called `myPoint`. We want to tell it to execute the method, named `norm`, that returns the norm of the object. This can be done by sending the message `norm` to `myPoint`:

```
[myPoint norm]
```

This expression can be used to define a function `isOrigin` that checks whether a given `2DPoint` coincides with the origin of the cartesian plane or not (it just verifies whether the norm of the point is equal to zero); we can then apply this function to `myPoint`:

```
let isOrigin = fn(p:2DPoint) => ([p norm] == 0)
in isOrigin(myPoint)
```

As we said before some methods may require additional parameters; in the case of the method that moves a `2DPoint`, it requires the *dx* and *dy* of the displacement:

```
[myPoint move](3,5)
```

From the viewpoint of types, *message-expressions* can return either a basic value (such as in the case of `norm`) or a function (as in this case): in the example above the expression `[myPoint move]` returns a function whose type is `Real` $\times$ `Real` $\rightarrow$ `2DPoint`: it accepts a pair of reals and returns the object in a different position (i.e. it modifies the *instance variables*). □

1.1.3 Methods and functions

At first sight methods seem to play the role of functions and message passing the one of functional call. Though the fact that a method *belongs* to a specific object (more precisely to a specific *class* of objects) implies that *message passing* is a mechanism different from the usual function call (i.e. the β-reduction). We stress in this section the two main characteristics that distinguish methods from functions.

Overloading

Two objects can respond differently to the same message. For instance suppose we have a `Chessman` object. The effect of sending to it the message `move` (supposing that a method

[1] The leitmotiv of this thesis is to give the theoretical basis for the definition of such an algorithm.

with that name has been defined for `Chessman`) would be different from the one of sending `move` to `myPoint`, in the sense that a different code would be executed. Though the same message behaves uniformly on objects of the same kind: the message `move` has the same effect on `myPoint` as on every other `2DPoint` object[2]. Thus a message may behave in a different way on values of different types. Note that this kind of *polymorphism* is quite different from the one that characterizes, say, the function `head` in ML, which works on lists of any type: in the case of `head` the behavior of the function on values of different types is essentially the same, in the sense that always the same code is executed. On the contrary in the case of methods, to different types of the input may correspond completely different codes (as for the case of `2DPoint` and `Chessman`). This behavior is known as *overloading* since one overloads the same operator (in this case `move`) by different operations; the actual operation depends on the type of the operands. Thus *messages are identifiers of overloaded functions* and in message passing the *receiver* is the argument of an overloaded function, i.e. the one on whose type the selection of the code to be executed is based. Each method constitutes a branch of the overloaded function denoted by the message it is associated to.

Late Binding

The second crucial distinction between function calls and message passing is that a function and its arguments are bound together in the compiled code while a method and the receiving object are united only at run-time, i.e. during the computation. This tool, called *late binding*, is one of the most powerful characteristics of object-oriented programming and, in our case, has to do with the combination between overloading and subtyping: indeed we define on types a partial order which concerns the utilization of values: a value of a certain type can be used wherever a value of a supertype is required. In this case the exact type of the receiver cannot be decided at compile time since it may change (notably decrease) during computation.

For example consider again the function `isOrigin`. We can apply it to any object whose type is `2DPoint` but also to any object whose type is a *subtype* of `2DPoint`; for example a cartesian point with some additional features as, say, a color. Thus when compiling `[p norm]` we cannot link at compile time the message `norm` to the method defined for `2DPoint` objects: even if the formal parameter `p` has type `2DPoint` we may discover, after having performed the substitution of the application, that `p` actually refers to a, say, `2DColorPoint` object and thus the method for these objects must be chosen. In other words if the compile time type of `p` is used for the branch selection (early binding) the function `isOrigin` is always executed by using the `norm` code for `2DPoint`. Using late binding, each time the whole function is applied, the code for `norm` is chosen only when the `p` parameter has been bound and evaluated, on the basis of the run time type of `p`, i.e. according to whether `p` is bound to a `2DPoint` or a `2DColorPoint`.

Therefore in our model overloading with late binding is the basic mechanism.

Excursus (late vs. dynamic binding) *Overloaded operators can be associated with a specific operation using either "early binding" or "late binding". As we already*

[2]This is not always true in object-oriented programming, and it is one of the main features the distinguish our approach from the record-based one. See section 2.6.1

said in the introduction of the thesis, this distinction applies to languages where the type which is associated at compile time with an expression can be different (less informative) from the type of the corresponding value, at run time. The example above with `2DPoint` *and* `2DColorPoint` *should be clarifying enough. Note though that what here we call* late *binding, in object-oriented languages is usually referred as* dynamic *binding (see for example [Mey88, NeX91]). Late and dynamic binding (or "dynamic scoping") are yet two distinct notions. Early vs. late binding has to do with overloading resolution, while static vs. dynamic binding means that a name is connected to its meaning using a static or a dynamic scope. However this mismatch is only apparent, and it is due to the change of perspective between our approach and the one of the languages cited above: in [Mey88] and [NeX91], for example, the suggested understanding is that a message identifies a method, and the method (i.e. the meaning of the message) is* dynamically *connected to the message; in our approach a message identifies an overloaded function (thus a set of methods) and it will always identify this function (thus it is statically bounded) but the selection of the branch is performed by late binding.*

The situation is actually more complex. As a matter of fact, messages obey an intermediate scoping rule: they have a "dynamically extensible" meaning. If the type `2DPoint` *is defined with the method* `norm`*, then the meaning of the* `norm` *method is fixed for any object of type* `2DPoint`*, like what happens with static binding. However, if later a new type* `3DPoint` *is added to the system, the set of possible meanings for the* `norm` *message is dynamically extended by the method for* `3DPoint` *and the function* `isOrigin` *in the previous example will use the correct method for* `3DPoint`*, even if* `3DPoint` *did not exist when the function was defined. This combination of late binding and dynamic extensibility is one of the keys of the high reusability of object-oriented languages. Essentially, these languages allow one to extend an application by simply* adding *a subclass of an existing class, while in traditional languages one usually also needs* modifying *the old code, which is a costlier operation.*

The use of overloading with late-binding automatically introduces a further distinction between message passing and ordinary functions. As a matter of fact, overloading with late-binding requires a restriction in the evaluation technique of arguments: while ordinary function application can be dealt with by either call-by-value or call-by-name, overloaded application with late binding can be evaluated only when the run time type of the argument is known, i.e. when the argument is fully evaluated (closed and in normal form). In view of our analogy "messages as overloaded functions" this corresponds to say that message passing (i.e. overloaded application) acts by call-by-value or, more generally, only closed and normal terms respond to messages.

1.1.4 Classes

An object-oriented program consists of a bunch of objects that interact by message passing. A program simulating a chess game would be probably built around thirty-two `Chessman`

objects and two `Player` objects. Of course you must not repeat the definition of the methods for every object; it is possible to describe all the objects of a certain type by just one definition; this description is given by a *class*. Thus a class fixes the prototype for the objects of the same type: it declares the instance variables (with their initial values) that form the data structure of every object of that class, and defines the methods that all the objects of the class can use. The name of the class is used for the type of its objects. The name of a class will be considered as an "atomic type" of our type system. Besides the name, the instance variables and the methods, a class also defines an *interface*. The interface is the description of the type of the methods[3]. Class definition is the main task in object-oriented programming.

In Objective C the declaration of the instance variables is done by a record type whose labels are the instance variables; for our prototypical language we have chosen to give to instance variables also an initial value, thus we add to the record type of the class definitions in Objective C also a record value defining the initial values. A record value is an expression of the form `{`$x_1{=}exp_1;\ldots;x_n{=}exp_n$`}`; if T is a type and e an expression then $e : T$ means "e has type T". A record type is denoted by $\langle\langle x_1 : T_1; \ldots; x_n : T_n \rangle\rangle$. We use `{`$x_1 : T_1{=}exp_1; \ldots;x_n : T_n{=}exp_n$`}` as an abbreviation for `{`$x_1{=}exp_1;\ldots;x_n{=}exp_n$`}`: $\langle\langle x_1 : T_1; \ldots; x_n : T_n \rangle\rangle$. The value of an instance variable `x` is referred, in the body of a method, by `self.x`. The instance variables of an object are "modified" by an operation `update` which returns a new object of the same type. Interfaces are enclosed in `[[...]]`.

Example 1.1.3 The class that describes the `2DPoint` objects is defined as follows:

```
class 2DPoint
  {
    x:Int = 0;
    y:Int = 0
  }
    norm = sqrt(self.x^2 + self.y^2);
    erase = (update{x = 0});
    move = fn(dx:Int,dy:Int) => (update{x=self.x+dx; y=self.y+dy})
  [[
    norm: Real;
    erase: 2DPoint;
    move: (Int x Int) -> 2DPoint
  ]]
```

□

After that a new class has been defined, one can use the command `new` to create objects that match the characteristics of the class. Such objects are called *instances* of the class. For example the expression

```
new(2DPoint)
```

returns a `2DPoint` object whose internal state is the one defined in the class (i.e. $x = 0$ and $y = 0$). Since the name of a class is used for the type of its instances then `new(2DPoint):2DPoint`.

[3]The interface should typically be written in a separate file if we are interested in modular programming allowing separate compilation.

A *program* in our prototypical language is a sequence of declarations of classes followed by an expression (the so-called *body* of the program) where objects of these classes are created and interact by exchanging messages.

1.1.5 Inheritance

In this section we describe two of the most delicate and powerful mechanisms of object-oriented programming: inheritance and subtyping.

It is often the case that, in an expanding environment, we need to define some new objects which are a refinement or a specialization of existing ones. Consider again the example of chess: all the `Chessman` objects possess the same methods to return the position of the piece and to capture a chessman; but some parts of each object must be specialized according to the particular chessman the object represents: we have to implement the method which moves a chessman in a different way for each chessman. Also instance variables may need to be specialized: *castling* is allowed only if King and Rook have not moved from their original positions, therefore these two chessmen need a further instance variable which records if the object is still unmoved. The naive solution to all these problems of specialization would be to define a different class for each different chessman; but in this way methods that are the same for all chessmen would be duplicated in every class, with the usual problems of consistency and redundancy of duplicated code. The alternative is to use the mechanism of inheritance which permits to reuse the code written for a class in the definition of an other. For example it is possible to start by defining a class `Chessman` where we describe the instance variables and the methods common to all chessmen. Then we specialize this class defining a *subclass* for every kind of chessman: the definition of a subclass specifies the name of the new class (i.e. `King`, `Queen`, `Bishop`, etc ...), of its direct ancestor (i.e. `Chessman`), the declaration of all instance variables (which must contain at least all instance variables of the superclass and with the same type)[4] and the definitions only of those methods that are specific to the class. The methods that are defined in the *superclass* (i.e. `Chessman`) are visible to the objects of all subclasses. And we say that a class *inherits* the methods of its superclasses. In other terms, an object has access not only to the methods defined for its class but also to the methods for its superclass, and for its superclass's superclass, all the way back to the root of the hierarchy. However there is an exception, quite useful, to this mechanism: when a class is defined as subclass of another, it can define a new method with the same name of another already defined in the hierarchy; in this case we say that this method *overrides* the old one: thanks to late binding, the objects of this class and of their subclasses will always use this new definition instead of the old one. Note though that the old definition is not erased, as it will be still used by the objects belonging to the classes up in the hierarchy. Thus *inheritance* is the mechanism that permits us to define a new class as a refinement of an old one: it establishes when the objects of a given class can use the operations originally defined for the objects of a different class. The refinement consists in the addition of new instance variables

[4]In running object-oriented languages it is not required to repeat all the instance variables, but it suffices to declare the instance variables that must be added in the specialization. It is possible to do so also in our language, but this would much complicate the definitions for the type-checker; thus we prefer to ease the presentation of the type-checker since this is the main concern of our work. We leave the motivated reader to do the easy (but twisted) modifications to obtain the wanted system.

or new methods or in the redefinition of existing methods.

Subtyping instead is a mechanism that permits us to use an object of some class where an object of a different class is required. For example suppose we have a function that works on the objects of class `Chessman`. Since the methods defined for `Chessman` are also defined for, say, `Bishop`, then, intuitively, this function should be able to work also with objects of the latter class. In general it is very likely that a function defined for `Chessman` will be fed by objects belonging to a subclass of `Chessman`. Then we have to prevent the type checker from signaling this situation as an error. To this end we define a partial order on types called the *subtyping relation*. Intuitively one type S is smaller than another T if every value of S can be safely used wherever a value of T is needed. Thus in the example above it suffices to declare that `Bishop` is a subtype of `Chessman`.

Recapitulating, inheritance is the mechanism that allows one to reuse code written for other classes: it mainly concerns the definition of the objects. Subtyping is the mechanism that allows one to use one object instead of one of another class: it mainly concerns the computation of the objects. We thus have two hierarchies, one induced by inheritance the other corresponding to the subtyping relation. The terms *subclass* and *superclass* are used to refer to the former, and *subtype* and *supertype* to refer to the latter.

In object-oriented languages inheritance is defined only on classes, i.e. on atomic types (see section 11.2.1 on how to overcome this limitation); on these types the subtyping relation implies the inheritance relation, for if the objects of a certain class can substitute those of another class then they can also use all the operations defined for the objects of that class. In other terms if a class is a *subtype* of another then it is also a *subclass* of it. But while the inheritance relation is defined only for classes, subtyping can be extended also to higher types (see sections 1.2.2 and 5.1.2). Thus these hierarchies are completely distinct, even if they are so tightly related on classes.

Of course not every class can be a subclass of another, since every method is defined for object possessing certain characteristics. Thus refinement must satisfy some conditions.

In this thesis we focus only on the form of inheritance that one has by subtyping. We will define type systems that will not check the conditions for full inheritance in class definitions; they will check only the more restrictive conditions for subtyping. We will always have inheritance associated to subtyping. Thus also in the toy language that we are presenting it is not possible to have "pure" inheritance (i.e. code reuse without the substitutivity given by subtyping). In view of this, in the rest of the thesis we will confound the terms "inheritance" and "subtyping" for atomic types, since under the assumption above they coincide.

This does not imply that we consider pure inheritance as uninteresting; indeed it has an undeniable practical utility. Though, subtyping appears as a more fundamental issue: once it is understood, it is then not too hard to tackle the modeling of pure inheritance. We informally do it in the conclusion of this thesis, in section 11.2.1, where we also define two new forms of inheritance such as partial inheritance and inheritance for higher types.

Example 1.1.4 We can refine the class `2DPoint` of Example 1.1.3 by adding an instance variable for the color, a new method `isWhite` and by overriding the method `move`. The class we obtain, which we call `2DColorPoint`, is defined in the following way:

```
class 2DColorPoint is 2DPoint
  {
    x:Int = 0;
    y:Int = 0;
    c:String = "black"
  }
    isWhite = (self.c == "white")
    move = fn(dx:Int,dy:Int) =>
                      (update{x=self.x+dx; y=self.y+dy; c="white"})
  [[
    isWhite: Bool
    move: (Int x Int) -> 2DColorPoint
  ]]
```

The methods `norm` and `erase` are *inherited* from `2DPoint`; thus for example the expression `[new(2DColorPoint) norm]` returns 0. The method `move` is redefined (overridden) so that if a colored point is moved, its color is set to white. As we said above, inheritance is always associated to subtyping; thus the keyword `is` in the definition above says that `2DColorPoint` inherits from `2DPoint` *and* that it is a subtype of it (denoted `2DColorPoint` $\leq$ `2DPoint`). This implies that one can use a `2DColorPoint` object wherever a `2DPoint` is required. □

We said that to substitute values of some type by those of another some requirements must be satisfied. If the type at issue is a class then the following condition on instance variables must be fulfilled:

> The set of the instance variables of a class must contain those of all its superclasses. Moreover these variables must appear always with the same type

Besides, the refinement must also satisfy the condition of *covariance*:

> A method that overrides an old one given in a supertype must specialize it, in the sense that the type returned by the overriding method must be a subtype of the type returned by the overridden one.

We will say more on these two conditions in the section devoted to type checking.

1.1.6 Multiple inheritance

It is sometimes very useful to define a class as the refinement of two or more classes. In fact, apart from implementation matters, there is no reason for a class to have just one supertype. Thus in the class definitions one may specify more than one ancestor. Obviously the set of instance variables of the new class must contain the union of the instance variables of all supertypes, and the new class inherits all the methods of its superclasses.

Example 1.1.5 The `2DColorPoint` could be also defined by multiple refinement in the following way:

```
class Color
  {
    c:String = "black"
  }
    isWhite = (self.c == "white")
  [[
    isWhite: Bool
  ]]


class 2DColorPoint is 2DPoint, Color
  {
    x:Int = 0;
    y:Int = 0;
    c:String = "black"
  }
    move = fn(dx:Int,dy:Int) =>
                     (update{x=self.x+dx; y=self.y+dy; c="white"})
  [[
    move: (Int x Int) -> 2DColorPoint
  ]]
```

The class `2DColorPoint` inherits the methods `norm, erase` from `2DPoint`, `isWhite` from `Color` and overrides the definition of `move` given in `2DPoint`. □

Since a class may have many incomparable supertypes, we no longer have a hierarchy but rather a dag (this is sometimes called a *heterarchy*: see [App92]).

Suppose we define a new class C by multiple inheritance from two unrelated classes A and B, and that both A and B have defined (or inherited) a method for a message m. Then comes the problem to decide which method to execute when m is sent to an instance of the new class C. In object-oriented languages two different solutions have been adopted: the first consists in establishing a search order on the supertypes (as it is done for example in CLOS [DG87] where this order is called *class precedence list*). In the example above we can choose the order in which supertypes appear in the definition so that a method not defined in the `2DColorPoint` class would be first searched in the hierarchy of `2DPoint` and then in the one of `Color`; thus if `2DPoint` and `Color` had defined a method for the same message `mesg` then the one defined in `2DPoint` would be executed. The other solution is to impose that methods common to more then one ancestor must be explicitly redefined (as it is required in Eiffel [Mey88]). In this case the redefinition of the method for `mesg` would have to appear in the definition of `2DColorPoint`. In our system we have chosen the second solution which is less syntax dependent and mathematically cleaner. Thus we have condition of *multiple inheritance*:

> When a class is defined by multiple refinement, the methods that are in common to more than one unrelated supertype must be explicitly redefined

Note however that this applies only to *unrelated* supertypes: if we have defined a class as a refinement of two classes A and B, and A is a subtype of B, then all the methods of B are in common with A but they need not to be redefined since the system will always choose the more recently defined, i.e. those of A (even if this probably is a programming error).

1.1.7 Extending classes

Refinement is not the only way to specialize classes. It would be very annoying if every time we have to add a method to a class we were obliged to define a new class: the existing objects of the old class could not use the new method. The same is true also in the case that a method of a class must be redefined: *overriding* would not suffice. For this reason some object-oriented languages offer the capability to add new methods to existing classes or to redefine the old ones (this capability is very important in persistent systems). In our prototypical language this can be done by the following expression:

`extend` *classname*
  methodDefinitions
  interface
`in` *expr*

the newly defined methods are available in the expression *expr*. Remark that by this construction we do not define a new class but only new methods; in other terms we do not modify the existing types but only (the environment of) the expressions. This is possible in our system since the type of an object is not bound to the procedures that can work on it (and for this reason it differs from abstract data types and the "objects as records" approach). Finally, the extension of a class affects all its subtypes, in the sense that when you extend a class with a method then that method is available to the objects of every subtype of that class. Besides the advantages cited above this mechanism can benefit also the development process in some ways:

1. It simplifies the management of large classes when they are defined by more than one developer.
2. It enables to configure a class differently for different applications and in the same applications for different expressions as well.
3. It helps in the tuning up and debugging of existing programs: sometimes it is necessary to modify existing methods slightly in order to obtain the required performances from the new ones.

Addition and redefinition of methods are implemented by some object-oriented languages (e.g. Objective-C [PW92], CLOS [DG87] and Dylan [App92]). Anyway it must be clear that these features constitute a trade-off between encapsulation and flexibility, and thus should be coupled with some further mechanism of protection. For example Dylan has a function `freeze-methods` which prevents certain methods associated to a message from being replaced or removed.

1.1.8 Super, self and the use of coercions

Very often a method needs to refer to the object that performs it. Suppose for example we wish to extend the definition of `2DPoint` by a method `reposition` which must send the message `move` to the object that actually called it. Then we have an expression of the form

```
extend 2DPoint
  reposition = ... [ ??? move ] ...
  [[ ... ]]
in ...
```

But we do not know which receiver is to use in the expression above. Note that in this case it is not the same to substitute the message expression by the definition of the method `move` as defined in `2DPoint`: `reposition` is inherited by `2DColorPoint`; thus if the method `reposition` is performed by an instance of `2DColorPoint` we want the definition of `move` in `2DColorPoint` to be used; this is automatically obtained by late binding, once we know what to put in the place of the question marks; but it would not work if we directly used the definition of the method.

The solution is to put in the place of the question marks the reserved keyword `self`. This keyword refers to the receiver of the message that called the method. This object is often referred as the *current object* and its class as the *current class*. Remark that the current class is not always the class where the method has been defined, but it may also be a subclass of its (when the method is an inherited method). Thus in the definition of `reposition` we use `[ self move ]`; if we send the message `reposition` to `myPoint` then the definition of `move` in `2DPoint` is used. If the receiver is instead a `2DColorPoint` then the overriding definition for `move` is called. To put it in other terms, recall that the message `move` is an identifier of overloaded function and the receiver is the argument of this function; thus in the definition of a method (a branch) we use the keyword `self` to denote its (hidden) argument, i.e. the one the selection is based on. This explains why to access to an instance variable x we use the notation `self.`x: the hidden argument of the method is thought to be the record value of the instance variables.

Anyway, it may be the case that one wants always to use the definition of `move` given in `2DPoint`. Again the substitution of the code of the method is not a good solution since we know that by `extend` this code may be changed or updated. In object-oriented languages there is a way to refer to the overridden code of a method. This is usually done by a construct called `super`: in object-oriented programming languages when one sends a message to `super`, the effect is the same as sending it to `self` but with the difference that the *selection is performed as if the receiver were* an instance of a super-class. Here we generalize this usual meaning of super in two ways: the selection does not assume that the receiver is `self`, but takes as receiver the parameter of `super`; and `super` does not necessary appears in the receiver position, but it is a first-class value (i.e. it can appear in any context its type allows to). Finally, since we use multiple inheritance without class precedence lists, we are obliged to specify in the expression the supertype from which to start the search of the method[5]. Thus the general syntax of super is `super[`A`](`exp`)`. When a message is sent to this expression

[5]For instance, this is what is done in Fibonacci [ABGO93], developed at the University of Pisa

then *exp* is considered the receiver but the search of the method is started from the class A (which then must be a supertype of the class of *exp*).

In the previous example, to specify that the method selected for `move` must be the one which would be selected for a receiver of class `2DPoint` one writes:

```
extend 2DPoint
  reposition = ... [ (super[2DPoint](self)) move ] ...
  [[ ... ]]
in ...
```

Very close to the use of `super` is the use of coercions. By a coercion one changes the class of an object by a supertype. The difference between them is that `super` changes the class of an object only in the first message passing, while coerce changes it for the whole life of the object. The syntax is the same as that of `super`: thus we write `coerce[`A`](`*exp*`)` to change to A the type of the expression *exp*. A short example can clarify the behavior of `super` and `coerce`: suppose to have these three classes

- a class A in which we define a method `m1`
- a class B subtype of A in which we define a method `m2` whose body contains the expression `[self m1]`
- a class C subtype of B in which we override both `m1` and `m2`.

Let `M` be an object of type C. Consider now these two expressions `[ super[B](M) m2 ]` and `[ coerce[B](M) m2 ]`. In both cases the method selected is the one defined in B. But in the body of `m2` the meaning of `self` is , in the former case, `M` while in the latter it is `coerce[B](M)`: therefore the method used for `[self m1]` will be the one defined in C when using `super` and the one in A with `coerce`. To sum up, `coerce` changes the class of its argument and `super` changes the rule of selection of the method in message passing (it is a coercion that is used only once and then disappears) [6].

1.1.9 Multiple dispatch

We have seen that when a message is passed the method executed is chosen according to the class of the receiver. Sometimes it is useful to base the choice of the method also on the class of the parameters of the method and not only on the receiver. For example recall the definition of `2DPoint` and `2DColorPoint` given in section 1.1. The objects of the former class respond to the messages `move` and `norm` while the objects of the latter accept also the message `isWhite`. Suppose that we want to extend the class `2DPoint` by a method `compare` which takes a point as parameter and if this point is a `2DPoint` then it checks the equality of the norms while if the point is a `2DColorPoint` it checks whether the passed point is white or not. The choice of a method based on the classes of possible parameters is called *multiple dispatch* and the method at issue is usually referred as a *multi-method* (see e.g. [Kee89]). In our toy language this can be obtained by the following expression:

[6] It is interesting that with our generalization of `super` it is possible to predetermine the life of a coercion: for example `super[A](super[A](M))` coerces `M` to A only for the first two message passing.

```
extend 2DPoint
  compare = & fn(p:2DPoint) => [self norm] == [p norm]
            & fn(p:2DColorPoint) => [p isWhite];
  [[ compare:#{2DPoint -> Bool; 2DColorPoint ->Bool} ]]
in ...
```

Each possible choiche is introduced by the symbol `&`. Note that the type of a multi-method appears in the interface as the set of the types of the possible choices (the reason of `#` is explained in the next section).

The number of parameters on which the dispatch is performed may be different in every branch. For this reason, when a message denoting a multi-method is sent, we must single out those parameters the dispatching is performed on. This is done by including them *inside* the brackets of the message-passing, after the message. Thus the general syntax of message passing gets: `[`*receiver message parameter*`,` ... `,` *parameter*`]`. For example `[myPoint compare myPoint]` will dispatch on the first branch of the method. For example, consider a class C with the following interface: `[[ msg:#{Int -> (Int -> Bool), Int x Int -> Bool}]]`; then `msg` is a message that when it is sent to an integer it returns a function from integers to booleans, when it is sent to a pair of integers it returns a boolean. Thus if M is of class C then the expression `[` M `msg 3 ] 4` selects the first branch while `[` M `msg 3,4]` selects the second one. We have to impose a restriction in our system: `super` cannot work with multiple dispatching. When super selects a multi-method, it works as `coerce`[7]

1.1.10 Messages as first-class values: adding overloading

We said from the very beginning that messages are identifiers of overloaded functions. But up to now we have no tool to work directly with overloaded functions: overloaded functions can be defined only through class definitions and cannot be passed as a parameter to a function. Thus the next step is to introduce explicit definitions for overloaded functions and to render them (and thus messages) first-class values. The gain is evident: for example we can have functions accepting or calculating messages (indeed overloaded functions) and to write message passings of the form `[`*receiver* $f(x)$`]`.

We already possess all the syntax we need: We already have the syntax for overloaded application which is `[`$exp_0\ exp\ exp_1$`, ...,` exp_n`]` where exp is the overloaded function and $exp_0, exp_1, \ldots, exp_n$ are the arguments. We already have the syntax for the definition of an overloaded function: note that a multi-method is an overloaded function (quite special indeed, as we will see in the section for type checking), since the branch is selected according to the type(s) of the argument(s). Therefore we build an overloaded function by concatenating the various branches by `&`: if f is a normal function then (`&`f) is the overloaded function with just one branch, i.e. f itself; and if g is an overloaded function then (g`&`f) is the overloaded function g where we have added the branch f. However we want to impose the same restrictions as in the multi-methods, i.e. we impose that a branch is a term of the form `fn(`x_1`:`$T_1, \ldots, x_n$`:`T_n`)=>`v that is the branch is already in normal form (this is suggested by

[7]This restriction is due to the definition of the language we translate this toy language in. Indeed, we could define the target language so that super worked also with multiple dispatch, but it would greatly complicate its operational semantics

common sense); and we require that the T_i's are atomic types (this is strongly recommended by the implementation: if we select a branch only on atomic types then the selection is simply implemented by the check of a tag; if higher types were involved then type checking should be executed at each overloaded application: see section 5.2).

The type of an overloaded function is the set of the types of its branches.

Note that the use of the same syntax for message passing and overloaded application, while providing a conceptual uniformity, has a major drawback: when the overloaded function bases the selection on more than one argument then the arguments have to be "split" around the overloaded function as in the case of multiple dispatching. And while before it had a sense to isolate a particular argument, since it was the receiver in whose class the multi-method had been defined (or inherited), in this case it is misleading. Note however that in case of binary infix overloaded operators this turns out to be very interesting: for example an overloaded plus working both on integers and reals can be defined in the following way:

```
let plus = (& (fn(x:Real,y:Real) => x real_plus y)
            & (fn(x:Int,y:Int) => x int_plus y))
```

which has type $\{Real \times Real \to Real, Int \times Int \to Int\}$. Thus the sum of two numbers using `plus` is written `[ x plus y ]`. But, apart from these special cases, it remains a problem and it may suggest us to consider a different syntax for message passing where the message is the left argument, as done in CLOS and Dylan.

Finally note that the use `of #` in the interfaces is necessary to distinguish multi-methods from methods returning an overloaded function. Use the same interface as in the section before but without the "`#`" i.e. `[[ msg:{Int -> (Int -> Bool), Int x Int -> Bool}]];` then `msg` is now a ordinary method returning an overloaded function; thus now the expression `[3 [ M msg ] ] 4` selects the first branch while `[ (3,4)[ M msg ] ]` selects the second one (once more the notation is misleading).

1.2 Type checking

In this section we informally describe the type system of our toy language. We give the general rules, with their intuitive explanation.

1.2.1 The types

The types that can be found in a program of our toy-language are the following:

- Built-in atomic types (as `Int, Bool` etc) and class-names which are user-defined atomic types.
- Product types $(T_1 \times T_2)$, for pairs
- Arrow types $T_1 \to T_2$, for ordinary functions
- Sets of arrow types $\{A_1 \to T_1, \ldots, A_n \to T_n\}$ called *overloaded types* and used for overloaded functions. We call $A_1 \ldots A_n$ and $T_1 \ldots T_n$ *input* and *output* types respectively. In an overloaded type there cannot be two different arrow types with the same input type (*input type uniqueness*).

1.2.2 Intuitive typing rules

We describe here only the rules for the object-oriented part of the language, since the typing of the functional part is quite standard.

Rules for Terms

1. The type of an object is (the name of) its class.
2. The type of a coercion is the class specified in it, provided that it is a supertype of the type of the argument.
3. The type of a super is the class specified in it, provided that it is a subtype of the type of the argument
4. The type of `self` is the name of the class whose definition `self` appears in.
5. The type of an overloaded function is the set of the types of its branches
6. The type of an overloaded application is the output type of the branch whose input type "best approximates" the type of the argument. This branch is selected among all the branches whose input type is a supertype of the type of the argument and it is the one with the least input type.

These are all the typing rules we need to type the object-oriented part of the toy language, since we said that messages are nothing but overloaded functions and message passing reduces to overloading application. However to fully understand message passing we must specify which overloaded function a message denotes. Consider again message passing: we said that the receiver is the argument of the overloaded function the selection is based on. Suppose that you are defining a class C and remember that inside the body of a method, the receiver is denoted by `self`. Then there are two cases:

1. The method `msg=`*exp* is not a multi-method and returns (according to the interface) the type T. This corresponds to add to the overloaded function denoted by `msg` the branch `fn(self:`C`).`*exp* whose type is $C \to T$.
2. We have the multi-method

   $$\begin{aligned}\texttt{msg = \& fn(}&\texttt{x}_1\texttt{:A}_1\texttt{,...,x}_i\texttt{:A}_i\texttt{) => } expr_1\\ &\vdots\\ \texttt{\& fn(}&\texttt{y}_1\texttt{:B}_1\texttt{,...,y}_j\texttt{:B}_j\texttt{) => } expr_n\end{aligned}$$

   which returns the type $\#\{(A_1\times\ldots\times A_i) \to T_1,\ldots,(B_1\times\ldots\times B_j) \to T_n\}$. This corresponds to add to the overloaded function denoted by `msg` the n branches `fn(self:`C`,` $\texttt{x}_1{:}\,A_1,\ldots,\texttt{x}_i{:}\,A_i$`) =>` $expr_1$ ... `fn(self:`C`,` $\texttt{y}_1{:}\,B_1$, ..., $\texttt{y}_j{:}\,B_i$`) =>` $expr_n$ of types $(C \times A_1 \times \ldots \times A_i) \to T_1,\ldots,(C \times B_1 \times \ldots \times B_j) \to T_n$

In conclusion a message denotes an overloaded function that possess one branch for every class in which a method has been defined for it and one for every branch of a multi-method associated to it. Message passing is typed as the overloaded application. The selection of the branch corresponds to the search of the least supertype of the class of the receiver (a class is a supertype of itself) in which a method has been defined for the message (this is the usual method look-up mechanism of Smalltalk [GR83]).

Rules for Subtyping
The rule to use the subtyping relation is very simple:

> It is safe to use a term of a certain type wherever one of greater type is expected

Thus a type safe computation must always preserve or decrease the type of a term. For example suppose to have an application, say, $f(e)$ and that the system performs a call-by-value reducing e to e'; if the type of e' were not smaller than the type of e then $f(e')$ might be not type safe.

The subtyping relation is predefined by the system on the built-in atomic types; the programmer defines it on the atomic types (i.e. the classes) he introduces, by means of the construct `is`. Therefore the system and the programmer completely define the subtyping relation on atomic types, but what about higher types? Once defined on atomic types this relation is automatically extended to higher types according to some rules which fit very well the intuition under the types. These rules are obtained by answering the question: "when can an expression of this type be used in the place of an expression of that other type?". A first partial answer is that an expression can be substituted by another only if their types have the same form: for example a function can only be used in the place of another function (i.e. arrow types can be compared only with other arrow types), a pair instead of another pair and so on. Thus, for each type there is a different rule that answers the question:

1. A pair can be used instead of another if and only if each of its component can be so. Thus the subtyping relation is extended to product types by the componentwise ordering
2. A function returning a certain type can be used instead of one returning a greater since it causes no harm to substitute a result of a certain type by one of a smaller type. Furthermore a function accepting arguments of a given type, works also for arguments of smaller type and thus it can substitute it.
3. An overloaded function can substitute another overloaded function if and only if for every branch of the latter there is at least one in the former that can substitute it.

Rules for Refinement
We already pointed out the three conditions that a class defined by refinement must undertake. Let us reformulate them in term of overloading:

1. *state coherence*: An instance variable of a class must appear in all subtypes of that class and with always the same type

2. *covariance*: In an overloaded type, if an input type is a subtype of another input type then their corresponding output types must be in the same relation

3. *multiple inheritance*: In an overloaded type if two unrelated input types have a common subtype then for every maximal type of the set of their subtypes there must be one branch whose input type is that maximal type.

The reasons for these restrictions are very simple:

An inherited method can be used on the instance variables of a new class only if they contain those of the class in which the method has been defined; furthermore the " inherited" instance variables must always have the same type: at first sight one would say that these instance variables could be typed also by a subtype of the type they had in the super-class (this is what for example happens in O_2 [BDe92]); though, this type discipline would not be type safe. Let us show it by an example: suppose that in a given class C there is an instance variable x of type *Real*, and a method whose body is (`update{x=3.7}`); now define a subclass C' of C which redefined the type of x to *int* (note that $Int < Real$) and inherits the method from C; when the message corresponding to the method is sent to an instance of C' then the value 3.7, *which is not integer*, is assigned to x, even if x must contain an integer.

The covariance condition is introduced in order to assure that types always decrease during the computation. Indeed note that since the selection of the branch is based (by late binding) on the type a term possesses during computation and since types change at runtime, then the selected branch also changes with them. In other words, in an overloaded function the branch which would be chosen at compile time may be different from the one effectively chosen in the computation. We already met this phenomenon when explaining the late binding with the example of `isOrigin`; recall that the body of `isOrigin` was of the form

```
fn(p:2DPoint) => ..  [ p norm ] ..
```

Since `p` has type `2DPoint` the method selected at compile time would be the one for `2DPoint`. But by late binding, if `isOrigin` is applied to a `2DColPoint` then the code for this last class is chosen at run time.

Therefore the types of different branches cannot be totally unrelated if we want the property of the decrease of type to hold: in the example above, type decreases if and only if the method associated to `norm` in `2DColPoint` returns a type smaller than or equal to the one returned by the method in `2DPoint`. And this is guaranteed by the covariance condition.

The inheritance condition, as formulated in the previous section, said that methods in common to more than one unrelated ancestor must be redefined to disambiguate the selection. To see that this is equivalent to the rule we have written above note that the definition of a class by refinement of some other classes exactly corresponds to define a common subtype of these classes, which is also maximal since it is not possible in the language to construct a type greater than another that has already been defined (we can add new types only as leaves of the dag of the type hierarchy). If two *unrelated* ancestors respond to a same message then they both appear as input type in the type of this message, and, thus, the condition says that a new branch (method) must be defined for the new maximal subtype. The requirement of uniqueness of input types, assures that there will be no ambiguity in the selection of the branch [8].

[8] Roughly speaking this means that we cannot have declared two classes with the same name. Indeed this uniqueness is only a necessary condition since there might still be two classes with the same name but

Finally note that it is useful to have the possibility of defining both multi-methods and methods returning an overloaded type: adding branches whose type has the form, say, $A \to \{B \to T\}$ is different from adding branches whose form is $(A \times B) \to T$: the conditions for inheritance might be satisfied in one case and not in the other. This will be shown by an example in section 2.6.2.

---

responding to disjoint sets of messages. Therefore this situation has to be detected and rejected during the type-checking

Chapter 2

The $\lambda\&$-calculus

La nature est un temple où de vivants piliers
Laissent parfois sortir de confuses paroles;
L'homme y passe à travers des forets de symboles
Qui l'observent avec des regards familiers

Charles Baudelaire
Les Fleurs du Mal (1861)

In the previous chapter we have tried to convey the intuition that the basic mechanisms of (one peculiar style of) object-oriented programming are overloading and late binding. In this chapter we define an extension of the simply typed lambda calculus to model these mechanisms. We call this calculus $\lambda\&$.

We first show the underlying intuition of the calculus, then we give its formal presentation and we prove that it enjoys some relevant properties. We end this chapter by hinting the connections between this calculus and the object-oriented language presented in the previous section. This chapter is based on a joint work with Giorgio Ghelli and Giuseppe Longo.

2.1 Informal presentation

An overloaded function is constituted by a set of ordinary functions (i.e. lambda-abstractions), each one forming a different branch. To glue together these functions in an overloaded one we have chosen the symbol &; thus we have added to the simply typed lambda calculus the term

$$(M\&N)$$

which intuitively denotes an overloaded function of two branches, M and N, that will be selected according to the type of the argument. We must distinguish ordinary application from the application of an overloaded function since, as we tried to explain in section 1.1.3, they constitute different mechanisms. Thus we use "$\bullet$" to denote the overloaded application and "$\cdot$" for the usual one. Overloaded functions are built as it is customary with lists, by starting with an *empty* overloaded function that we denote by ε, and by concatenating new

branches by means of &. Thus in the term above M is an overloaded function while N is a regular function, which we call a "branch" of the resulting overloaded function. Therefore an overloaded function with n branches $M_1, M_2, \ldots M_n$ can be written as

$$((\ldots((\varepsilon\&M_1)\&M_2)\ldots)\&M_n)$$

The type of an overloaded function is the ordered set of the types of its branches.[1] Thus if $M_i{:}U_i \to V_i$ then the overloaded function above has type

$$\{U_1 \to V_1, U_2 \to V_2, \ldots, U_n \to V_n\}$$

and if we pass to this function an argument N of type U_j then the selected branch will be M_j. That is:

$$(\varepsilon\&M_1\&\ldots\&M_n)\bullet N \;\rhd^*\; M_j \cdot N \qquad (2.1)$$

We have also a subtyping relation on types. Its intuitive meaning is that $U \leq V$ if and only if any expression of U can be safely used in the place of an expression of V. An overloaded function can be used in the place of another when for each branch of the latter there is one branch in the former that can substitute it; thus, an overloaded type U is smaller than another overloaded type V if and only if for any arrow type in V there is at least one smaller arrow type in U.

Due to subtyping, the type of N in the expression above may not match any of the U_i but it may be a subtype of some of them. In this case we choose the branch whose U_i "best approximates" the type, say, U of N; i.e. we select the branch z s.t. $U_z = \min\{U_i | U \leq U_i\}$.

2.1.1 Subtyping, run-time types and late binding

It is important to notice that, because of subtyping, in this system types evolve during computation. This reflects the fact that, in languages with subtypes, the run-time types of the values of an expression are not necessarily equal to its compile-time type, but are always subtypes of that compile-time type. In the same way, in this system, the types of all the reducts of an expression are always smaller than or equal to the type of the expression itself.

The meaning of terms like "run-time type" and "compile-time type" is reasonably clear in the context of a traditional, eagerly evaluated programming language: in that case, a single term, such as an occurrence of a formal parameter x of a function, is "evaluated" many times, once each time the function is called. Each time x is bound to a value, the run-time type of that value becomes the "run-time type" of x, while in the source code that occurrence of x has a unique compile-time type, the one written by the programmer. However, the "compile-time type" of a term and the "run-time types" of its values are not unrelated: the property holds that all the run-time types of the values will be subtypes of the unique compile-time type of the term.

This distinction may not be intuitive in the context of a rewriting system, such as λ-calculus, where a more formal definition is needed. To follow the different "evaluations" of an occurrence of a term, we may use the notion of *residual* of an occurrence of a term (see

[1] This is just a first approximation; see later for the exact meaning of overloaded types.

[Bar84] where this definition is used only when the term is a redex). Intuitively, a *residual* is what the term has become after a reduction. As happens in traditional languages, in a rewriting system an occurrence of a term has many different residuals with possibly many different types, which are only guaranteed to be subtypes of the original one.

We will adopt the following definition: when a term is closed and normal, we then say that it is "a value",[2] and we mean by this that it cannot evolve anymore (since it is invariant by substitution and reduction). We similarly say that its type is "a run-time type", which means that no more information can be specified about the type of that term. The type of a *value* which is the residual of a given term is a *run-time type* for that term.

Thus the relation between a compile-time type and a run-time type is the same as the relation between a term and a value: a value for a term is any closed normal form obtained by performing reductions and substitutions over that term; a run-time type for that term is the type of any of its values. Note that an open term is bound, during a computation, to many different values, and so it gets many different run-time types. Note also that we did not formally define the notion of "evolution of a term", thought it would be possible. We are now just trying to convey the intuition behind the idea of run-time types, while the formal definition of the reduction rules is given in the next section.

Now that we have outlined the notion of run-time type, we can also define late binding in λ&: overloading is implemented by late binding when the selection of the branch is based the run-time types. This can be obtained by requiring that a reduction as (2.1) can be performed only if N is a closed normal form, and that the chosen branch depends on the type of the reduced term. This is late-binding since the branch choice cannot be performed before evaluating the argument, and this choice does not depend on the compile-time type of the expression which generated the value, but on the run-time type of the value itself.

Although the selection of the branches of overloaded functions is based on the run-time types, the static typing of a term must be enough to assure that the computation will be type-error free. This can be obtained by guaranteeing that types can only decrease during computation (so that the run-time type of any residual of a term is always a subtype of its compile-time type) and thus that well-typed terms rewrite to well-typed terms (see theorem 2.3.2). To guarantee it a "consistency" condition must be imposed on overloaded types. In short, an overloaded type $\{U_i \to V_i\}_{i\in I}$ is well-formed if and only if for all $i, j \in I$ it satisfies the following conditions:

$$U_i \leq U_j \quad \Rightarrow \quad V_i \leq V_j \tag{2.2}$$

$$U_i \Downarrow U_j \quad \Rightarrow \quad \text{there exists a unique } z \in I \text{ such that } U_z = \inf\{U_i, U_j\} \tag{2.3}$$

where $U_i \Downarrow U_j$ means that U_i and U_j are downward compatible, i.e. they have a common lower bound.

Condition (2.2) is a consistency condition, which assures that during computation the type of a term may only decrease. In a sense, this takes care of the common need for some sort of covariance of the arrow in the practice of programming. More specifically if we have a two-branched overloaded function M of type $\{U_1 \to V_1, U_2 \to V_2\}$ with $U_2 < U_1$ and we pass it a term N which at compile-time has type U_1 then the compile-time type of $M \bullet N$ will

[2]For example $\lambda x.\lambda y.x$ and $\lambda x.\lambda y.y$ are the only two values of type $Bool \equiv \alpha \to \alpha \to \alpha$.

be V_1; but if the normal form of N has type U_2 then the run-time type of $M{\bullet}N$ will be V_2 and therefore $V_2 < V_1$ must hold. The second condition concerns the selection of the correct branch: we said before that if we apply an overloaded function of type $\{U_i \to V_i\}_{i\in I}$ to a term of type U then the selected branch has type $U_j \to V_j$ such that $U_j = \min_{i\in I}\{U_i | U \leq U_i\}$; condition (2.3) assures the existence and uniqueness of this branch.[3]

It is not very surprising that these conditions strongly resemble the rules of *covariance* and *multiple inheritance* we met in section 1.2.2 when dealing with the types system of the toy language.

2.2 The syntax of the λ&-calculus

In this section we define the extension of the typed lambda calculus we study in the rest of the paper. We use the following conventions: A, B denote Atomic Types, $S, T, U, V, W \ldots$ denote (Pre)Types, M, N, P, Q, denote Terms, H, I, J, K denote sets of indexes and h, i, j, k, n indexes. We first define a set of *Pretypes* and then from them we select those that satisfy the conditions (2.2) and (2.3) and that constitute the types.

$$\textbf{PreTypes} \qquad V \quad ::= \quad A \mid V \to V \mid \{V_1' \to V_1'', \ldots, V_n' \to V_n''\}$$

For technical reasons we consider overloaded types as lists, i.e. possessing an order; the list may also be empty: in this case the type is denoted by $\{\}$.

2.2.1 Subtyping rules.

We define a *subtyping* relation on the set of Pretypes. This relation is used to define the types. The idea is that one may start from a partial order which is predefined on atomic (pre)types and extend it to a preorder on all Pretypes: the relation is obtained by adding the rules of transitive and reflexive closure to the following ones[4] :

$$\frac{U_2 \leq U_1 \quad V_1 \leq V_2}{U_1 \to V_1 \leq U_2 \to V_2}$$

$$\frac{\text{for all } i \in I, \text{ there exists } j \in J \text{ such that } U_i'' \leq U_j' \text{ and } V_j' \leq V_i''}{\{U_j' \to V_j'\}_{j\in J} \leq \{U_i'' \to V_i''\}_{i\in I}}$$

Intuitively, if we consider two overloaded types U and V as sets of functional types then the last rule states that $U \leq V$ if and only if for every type in V there is some type in U smaller than it. In contrast to the usual partial order on record types, the cardinalities of I and J are unrelated.

Note that $\leq$ is just a preorder, and not a partial order, for $U \leq V$ and $V \leq U$ do not imply $U = V$. For example take $S' \leq S$, then $S \to T \leq S' \to T$, and thus $\{S \to T\} \lesseqgtr$

[3] By the way note how these conditions are very related to the regularity condition discussed in [GM89], in the quite different framework of order-sorted algebras and order-sorted rewriting systems

[4] Strictly speaking we should define $\Sigma \vdash S \leq T$ where Σ parameterizes the definition and describes the subtyping relation on atomic types; i.e. it is the axiom set of the theory of subtyping

$\{S \to T, S' \to T\}$. Given a preorder on a set it is possible to define the following equivalence relation: $U \sim V$ if and only if $U \leq V$ and $V \leq U$ (this is a congruence w.r.t. the type constructors). And the preorder induces an order on the set of equivalence classes.

In the rest of this chapter we will informally suppose to work on an order, indeed to work on **Types**$/\sim$, in order to gain in clarity in the exposition. We will point out by a footnote all the places where having a preorder instead of an order makes a difference.

We formally deal with the partial order in chapter 6 devoted to the semantics of λ&, where we interpret this partial order by an order in the model (see in particular section 6.2).

In our presentation of the subtype rules we implicitly defined a transitivity rule. We can easily prove that this rule is not really needed.

Theorem 2.2.1 (Transitivity Elimination) *$\vdash T \leq U$ if and only if $\vdash^- T \leq U$ where $\vdash^-$ is defined by the rules of $\vdash$ minus transitivity.*

Proof. Observe that if $\vdash^- T \leq U$ then either T and U are both atomic types, or they are both arrow types or they are both overloaded types. We first prove that $\vdash^- T \leq U$ and $\vdash^- U \leq V$ implies $\vdash^- T \leq V$ by induction on the size of T, U, V. If they are all atomic types the thesis is immediate. If $T = \{T'_i \to T''_i\}_{i \in I}$, $U = \{U'_j \to U''_j\}_{j \in J}$ and $V = \{V'_l \to V''_l\}_{l \in L}$, then for all $l \in L$ exists $j \in J$ such that $\vdash^- U'_j \to U''_j \leq V'_l \to V''_l$ and for all $j \in J$ exists $i \in I$ such that $\vdash^- T'_i \to T''_i \leq U'_j \to U''_j$. By induction, for all $l \in L$ exists $i \in I$ such that $\vdash^- T'_i \to T''_i \leq V'_l \to V''_l$, hence $\vdash^- \{T'_i \to T''_i\}_{i \in I} \leq \{V'_l \to V''_l\}_{l \in L}$, q.e.d.. The arrow case is similar and simpler. Now the theorem follows by induction on the proof of $\vdash T \leq U$ and by cases on the last applied rule, where the only interesting case is transitivity. $\Box$

The interest of this theorem is that it proves that the subtyping rules given above (plus reflexivity on atomic types) describe a deterministic algorithm. This means that the decidability of $\leq$ on the atomic types implies its decidability on all pretypes and, thus on **Types** too.

2.2.2 Types

Our system is an extended strongly typed λ-calculus. Arrow types and overloaded types are defined inductively from atomic types. As mentioned in the introduction, the overloaded types have a good formation rule that allows a consistent application of the reduction rules.

1. $A \in$ **Types**
2. if $V_1, V_2 \in$ **Types** then $V_1 \to V_2 \in$ **Types**
3. if for all $i, j \in I$
   - **(a)** ($U_i, V_i \in$ **Types**) and
   - **(b)** ($U_i \leq U_j \Rightarrow V_i \leq V_j$) and
   - **(c)** ($U_i \Downarrow U_j \Rightarrow$ there exist a unique $h \in I$ such that $U_h = \inf\{U_i, U_j\}$)[5].

   then $\{U_i \to V_i\}_{i \in I} \in$ **Types**

[5]This notation is not very precise; since $\leq$ is just a preorder. A set generally has many equivalent g.l.b.'s; we should then write $U_h \in \inf\{U_i, U_j\}$ or $[U_h]_\sim = \inf\{U_i, U_j\}$

In a system with subtyping, if $f: U \to V$, this means that when f is applied to a term a with a run-time type $U' \leq U$, the run-time type of the result will be a type $V' \leq V$. Intuitively, an overloaded type $\{U_i \to V_i\}_{i \in I}$ is inhabited by functions, made out of different pieces of code, such that when they are applied to a term whose run-time type U' is the subtype of some U_i, the run-time type of the result will be a subtype V' of the corresponding V_i. This is assured by condition **(b)** above.

To ensure the existence of an *inf* for any pair of downward compatible types, we require that $\leq$ yields a "partial lattice" on Atomic Types. In accordance with the rules given in the previous section, the whole **Types** inherits this structure.[6] In object-oriented languages this is not always the case. We can distinguish object-oriented languages where Atomic Types have a tree structure (the so called "single inheritance") and object-oriented languages where Atomic Types have a free order relation and where additional structure is used to solve the problems caused by compatible types without an *inf*. The same kind of technique can be used to extend our approach to this situation (we will do it in section 4.1.1), since the partial lattice property is not essential, but is useful for getting a simple branch selection rule, as described in the section on reduction and a simpler formulation of some technical definitions we will use in giving the semantics (cfs. 6.2). Likewise, while condition **(b)** above is an essential feature of our approach, condition **(c)** is linked to the branch selection rule, and could easily be modified (see [Cas93b, Ghe91]). In particular we will propose in section 4.1 a weakening of condition **(c)** to deal with the so-called "multiple-inheritance" without a class-precedence list.

Furthermore, we suppose that the subtyping relation is decidable on atomic types, which implies that it is decidable on **Types** as well. Note that this poses no problem in the current (simple) approach, as we have fixed atomic types; more work would be needed in order to allow the programmer to define its own base types.

Henceforth we only deal with **Types** and completely forget **PreTypes**; thus we will intend that all the pretypes which appear in the rest of the paper satisfy the conditions above.

2.2.3 Terms

Roughly speaking, terms correspond to terms of the classical lambda calculus plus operations to build and apply overloaded functions. Overloaded functions are built as customary with lists, by starting from an empty overloaded function and adding branches with the & operator. We distinguish the usual application $M \cdot M$ of lambda-calculus from the application of an overloaded function $M \bullet M$ since they constitute two completely different mechanisms: indeed a notion of variable substitution is associated with the former, while in the latter there is the notion of selection of a branch. This is also stressed by the proof-theoretical viewpoint where these constructors correspond to two different elimination rules. Finally, a further difference, specified in the reduction rules, is that overloaded application is associated with call by value, which is not needed by the ordinary application. For the same reason we must distinguish between the type $U \to V$ and the overloaded function type with just one branch $\{U \to V\}$.[7]

[6] More precisely, since $\leq$ is not an order, it is **Types** modulo $\sim$ which inherits the partial lattice structure.

[7] In section 4.3 we define a calculus in which these two types will coincide

However, in some cases it will be useful to have only one notation to deal with both kinds of application; for this aim the simple juxtaposition will be used.

Variables are indexed by their type, to avoid the use of type environments in the type-checking rules.

$$\textbf{Terms} \qquad M \quad ::= \quad x^V \mid c \mid \lambda x^V.M \mid M \cdot M \mid \varepsilon \mid M\&^V M \mid M \bullet M$$

The type which indexes the & is a technical trick to allow the reduction inside overloaded function, as explained later on (see page 93). c represents generic constants while ε is a distinct constant for the empty overloaded function.

Hereafter we may omit the type indexing of &, when it will be clear from the context, and the ε at the beginning of &-terms, in the examples.

2.2.4 Type checking

We define here the typing relation ":", a proper subset of **Terms**×**Types**. Therefore, as already pointed out, in the rules below we omit the condition $V \in \textbf{Types}$. This means that, all the *PreTypes* that appear in the following rules are to be considered as well-formed types. Anyway we observe that an algorithm implementing the following type-checking rules should check that the types appearing in the conclusions of the rules [Taut], [→ Intro] and [{}Intro] are well-formed.

We use the notation $\vdash M : V \leq U$ as a shorthand for the conjunction "$\vdash M : V$ and $V \leq U$".

[Taut] $$\vdash x^V : V$$

[→ Intro] $$\frac{\vdash M : V}{\vdash \lambda x^U.M : U \to V}$$

[→ Elim$_{(\leq)}$] $$\frac{\vdash M : U \to V \quad \vdash N : W \leq U}{\vdash M \cdot N : V}$$

[Taut$_\varepsilon$] $$\vdash \varepsilon : \{\}$$

[{}Intro] $$\frac{\vdash M : W_1 \leq \{U_i \to V_i\}_{i \leq (n-1)} \quad \vdash N : W_2 \leq U_n \to V_n}{\vdash (M \&^{\{U_i \to V_i\}_{i \leq n}} N) : \{U_i \to V_i\}_{i \leq n}}$$

[{}Elim] $$\frac{\vdash M : \{U_i \to V_i\}_{i \in I} \quad \vdash N : U \quad U_j = \min_{i \in I}\{U_i | U \leq U_i\}}{\vdash M \bullet N : V_j}$$

In the last rule the premise on U_j as well as the type constraints are indeed *meta-premises*, i.e. they are conditions to the application of the rules but they do not belong to the tree-structure of the deduction. The empty term ε and the empty type $\{\}$ are used to start the formation of overloaded terms and types. We read $M\&N\&P$ as $(M\&N)\&P$.

As the careful reader will have noted, we do not use the *subsumption* rule (see below) in type-checking. We utilized a slightly different type discipline, where the use of subsumption is distributed where needed. The resulting system is equivalent, in the sense explained below, to the subsumption discipline, but every term possesses a unique type, which simplifies the definition of the operational semantics and some proofs.

Consider the functional core of our system, i.e. only the first three typing rules at the beginning of this section and let denote this system by $\vdash_{\leq}$. The subsumption system (denoted by $\vdash_{sub}$) is obtained from this one by replacing $\vdash N\colon W \leq U$ with $\vdash N\colon U$ in $[\rightarrow \text{ELIM}_{(\leq)}]$ and by adding the subsumption rule:

$$[\rightarrow\text{ELIM}]\quad \frac{\vdash_{sub} M\colon U \rightarrow V \quad \vdash_{sub} N\colon U}{\vdash_{sub} MN\colon V} \qquad\qquad [\text{SUBSUMPTION}]\quad \frac{\vdash_{sub} M\colon U \quad U \leq V}{\vdash_{sub} M\colon V}$$

Now, we can prove the following theorem.

Theorem 2.2.2 $\vdash_{\leq} M\colon V$ *if and only if* $V = \min\{U | \vdash_{sub} M\colon U\}$ *(which implies that the set* $\{U | \vdash_{sub} M\colon U\}$ *is not empty).*

Proof. ($\Rightarrow$) By induction on the proof of $\vdash_{\leq} M\colon V$ and by cases on the last applied rule.

($\Leftarrow$) By induction on the smallest proof that $\vdash_{sub} M\colon V$ and by cases on the last applied rule. □

Corollary 2.2.3 *Every well-typed λ&-term possesses a unique type*

In conclusion, the theorem states that $\vdash_{\leq}$ is equivalent to $\vdash_{sub}$ in the sense that it always returns the smallest (i.e. most precise) type returned by the subsumption system. This theorem suggests that it is possible to define a subsumption based version $\vdash_{sub}$ for the full system too. We must add subsumption, substitute all judgements $\vdash N : T \leq U$ in the rules with $\vdash N : U$ and finally, in the $[\{\}\text{ELIM}]$ rule, substitute $\vdash N : U$ with "U is the minimum type such that $\vdash_{sub} N : U$".

We can then extend theorem 2.2.2 to our entire calculus.

Theorem 2.2.4 (Subsumption Elimination) *For the whole λ&,* $\vdash M\colon V$ *if and only if* $V = \min\{U | \vdash_{sub} M\colon U\}$

Since we have chosen the subsumption-free presentation, every term possesses a unique type, because there is a unique derivation for the type of a term.

Note that the lack of type variables makes the proof of subsumption elimination and transitivity elimination much easier for this calculus than for $F_{\leq}$ (see [CG92]).

Finally observe that even if $\leq$ is just a preorder it makes perfect sense w.r.t. type-checking. Suppose that $M\colon \{U_i \rightarrow V_i\}_{i \in J} \leq \{U'_i \rightarrow V'_i\}_{i \in I}$. The intended meaning of this subtyping relation is that M can be fed with any input N which would be acceptable for a term M' in $\{U'_i \rightarrow V'_i\}_{i \in I}$, and that the output can be used in any context where $M' \bullet N$ would be accepted. Indeed, let $N\colon U'_i$ and $C[\,]$ be a context where a value of type V'_i can be put. Then, for some $j \in J$, $U_j \rightarrow V_j \leq U'_i \rightarrow V'_i$, so that $U'_i \leq U_j$ and $V_j \leq V'_i$, hence the application $M \bullet N$ type-checks and can be used in the context $C[M \bullet N]$. Our "$\leq$" is the least (or less fine) preorder that one can define with this property.

The fact that two types may be equivalent even if different simply means that, from the type-checker's point of view, they are completely interchangeable: one or the other makes no difference (if we used subsumption the two types would denote the same set of terms). Though, we do not study here (as it could be done) the set of types modulo the equivalence relation since we are developing the syntax of the system, and syntactically these types are different. If you prefer, we placed ourselves in the place of the average programmer on the street who, we believe, would not like to work with a system that equates two types with a different number of branches, nor he would want that the system returned a type different from the one he expected, just because it is the canonical representative of the equivalence class that types that term (take $U \leq V$ and imagine that the system would respond to the programmer that $(M \,\&^{\{U \to T, V \to T\}} N)$ has type $\{V \to T\}$). This "problem" will be tackled when dealing with the semantics where one wants equivalent types to have the same interpretation (see section 6.2).

2.2.5 Reduction Rules

In order to simplify the definition of the system, we consider the types of overloaded functions as ordered sets, where the order corresponds, more or less, to the order in which branches are added when an overloaded function is built. However the reader may note that this order is completely irrelevant in subtyping and typing rules, with the only exception of [{}INTRO], where we want to be able to distinguish the only arrow type associated with the right hand side of the & from the set of the other ones. But for that it would suffice to consider only the indexes ordered (this will be better understood in chapter 9). Exactly the same information is all that is needed by the reduction rules.

As we mentioned before, the run-time types are used during computation to perform branch selection. Thus, we have to define what the run-time type of a term is. We propose here a simple solution: the deduction system that infers the run-time type of a term is the same as the one used for type-checking. What distinguishes run-time types and compile-time types is thus the time when the deduction is made. In fact, during the computation the type of a term may change since reduction and substitution may decrease the type of a term (as shown in Theorems 2.3.1 and 2.3.2).

> We say that the type of a term is its run-time type when that term is a "value", i.e. when it is normal and closed; a run-time type of a residual of a term is also a run-time type of the term. We allow a reduction of the application of an overloaded function only when the argument is a value, i.e. when it is typed by a run-time type.

This is a crucial point. If we allowed selecting the branch of an overloaded function on the basis of the type of an argument whose type could still be decreased (by reduction or by substitution) then the selection would give different results depending on the time when it is applied, and the system would be no longer confluent. However this discipline can be weaken as we show in section 4.1.3.

As a matter of fact, this call-by-value constraint is not a limitation if our aim is to model object-oriented languages. In these languages message passing evaluation always requires

that the receiving object has been fully evaluated.[8]

We start by defining, in a standard way, substitutions on the terms of our system:

Definition 2.2.5 (Substitution) *We define the term $M[x^T := N]$ by induction on the structure of M:*

1. $x^T[x^T := N] \equiv N$
2. $y^S[x^T := N] \equiv y^S$ *if* $y^S \not\equiv x^T$
3. $\varepsilon[x^T := N] \equiv \varepsilon$
4. $(\lambda y^S.P)[x^T := N] \equiv (\lambda y^S.(P[x^T := N]))$ *where y is not free in N*
5. $(P\&^V Q)[x^T := N] \equiv ((P[x^T := N])\&^V(Q[x^T := N]))$
6. $(PQ)[x^T := N] \equiv (P[x^T := N])(Q[x^T := N])$

□

Of course, this definition only makes sense when the type of N is a subtype of T. Note that in *5*, even if the types of the subterms change, the type of the whole term is always the same, since it is frozen in the index of the &; thus the selection of the branch does not depend on the *degree* of reduction of the &-term. This is a decisive point in our approach, which makes the system type-safe though reductions within an overloaded term are allowed.

We define the one-step reduction relation $\triangleright$ which is a proper subset of **Terms** × **Terms**. We denote by $\triangleright^*$ its reflexive and transitive closure, under the usual conditions (in β) to avoid free variables being captured:

β) $(\lambda x^S.M)N \triangleright M[x^S := N]$

$\beta_\&$) If $N{:}U$ is closed and in normal form and $U_j = min\{U_i | U \leq U_i\}$ then

$$((M_1\&^{\{U_i \to V_i\}_{i=1..n}}M_2)\bullet N) \triangleright \begin{cases} M_1\bullet N & \text{for } j < n \\ M_2 \cdot N & \text{for } j = n \end{cases}$$

context) If $M_1 \triangleright M_2$ then

$$(M_1 \cdot N) \triangleright (M_2 \cdot N) \quad (2.4)$$
$$(N \cdot M_1) \triangleright (N \cdot M_2) \quad (2.5)$$
$$(M_1\bullet N) \triangleright (M_2\bullet N) \quad (2.6)$$
$$(N\bullet M_1) \triangleright (N\bullet M_2) \quad (2.7)$$
$$(\lambda x^U.M_1) \triangleright (\lambda x^U.M_2) \quad (2.8)$$
$$(M_1\&N) \triangleright (M_2\&N) \quad (2.9)$$
$$(N\&M_1) \triangleright (N\&M_2) \quad (2.10)$$

[8] This happens not only for the essential reason we pointed out (the run-time type is generally only known for fully evaluated terms) but also since object oriented languages heavily rely on state and state-updating operations, and programs using updates are much more readable if eagerly evaluated.

The intuitive operational meaning of $(\beta_{\&})$ is easily understood when looking at the simple case, i.e. when there are as many branches as arrows in the overloaded type. In this case, under the assumptions in the rule:

$$(\varepsilon \& M_1 \& \ldots \& M_n) \bullet N \ \rhd^* M_j \cdot N$$

However, in general, the number of branches of the overloaded function may be different from the number of arrows in the overloaded types, both since an overloaded function could begin with an application or with a variable, accounting for an initial segment of the overloaded type (they are just required to possess an overloaded type), and because of the subtyping relation used in the rule of [{}INTRO].

If we allowed $(\beta_{\&})$ reductions with open or non normal arguments the system would not be confluent, since the type of an open or non normal argument can be different in different phases of the computation. For example, consider a term

$$(\lambda x^V.((P \&^{\{V \to V, U \to U\}} M) \bullet x^V)) \cdot (N^U)$$

with $U \leq V$ (we superscript terms with their types, like in N^U, to increase readability of examples). If the inner $\beta_{\&}$ reduction were performed with the x argument (which is not closed), the first branch P would be chosen, while if the outer β reduction is performed first then the term becomes:

$$(P \&^{\{V \to V, U \to U\}} M) \bullet N^U$$

and the second branch M is (correctly) chosen. In short, the argument of an overloaded application must be closed and normal to perform the evaluation, since this is the only case where its type cannot decrease anymore, and describes the value as accurately as possible.

Complementary to the idea of freezing the argument of an overloaded application to its normal form, is the use of the type which indexes the &'s to freeze the type of &-terms. We outline two short examples to show the problems that arise with reduction and substitution inside &-terms without this index.

We suppose that $U' \leq U$ and $V' \leq V$; consider the term

$$F_1 \equiv (\varepsilon \& M^{U \to V} \& ((\lambda x^{U' \to V'}.x) \cdot N^{U \to V'}))$$

in which we have not specified any index; this can be intuitively typed as follows:

$$F_1 \colon \{U \to V, U' \to V'\}$$

Though, if we reduce the right branch, without freezing the index, we are no longer able to recover a type for the contractum, namely for $(\varepsilon \& M^{U \to V} \& N^{U \to V'})$ since both terms possess the same input type. Consider next the term

$$F_2 \equiv \lambda y^{U' \to V'}.(\varepsilon \& (\lambda x^U.M^V) \& (y^{U' \to V'}))$$

again we can intuitively type it as follows.

$$F_2 \colon (U' \to V') \to \{U \to V, U' \to V'\}$$

but if we apply F_2 to $N^{U \to V'}$ and β-reduce, then we are in the same case as above.

Note, finally, that our calculus is truly type-dependent (that is, the type erasure of a term is not enough to forecast its evolution or meaning) for two different reasons:

- $\beta_{\&}$ reduction depends on the type of the argument
- $\beta_{\&}$ reduction depends on the index T of the & in the overloaded term

More specifically, if $\{U_i \to V_i\}_{i=1..n}$ is the index of &, $\beta_{\&}$ reduction depends on the list $[U_i]_{i=1..n}$ of the *input types* of the overloaded function. For example, if $U' \leq U$, both terms

$$(\varepsilon \& M^{U\to V} \&^{\{U\to V, U'\to V\}} M'^{U\to V})$$

and

$$(\varepsilon \& M^{U\to V} \&^{\{U'\to V, U\to V\}} M'^{U\to V})$$

are well-typed, but they behave differently if applied to a normal closed term N^U.

Note that we are here in a different and more flexible situation than in object-oriented languages, since in those languages every branch of an overloaded function (every method) must be understood as a λ-abstraction (when viewing methods as global, overloaded functions in our sense.) In this language, on the other hand, any expression with a functional type (in particular an application) can be concatenated by using &. Thus, when following the object-oriented style, the left hand side U of the type $U \to V$ of an expression $\lambda x^U.M$ does not change when reductions and substitutions are performed inside $\lambda x^U.M$. In our approach, when reducing inside an &, one may obtain a smaller type for the reductum, in particular a larger U in a type $U \to V$. To allow this possibility of "inside" reductions and preserve determinism, we label the &'s with types .

2.3 The Generalized Subject Reduction Theorem

The Subject Reduction Theorem in classical λ-calculus proves that the type of a term does not change when the term is reduced. In this section, we generalize this theorem for our calculus, since we prove that if a term is typable in our system, then it can only be reduced to typable terms and that these terms have a type smaller than or equal to the type of the redex.

In order to enhance readability, in this and in the following section, we will often omit the turn-style symbol.

Lemma 2.3.1 (Substitution Lemma) *Let $M{:}U$, $N{:}T'$ and $T' \leq T$.*
Then $M[x^T{:=}N]{:}U'$, where $U' \leq U$.

Proof. By induction on the structure of M.

$M \equiv \varepsilon$ straightforward

$M \equiv x$ straightforward

$M \equiv y \not\equiv x$ straightforward

$M \equiv \lambda x^V.M'$ straightforward

$M \equiv \lambda y^V.M'$ Then $U = V \to W$ and $M'\colon W$.

By induction hypothesis $M'[x := N]\colon W' \leq W$, therefore $M[x := N] \equiv \lambda y^V.M'[x := N]\colon V \to W' \leq V \to W$

$M \equiv (M_1 \&^T M_2)$ Then $M[x := N] \equiv (M_1[x := N] \&^T M_2[x := N])$; by induction hypothesis and the rule [{}INTRO] $M[x := N]$ is well-typed and its type is the same of the one of M that is T.

$M \equiv M_1 \cdot M_2$ where $M_1\colon V \to U$ and $M_2\colon W \leq V$. By induction hypothesis:

$M_1[x := N]\colon V' \to U'$ with $V \leq V'$ and $U' \leq U$

$M_2[x := N]\colon W'$ with $W' \leq W$

Since $W' \leq W \leq V \leq V'$ we can apply the rule [$\to$ELIM$_{(\leq)}$] and thus $M[x := N] \equiv (M_1[x := N]) \cdot (M_2[x := N])\colon U' \leq U$

$M \equiv M_1 \bullet M_2$ where $M_1\colon \{V_i \to W_i\}_{i \in I}$ and $M_2\colon V$.

Let $V_h = \min_{i \in I}\{V_i | V \leq V_i\}$. Thus $U = W_h$.

By induction hypothesis:

$M_1[x := N]\colon \{V'_j \to W'_j\}_{j \in J}$ with $\{V'_j \to W'_j\}_{j \in J} \leq \{V_i \to W_i\}_{i \in I}$

$M_2[x := N]\colon V'$ with $V' \leq V$

Let $V'_k = \min_{j \in J}\{V'_j | V' \leq V'_j\}$. Thus $M[x := N]\colon W'_k$. Therefore we have to prove that $W'_k \leq W_h$

As $\{V'_j \to W'_j\}_{j \in J} \leq \{V_i \to W_i\}_{i \in I}$ then for all $i \in I$ there exists $j \in J$ such that $V'_j \to W'_j \leq V_i \to W_i$. Given $i = h$ we chose an $\tilde{h} \in J$ which satisfies this condition: that is,

$$V'_{\tilde{h}} \to W'_{\tilde{h}} \leq V_h \to W_h \tag{2.11}$$

We now have the following inequalities:

$$V \leq V_h \tag{2.12}$$

by the definition of V_h, as $V_h = \min_{i \in I}\{V_i | V \leq V_i\}$;

$$V_h \leq V'_{\tilde{h}} \tag{2.13}$$

which follows from (2.11);

$$V' \leq V'_{\tilde{h}} \tag{2.14}$$

which follows from (2.12), (2.13) and $V' \leq V$;

$$W'_{\tilde{h}} \leq W_h \tag{2.15}$$

which follows from (2.11).

$$V'_k \leq V'_{\tilde{h}} \tag{2.16}$$

which follows from (2.14), as $V'_{\tilde{h}}$ belongs to a set with V'_k as least element. Finally,

$$W'_k \leq W'_{\tilde{h}} \tag{2.17}$$

follows from (2.16) and from the covariance rule on $\{V'_j \to W'_j\}_{j\in J}$

Thus, by (2.15) and (2.17), $W'_k \leq W_h$

□

Theorem 2.3.2 (Generalized Subject Reduction) *Let $M{:}U$. If $M \rhd^* N$ then $N{:}U'$, where $U' \leq U$.*

Proof. It suffices to prove the theorem for $\rhd$; the thesis follows from a simple induction on the number of steps of the reduction. Thus, we proceed by induction on the structure of M:

$M\equiv x$ x is in normal form and the thesis is straightforwardly satisfied.

$M\equiv\varepsilon$ as in the previous case.

$M\equiv\lambda x^V.P$. The only case of reduction is that $P \rhd P'$ and $N\equiv\lambda x^V.P'$; but from the induction hypothesis it follows that N is well-typed and the type of the codomain of N will be less than or equal to the one of M; since the domains are the same, the thesis thus holds.

$M\equiv(M_1\&^T M_2)$. Just note that whenever M is reduced it is still well-typed (apply the induction hypothesis) and its type doesn't change.

$M\equiv M_1 \cdot M_2$ where $M_1{:}V \to U$ and $M_2{:}W \leq V$. We have three subcases:

1. $M_1 \rhd M'_1$, then by induction hypothesis $M'_1{:}V' \to U'$ with $V \leq V'$ and $U' \leq U$. Since $W \leq V \leq V'$, then by rule $[\to\text{ELIM}_{(\leq)}]$ we obtain $M'_1 M_2{:}U' \leq U$.
2. $M_2 \rhd M'_2$, then by induction hypothesis $M'_2{:}W'$ with $W' \leq W$. Again, $W' \leq W \leq V$ and, thus, by $[\to\text{ELIM}_{(\leq)}]$ we obtain $M_1 M'_2{:}U$.
3. $M_1\equiv\lambda x^V.M_3$ and $M \rhd M_3[x{:=}M_2]$, with $M_3{:}U$. Thus, by Lemma 2.3.1, $M_3[x{:=}M_2]{:}U'$ with $U' \leq U$.

$M\equiv M_1\bullet M_2$ where $M_1{:}\{V_i \to W_i\}_{i\in I}$ and $M_2{:}V$.

Let $V_h = \min_{i\in I}\{V_i | V \leq V_i\}$. Thus $U = W_h$. Again we have three subcases:

1. $M_1 \rhd M'_1$ then by induction $M'_1{:}\{V'_j \to W'_j\}_{j\in J}$ with $\{V'_j \to W'_j\}_{j\in J} \leq \{V_i \to W_i\}_{i\in I}$. Let $V'_k = \min_{j\in J}\{V'_j | V \leq V'_j\}$. Thus $M'_1\bullet M_2{:}W'_k$. Therefore we have to prove that $W'_k \leq W_h$
   Since $\{V'_j \to W'_j\}_{j\in J} \leq \{V_i \to W_i\}_{i\in I}$, then for all $i \in I$ there exists $j \in J$ such that $V'_j \to W'_j \leq V_i \to W_i$. For $i = h$ we choose a certain $\tilde{h} \in J$ which satisfies this condition. That is:

$$V'_{\tilde{h}} \to W'_{\tilde{h}} \leq V_h \to W_h \tag{2.18}$$

We now have the following inequalities:

$$V \leq V_h \tag{2.19}$$

by hypothesis, since $V_h = \min_{i\in I}\{V_i | V \leq V_i\}$;

$$V_h \leq V'_{\tilde{h}} \tag{2.20}$$

follows from (2.18);

$$V \leq V'_{\tilde{h}} \tag{2.21}$$

follows from (2.19) and (2.20);

$$W'_{\tilde{h}} \leq W_h \tag{2.22}$$

follows from (2.18);

$$V'_k \leq V'_{\tilde{h}} \tag{2.23}$$

by (2.21), since $V'_{\tilde{h}}$ belongs to a set with V'_k as least element;

$$W'_k \leq W'_{\tilde{h}} \tag{2.24}$$

follows from (2.23) and the covariance rule on $\{V'_j \to W'_j\}_{j\in J}$
Finally, by (2.22) and (2.24), one has that $W'_k \leq W_h$

2. $M_2 \triangleright M'_2$ then by induction hypothesis $M'_2 \colon V'$ with $V' \leq V$. Let $V_k = \min_{i\in I}\{V_i | V' \leq V_i\}$. Thus $M_1 \bullet M'_2 \colon W_k$. Since $V' \leq V \leq V_h$ then $V_k \leq V_h$; thus, by the covariance rule in $\{V_i \to W_i\}_{i\in I}$, we obtain $W_k \leq W_h$.

3. $M_1 \equiv (N_1 \& N_2)$ and M_2 is normal. Then we have two cases, that is $M \triangleright (N_1 \bullet M_2)$ (case $h < n$) or $M \triangleright (N_2 \cdot M_2)$ (case $h = n$). In both cases, by [{}Elim] or [$\to$Elim$_{(\leq)}$], according to the case, it is easy to show that the terms have type smaller than or equal to W_h.

□

2.4 Church-Rosser

In this section we prove that this system is Church-Rosser (**CR**). The proof is a simple application of a lemma due to Hindley [Hin64] and Rosen [Ros73]:

Lemma 2.4.1 (Hindley-Rosen) *Let R_1,R_2 be two notions of reduction. If R_1, R_2 are* **CR** *and $\triangleright^*_{R_1}$ commutes with $\triangleright^*_{R_2}$ then $R_1 \cup R_2$ is* **CR**.

Set now $R_1 \equiv \beta_\&$ and $R_2 \equiv \beta$; if we prove that these notions of reduction satisfy the hypotheses of the lemma above, we thus obtain **CR** for our system. It is easy to prove that β and $\beta_\&$ are **CR**: indeed, the first one is a well known result while for the other just note that $\beta_\&$ satisfies the diamond property.

Thus it remains to prove that the two notions of reduction commute, for which we need two technical lemmas.

Lemma 2.4.2 *If* $N \rhd^*_{\beta_\&} N'$ *then* $M[x := N] \rhd^*_{\beta_\&} M[x := N']$

Proof. The proof is done by induction on the structure of M and consists in a simple diagram chase

M	LHS	RHS	comment
ε	ε	ε	OK
x	N	N'	OK
y	y	y	OK
PQ	$P[\ ]Q[\ ]$	$P[\ ']Q[\ ']$	use the induction hypothesis
$\lambda y.P$	$\lambda y.P[\ ]$	$\lambda y.P[\ ']$	use the induction hypothesis
$(P\&Q)$	$(P[\ ]\&Q[\ ])$	$(P[\ ']\&Q[\ '])$	use the induction hypothesis

□

Lemma 2.4.3 *If* $M \rhd_{\beta_\&} M'$ *then* $M[x := N] \rhd_{\beta_\&} M'[x := N]$

Proof. We proceed by induction on the structure of $M \rhd M'$ (we omit the subscript in $\rhd_{\beta_\&}$ since there is no ambiguity here); we have the following cases:

CASE 1 $\lambda y.P \rhd \lambda y.P'$ the thesis follows from the induction hypothesis on $P \rhd P'$.

CASE 2 $PQ \rhd P'Q$ the thesis follows from the induction hypothesis on $P \rhd P'$. The same for $QP \rhd QP'$, $P\&Q \rhd P'\&Q$ and $Q\&P \rhd Q\&P'$.

CASE 3 $(P_1\&P_2)Q \rhd P_iQ$ then

$$\begin{aligned} M[x := N] &\equiv (P_1[x := N]\&P_2[x := N])Q[x := N] \\ &\equiv (P_1[x := N]\&P_2[x := N])Q \qquad \text{since } Q \text{ is closed} \end{aligned}$$

Since substitutions do not change the type in $(P_1\&P_2)$ (just recall that the type is fixed on the & and does not change during computation) then the selected branch will be the same for both $(P_1\&P_2)Q$ and $(P_1[x := N]\&P_2[x := N])Q$, thus:

$$\begin{aligned} &\rhd P_i[x := N]Q \\ &\equiv P_i[x := N]Q[x := N] \qquad \text{since } Q \text{ is closed} \\ &\equiv M'[x := N] \end{aligned}$$

□

The next lemma shows that reductions are not context-sensitive: given a context $C[\ ]$, i.e. a lambda term with a hole, a reduction inside the hole is not affected by the context. This lemma will allow us to reduce the number of the cases in the next theorem:

Lemma 2.4.4 *Let* R *denote either* β *or* $\beta_\&$*; then for all contexts* $\mathcal{C}[\ ]$ *if* $M \rhd^*_R N$ *then* $\mathcal{C}[M] \rhd^*_R \mathcal{C}[N]$

Proof. The proof is a simple induction on the context $\mathcal{C}[\,]$ □

Theorem 2.4.5 (Weak commutativity) *If $M \rhd_{\beta} N_1$ and $M \rhd_{\beta_\&} N_2$ then there exists N_3 such that $N_1 \rhd^*_{\beta_\&} N_3$ and $N_2 \rhd^*_{\beta} N_3$*

Proof. We proceed by induction on the structure of M. Since M is not in normal form, then $M \not\equiv x$ and $M \not\equiv \varepsilon$. In every induction step we will omit the (sub)cases which are a straightforward consequence of lemma 2.4.4:

1. $M \equiv \lambda x.P$. This case follows from lemma 2.4.4 and induction.
2. $M \equiv (M_1 \& M_2)$ then the only subcase which is not resolved by straightforward use of lemma 2.4.4 is $N_1 \equiv (M_1 \& M_2')$ and $N_2 \equiv (M_1' \& M_2)$ or symmetrically. But then $N_3 \equiv (M_1' \& M_2')$.
3. $M \equiv M_1 \bullet M_2$
   Subcase 1: $N_1 \equiv M_1 \bullet M_2'$ and $N_2 \equiv M_1' \bullet M_2$ or symmetrically. Thus $N_3 \equiv M_1' \bullet M_2'$
   The remaining cases are when $M_1 \equiv (P \& Q)$ and M_2 is closed and in normal form. Then we can have:
   Subcase 2: $N_1 \equiv (P' \& Q) M_2$ and $N_2 \equiv P M_2$ but then $N_3 \equiv P' M_2$
   Subcase 3: $N_1 \equiv (P \& Q') M_2$ and $N_2 \equiv Q M_2$ but then $N_3 \equiv Q' M_2$
   Subcase 4: $N_1 \equiv (P \& Q') M_2$ and $N_2 \equiv P M_2$ but then $N_3 \equiv N_2$
   Subcase 5: $N_1 \equiv (P' \& Q) M_2$ and $N_2 \equiv Q M_2$ but then $N_3 \equiv N_2$
   Note that in the last four cases we have used the property that the type of an &-term doesn't change when we reduce inside it and therefore the selected branch will be the same for the same argument.
4. $M \equiv M_1 \cdot M_2$ then as in the previous case we have:
   Subcase 1: $N_1 \equiv M_1 M_2'$ and $N_2 \equiv M_1' M_2$ or symmetrically. Thus $N_3 \equiv M_1' M_2'$
   The other cases are when M_1 is of the form $\lambda x.P$. Then we can have:
   Subcase 2: $N_1 \equiv P[x := M_2]$ and $N_2 \equiv (\lambda x.P) M_2'$ But $N_1 \rhd^*_{\beta_\&} P[x := M_2']$ (by lemma 2.4.2) and $N_2 \rhd_{\beta} P[x := M_2']$. Thus $N_3 \equiv P[x := M_2']$.
   Subcase 3: $N_1 \equiv P[x := M_2]$ and $N_2 \equiv (\lambda x.P') M_2$ But $N_1 \rhd^*_{\beta_\&} P'[x := M_2]$ (by lemma 2.4.3) and $N_2 \rhd_{\beta} P'[x := M_2]$. Thus $N_3 \equiv P'[x := M_2]$

□

Corollary 2.4.6 $\rhd^*_{\beta_\&}$ *commutes with* $\rhd^*_{\beta}$

Proof. By lemma 3.3.6 in [Bar84]. □

Finally, by applying the Hindley-Rosen lemma, we obtain that the calculus is **CR**.

2.5 Basic encodings

In this calculus it is possible to encode powerful type constructors such as surjective pairings and various calculi of record values.

Definition 2.5.1 A type T is *isolated* if for every type S, $S \leq T$ or $T \leq S$ implies $S = T$ □

2.5.1 Surjective pairings

Surjective pairings (SP) can be encoded in λ& by defining two isolated atomic types P_1 and P_2 together with two constants $\pi_1 : P_1$ and $\pi_2 : P_2$.

$$(T_1 \times T_2) \quad \equiv \quad \{P_1 \to T_1, P_2 \to T_2\}$$

$$\begin{array}{lcl} \pi_i(M) & \equiv & M \bullet \pi_i \\ <M_1, M_2> & \equiv & (\varepsilon \& \lambda x^{P_1}.M_1 \& \lambda x^{P_2}.M_2) \qquad (\text{for } x^{P_i} \notin FV(M_i)) \end{array}$$

It is easy to verify that the subtyping rule for $\times$

$$\frac{S_1 \leq T_1 \qquad S_2 \leq T_2}{S_1 \times S_2 \leq T_1 \times T_2}$$

is the special case of the subtyping rule for overloaded types and the particular encoding[9].

Similarly, one also obtains the typing and reduction rules for SP.

2.5.2 Simple records

In various approaches to object-oriented programming records play an important role. In particular, current functional treatments of object-oriented features formalize objects directly as records (see section 2.6.1). In λ&, records can be encoded in a straightforward way.

Let $L_1, L_2, \ldots$ be an infinite list of atomic types. Assume that they are isolated, and introduce for each L_i a *constant* $\ell_i : L_i$. It is now possible to encode record types, record values and record selection, respectively, as follows:

$$\langle\!\langle \ell_1 : V_1; \ldots; \ell_n : V_n \rangle\!\rangle \quad \equiv \quad \{L_1 \to V_1, \ldots, L_n \to V_n\}$$

$$\begin{array}{lcl} \langle \ell_1 = M_1; \ldots; \ell_n = M_n \rangle & \equiv & (\varepsilon \ \& \ \lambda x^{L_1}.M_1 \ \& \ldots \& \ \lambda x^{L_n}.M_n) \qquad (x^{L_i} \notin FV(M_i)) \\ M.\ell & \equiv & M \bullet \ell \end{array}$$

Since $L_1 \ldots L_n$ are isolated, then the subtyping rule for records is a special case of the rule for overloaded types:

$$\frac{V_1 \leq U_1 \ \ldots V_k \leq U_k}{\langle\!\langle \ell_1 : V_1; \ldots; \ell_k : V_k; \ldots; \ell_{k+j} : V_{k+j} \rangle\!\rangle \leq \langle\!\langle \ell_1 : U_1; \ldots; \ell_k : U_k \rangle\!\rangle}$$

Similarly the type-checking rules special cases for [{}INTRO] and [{}ELIM]

$$[\langle\!\langle\rangle\!\rangle\text{INTRO}] \qquad \frac{\vdash M_1 : V_1 \ldots \vdash M_n : V_n}{\vdash \langle \ell_1 = M_1; \ldots; \ell_n = M_n \rangle : \langle\!\langle \ell_1 : V_1; \ldots; \ell_n : V_n \rangle\!\rangle}$$

$$[\langle\!\langle\rangle\!\rangle\text{ELIM}] \qquad \frac{\vdash M : \langle\!\langle \ell_1 : V_1; \ldots; \ell_n : V_n \rangle\!\rangle}{\vdash M.\ell_i : V_i}$$

[9]Being the *special case* of a rule is stronger than being a *derived* rule. The former signifies that the encodings of two types are in subtyping relation if and only if the types are in subtyping relation; the latter means implies only the "if" part (e.g. if we encode tuples by $\{P_1 \to T_1 \ldots P_n \to T_n\}$ then the subtyping rule is only a derived rule since the encoding of, say, $T_1 \times T_2 \times T_3$ is a subtype of the encoding of $T_1 \times T_2$)

Finally, the rewriting rules (ρ) and (r-ctx) below are the special cases of ($\beta_\&$) and (context) respectively.

ρ) $\langle \ell_1 = M_1; \ldots ; \ell_n = M_n\rangle.\ell_i \;\triangleright M_i \quad (0 \leq i \leq n)$

r-ctx) If $M \;\triangleright M'$ then $M.\ell \;\triangleright M'.\ell$ and $\langle \ldots \ell = M \ldots\rangle \;\triangleright \langle \ldots \ell = M' \ldots\rangle$

2.5.3 Updatable records

There are various definitions for updatable records in the literature. We will meet some of them in this thesis.

The updatable records defined in [Wan87] are constructed starting from an empty record value, denoted by $\langle\,\rangle$, and by two elementary operations:

- *Overwriting* $\langle r \leftarrow \ell_i = M\rangle$; if ℓ_i is not present in r, then it adds a field of label ℓ_i and value M to the record r; otherwise replaces the value of the field with label ℓ_i by the value M.

- *Extraction* $r.\ell_i$; extracts the value corresponding to the label ℓ_i, provided that a field having that label is present.

The record types are encoded in the same way as for simple records. To encode record values it is very useful to introduce the following meta-notations

Notation 2.5.2 *Let* $T \equiv \{S_i \to T_i\}_{i\in I}$ *be an overloaded type. We denote by* $T \smallsetminus S_j$ *the type* $T \equiv \{S_i \to T_i\}_{i\in I \smallsetminus \{j\}}$ *if* $S_j \in \{S_i\}_{i\in I}$ *the type* T *itself, otherwise. We denote by* $T \cup \{S \to T\}$ *the appending of the branch* $S \to T$ *to the list of branches of* T*.*

Thus for example $\{S_1 \to T_1\} \cup \{S_2 \to T\} = \{S_1 \to T, S_2 \to T\}$; also $\{S_1 \to T, S_2 \to T\} \smallsetminus S_1 = \{S_2 \to T\}$ and $\{S_1 \to T, S_2 \to T\} \smallsetminus S = \{S_1 \to T, S_2 \to T\}$ (for $S \neq S_1, S_2$). Note that even if T is a well-formed type $T \smallsetminus S$ may be not well-formed since S might be necessary to assure that T satisfies the condition of multiple inheritance. Though when we restrain our attention to overloaded types whose input types form a set of isolated types this problem does not persist since the condition is always trivially satisfied[10]. Therefore we know that applying the meta-notation $\smallsetminus$ to an overloaded type that encodes a record type always yields a well-formed type. Finally note that if $T \smallsetminus S$ is a well-formed type then $T \leq T \smallsetminus S$.

$$
\begin{array}{lcll}
\langle\,\rangle & = & \varepsilon & \\
r.\ell_i & = & r \bullet \ell_i & \\
\langle r \leftarrow \ell_i = M\rangle & = & (r \;\&^I \lambda x^{L_i}.M) & \text{where } I \equiv (S \smallsetminus L_i) \cup \{L_i \to T\} \\
 & & & \text{if } r\!:S \text{ and } M\!:T
\end{array}
$$

The idea is that when we overwrite in a record a field already present, we erase in the index its old reference, say $L \to T_1$ (we can do it since $T \leq T \smallsetminus L$), and we append to the index the new one $L \to T_2$. In this case T_2 does not have to be related to T_1.

[10] In section 4.1.1 we modify the typing rules of $\lambda\&$ in order to apply this technique also to non isolated types

Note that both the conditions in *Overwriting* and *Extraction* are enforced statically by the encoding: for example if $M_i\colon T_i$ then the record

$$\langle\, \langle\, \langle\rangle \leftarrow \ell = M_1 \rangle \leftarrow \ell = M_2 \rangle$$

is encoded by

$$(\varepsilon \;\&^{\{L\to T_1\}} \lambda x^L.M_1 \;\&^{\{L\to T_2\}} \lambda x^L.M_2)$$

This term has the expected type:

$$[\{\}\textsc{Intro}]\quad \frac{\vdash (\varepsilon \;\&^{\{L\to T_1\}} \lambda x^L.M_1)\colon \{L \to T_1\} \leq \{\} \qquad \vdash \lambda x^L.M_2\colon \{L \to T_2\}}{\vdash ((\varepsilon \;\&^{\{L\to T_1\}} \lambda x^L.M_1) \;\&^{\{L\to T_2\}} \lambda x^L.M_2)\colon \{L \to T_2\} (\equiv \langle\!\langle \ell\colon T_2 \rangle\!\rangle)}$$

Note the use of the index to hide the old branches.

Even if we mimic the calculus on the values of [Wan87], we have not the same power for the types: indeed these encodings lack all the powerful polymorphism of Wand's records. In particular we have not a unique operation "$\leftarrow$" of updating that applies to every record type, but a class of different updating operators one for each record type. The operation $\leftarrow$ is strictly tied to the type of the record it updates, since the index used to encode it fixes the type once forever. Consider for example a variable $x\colon \langle\!\langle \ell\colon S \rangle\!\rangle$. If $M\colon T$ then $\langle x \leftarrow \ell = M \rangle$ is the term $(x\&^{\{L\to T\}}\lambda y.M)$. Consider a term $r\colon \langle\!\langle \ell\colon S, \ell'\colon S' \rangle\!\rangle$ and the substitution $(\langle x \leftarrow \ell = M \rangle)[x\colon= r]$; this yields $(r\&^{\{L\to T\}}\lambda y.M)$. But note that the record we obtained has type $\langle\!\langle \ell\colon T \rangle\!\rangle$ rather than $\langle\!\langle \ell\colon T, \ell'\colon S' \rangle\!\rangle$. This is so because that particular updating where designed for the type $\langle\!\langle \ell\colon S \rangle\!\rangle$. Indeed this is an important lack in view of the modeling of the code reuse of inheritance; this problem has been the motivation of the works of Wand [Wan87, Wan88, Wan91], Rémy [Rém89, Rém90], Cardelli and Mitchell [CM91].

However at this stage, we do not tackle this problem since it can be framed into the more general problem of loss of information [Car88], which is the subject of the second part of this thesis.

2.6 λ& and object-oriented programming

In this section we investigate more in detail the relation between λ& and object-oriented programming. From the previous chapter it should be clear that we represent class-names as types, and methods as overloaded functions that, depending on the type (class-name) of their argument (the object the message is sent to), execute a certain code.

There are many techniques to represent the internal state of objects in this overloading-based approach to object-oriented programming. Since this is not the main concern at this point of the thesis (we will broadly discuss it in section 5.2), we follow a rather primitive technique: we suppose that a program (λ&-term) may be preceded by a declaration of *class types*: a *class type* is an atomic type, which is associated with a unique *representation type*, which is a record type. Two class types are in subtyping relation if this relation has been explicitly declared and it is *feasible*, in the sense that the respective representation types are in subtyping relation too. In other words class types play the role of the atomic types from which we start up, but in addition we can select fields from a value in a class type as if it belonged to its representation record type, and we have an operation $_^{classType}$ to transform

a record value $r\colon R$ into a class type value $r^{classType}$ of type *classType*, provided that the representation type of *classType* is R. We use *italics* to distinguish class types from the usual types, and $\doteq$ to declare a class type and to give it a name; we will use $\equiv$ to associate a name with a value (e.g. with a function). We use the examples of chapter 1 pages 69 and 72, which we adapt to the purposes of this section.

We first declare the following class types:

$$\begin{array}{l} \mathit{2DPoint} \doteq \langle\!\langle x : \mathrm{Int}; y : \mathrm{Int} \rangle\!\rangle \\ \mathit{2DColorPoint} \doteq \langle\!\langle x : \mathrm{Int}; y : \mathrm{Int}; c : \mathrm{String} \rangle\!\rangle \end{array}$$

and impose that on the types *2DColorPoint* and *2DPoint* we have the following relation $\mathit{2DColorPoint} \leq \mathit{2DPoint}$ (which is feasible since it respects the ordering of the record types these class types are associated with); note that this corresponds to having used in example 1.1.4 the keyword `is` in the definition of the class *2DColorPoint*. The method *norm* will be implemented by an overloaded function with just one branch:

$$\mathit{norm} \equiv (\varepsilon \;\&\; \lambda \mathit{self}^{\,\mathit{2DPoint}}.\sqrt{\mathit{self.x}^2 + \mathit{self.y}^2})$$

whose type is $\{\mathit{2DPoint} \to \mathrm{Real}\}$.

This function accepts also arguments of type *2DColorPoint*, since $\mathit{2DColorPoint} \leq \mathit{2DPoint}$. Let us now carry on with our example and have a look at what the restrictions in the formation of the types (section 2.2.2) become in this context.

The first condition, i.e. covariance inside overloaded types, expresses the fact that a version of a method which receives a more informative input returns a more informative output. Suppose that we have redefined in *2DColorPoint* the method *erase* so that it also sets to *white* the color field. Then *erase* is the following overloaded function:

$$\begin{array}{ll} \mathit{erase} \equiv (& \lambda \mathit{self}^{\,\mathit{2DPoint}}.\langle x = 0; y = \mathit{self.y} \rangle^{\mathit{2DPoint}} \\ & \&\; \lambda \mathit{self}^{\,\mathit{2DColorPoint}}.\langle x = 0; y = \mathit{self.y}; c = \text{“white”} \rangle^{\mathit{2DColorPoint}} \\ &) \end{array}$$

whose type is $\{\mathit{2DPoint} \to \mathit{2DPoint}, \mathit{2DColorPoint} \to \mathit{2DColorPoint}\}$. Here covariance arises quite naturally. In object-oriented jargon, covariance says that an overriding method must return a type smaller than the one returned by the overridden one.

As for the second restriction it simply says that in case of multiple inheritance the methods which appear in different ancestors not related by $\leq$, must be explicitly redefined. For example take the alternative definition of the class for colored points given in the example 1.1.5 page 73:

$$\begin{array}{l} \mathit{Color} \doteq \langle\!\langle c : \mathrm{String} \rangle\!\rangle \\ \mathit{2DColorPoint} \doteq \langle\!\langle x : \mathrm{Int}; y : \mathrm{Int}; c : \mathrm{String} \rangle\!\rangle \end{array}$$

then the ordering on the newly defined atomic types is extended in the following (feasible) way: $\mathit{2DColorPoint} \leq \mathit{Color}$ and $\mathit{2DColorPoint} \leq \mathit{2DPoint}$.

Now suppose that in the definition of the class *Color* we have defined a method for *erase*, too. Then the following definition for *erase* would not be legal, as the formation rule 3.c in Section 2.2.2 is violated:

$$
\begin{aligned}
\mathit{erase} \equiv (\ & \lambda \mathit{self}^{\,2DPoint}.\langle x = 0; y = \mathit{self}.y\rangle^{2DPoint} \\
& \&\ \lambda \mathit{self}^{\,Color}.\langle c = \text{“white”}\rangle^{Color} \\
)\ &
\end{aligned}
$$

In object-oriented terms, this happens since *2DColorPoint*, as a subtype of both *2DPoint* and *Color*, inherits the *erase* method from both classes. Since there is no reason to choose one of the two methods and no general way of defining a notion of "merging" for inherited methods, we ask that this multiply inherited method is explicitly redefined for *2DColorPoint*.

In our approach, a correct definition of the *erase* method would be:

$$
\begin{aligned}
\mathit{erase} \equiv (\ & \lambda \mathit{self}^{\,2DPoint}.\langle x = 0; y = \mathit{self}.y\rangle^{2DPoint} \\
& \&\ \lambda \mathit{self}^{\,Color}.\langle c = \text{“white”}\rangle^{Color} \\
& \&\ \lambda \mathit{self}^{\,2DColorPoint}.\langle x = 0; y = \mathit{self}.y; c = \text{“white”}\rangle^{2DColorPoint} \\
)\ &
\end{aligned}
$$

which has type:

$$
\begin{aligned}
\{\ & \mathit{2DPoint} \to \mathit{2DPoint}, \\
& \mathit{Color} \to \mathit{Color}, \\
& \mathit{2DColorPoint} \to \mathit{2DColorPoint}\ \ \}
\end{aligned}
$$

Before showing how inheritance, multi-methods and multiple dispatching are modeled in λ&, we want to recall the model based on the "objects as records" analogy, in order to start the comparison between this model and the one we are studying in this thesis.

2.6.1 The "objects as records" analogy

One of the earliest and most clear functional approaches to objects has been suggested in [Car88] and developed by several authors. The basic idea of that paper was inspired by the implementation of Simula where objects are essentially records with possibly functional components; these functional fields represent the methods of the object and message passing corresponds to the selection of those fields; the remaining fields are to form the "internal" (quotation marks are mandatory) state of the object. In short, this modeling is built around the so-called "object as record analogy" and the main concepts surveyed in chapter 1 are given a precise formal status as follows:

Objects	⇒	Record values
Classes	⇒	Record generators
Methods	⇒	Record fields
Messages	⇒	Record Labels
Message Passing	⇒	Field Selection
Inheritance	⇒	Record extension

To this end Cardelli [Car88] defined $\lambda_{\leq}$, an extension of simple typed lambda calculus by record, variant and recursive types and recursive terms; to type check terms he introduced the subsumption rule (see section 2.2.4). He then proved that the system so obtained prevented from run-time errors: well-typed terms rewrote only into well-typed terms. Furthermore

well-typing was statically decidable, i.e. there existed a type-checking algorithm that assured, at compile time, the absence of type errors during the execution. Record types were needed for objects, recursive types for methods that modify the internal state (e.g. `erase` in example 1.1.3) and recursive terms for `self`.

Consider our key example. In the record-based model a possible formalization of the classes `2DPoint` and `2DColorPoint` would be the following one:

$$2DPoint \equiv \langle\!\langle norm: Real; erase\colon 2DPoint; move\colon (Int \times Int) \to 2DPoint \rangle\!\rangle$$

$$2DColorPoint \equiv$$
$$\langle\!\langle norm: Real; erase\colon 2DColorPoint; move\colon (Int{\times}Int) \to 2DColorPoint; isWhite\colon Bool \rangle\!\rangle$$

Note that these types have recursive definitions. To integrate the property of encapsulation of the internal state a first solution is to code instance variables as local variables. For example in [CCH+89] the command `new` becomes a function that takes as argument the initial values of the instance variables and returns a record of methods; so that `new(2DPoint)` is implemented by the following function

$$\begin{array}{l} \mathsf{Y}(\lambda self. \\ \quad \lambda(x,y). \\ \qquad \langle norm = sqrt(x^2+y^2); \\ \qquad\ erase = self\,(0,y); \\ \qquad\ move = \lambda(p,q).self\,(x+p, y+q) \\ \qquad \rangle) \end{array}$$

to which we pass the initial state of the object (Y is a fix-point operator).

Another solution that is often present in the literature is to consider classes as a sort of abstract data types and then, following the results in [MP85], to model it by an existential type: the variable existentially quantified represents the (hidden) type of the instance variables. This idea is at the base of the approaches in [Cas90b, Cas90a], [Bru92] and [PT93] although they are very different one from the other.

Inheritance in the "objects as record" analogy

Inheritance is the ability to define the state, interface and methods of a class "by difference" with respect to another class; inheritance on methods is the most important one. In the record based model, inheritance is realized using the record concatenation operation to add to the record of the methods of a superclass the new methods defined in the subclass. However, the recursive nature of the hidden *self* parameter forces one to distinguish between the "generator" associated with a class definition, which is essentially a version of the methods where *self* is a visible parameter, from the finished method set, obtained by a fix point operation which transforms *self* into a recursive pointer to the object which the methods belong to. This operation is called "generator wrapping". In the example above the generator of the class *2DPoint* is

$$\begin{array}{l} G_{2DPoint} = \lambda(x,y). \\ \qquad\qquad \langle norm = sqrt(x^2+y^2); \\ \qquad\qquad\ erase = self\,(0,y); \end{array}$$

$$\begin{array}{l} \quad move = \lambda(p,q).self\ (x+p, y+q) \\ \rangle \end{array}$$

and the wrapping corresponds to $\mathsf{Y}(\lambda self.G_{2DPoint})$. Inheritance may be defined by record concatenation over generators.[11]

To be able to reuse a generator, the type of *self* parameter must not be fixed: it must be a type variable that will assume as value the type for which the generator is reused. A first approach is to consider the type of *self* as a parameter itself; let us call it *Mytype* (this is the name used in [Bru92]). In this case, if this "recursive type" appears in the result type of some method, then, when a generator is wrapped, the same operation must be performed on the type, to bind *Mytype* to the type of the class under definition, hence we need a fix point operator at the type level too. If, furthermore, there is some binary method[12], then *Mytype* must be linked to the type of the class under definition on the left hand side of arrows too.

But, if a generator G has such a binary method, and a generator G' is obtained by extending G, then the type obtained by wrapping G' is not a subtype of the one obtained by wrapping G, as explained in more detail in the next section. Hence, subtyping cannot be used to write functions operating on objects corresponding to both G and G', but F-bounded polymorphism must be introduced. F-bounded polymorphism is essentially a way of quantifying over all types obtained by wrapping an extension of a generator F. Thus it permits to define functions that accept as argument values of all the types that may have inherited from a certain type [13]. For an account of this approach see for example [CCH+89, CHC90, Mit90a, Bru91].

The feeling is that in the approach outlined above, recursion is too heavily used. An approach close to the previous one but that avoids the use of recursive types has been recently proposed in [PT93]. The idea is to separate the state of an object from its methods and then encapsulate the whole object by existentially quantifying over the type of the state. The type of a method that works on the internal state does not need to refer to the type of the whole object (as in the previous approach) but only to its state part; therefore recursive types are no longer needed. The type of the state is referred by a type variable since it is the abstract type of the existential quantification. The whole existential type is passed to the generator as in the previous case but without any use of recursive types. Finally, the behavior of F-bounded polymorphism is obtained by a clever use of higher order quantification.

Inheritance in $\lambda\&$

Our approach to method inheritance is even simpler since we also separate the state from the methods[14]. In our system, every subtype of a type inherits all the methods of its supertypes,

[11] We must remark that the generator based approach may account for the special identifier *super* used in object-oriented languages to refer to a method as it is implemented in a superclass, while we do not have this possibility in $\lambda\&$.

[12] A method is *binary* when it has a parameter whose type is the type of the receiver (of the message associated to that method). For some examples see section 2.6.2

[13] Recall that subtyping is sufficient but not necessary for inheritance. The F-bounded quantification is an example of inheritance not obtained from subtyping: see also [Bru91]

[14] Of course we pay this simplicity; for example by a minor encapsulation of the state (it is possible to add new methods that read and write it), and the absence of encapsulation of the methods (methods are no longer

since every overloaded function may be applied to every subtype of the types which the function has been explicitly written for. Moreover, the behavior of an inherited method M appearing as a branch of an overloaded function (i.e. a message) N can be overridden (i.e. defined in a way which is specific for a subtype T) by defining a branch for T inside the overloaded function N. Finally, new methods may be defined for a subtype by defining new overloaded functions. By this we can say that, in our system, *inheritance is given by subtyping plus the branch selection rule*. This can be better seen by an example: suppose to have a message for which a method has been defined in the classes $U_1 \dots U_n$; thus this message denotes an overloaded function of type $\{U_i \to T_i\}_{i=1..n}$ for some T_i's. When this overloaded function is applied to an argument of type U (i.e. the message is sent to an object of class U), the *selected branch* is the one defined for the class $\min_{i=1..n}\{U_i | U \leq U_i\}$. If this minimum is exactly U, this means that the receiver uses the method that has been defined in its class; otherwise, i.e. if this minimum is strictly greater, then the receiver uses the method that its class, U, has *inherited* from this minimum (a superclass); in other terms, the code written for the class which resulted to be the minimum, is *reused* by the objects of the class U.

2.6.2 Binary methods and multiple dispatch

Let us go back to the formalization of object-oriented programming by λ&: we tackle the problem of modeling binary methods. We introduce this problem by showing what happens in the "objects as records" analogy: if we add a method *equal* to *2DPoint* and *2DColorPoint* then, in the notation typical of formalisms built around this analogy, we obtain the following recursive record types (we forget the other methods):

$$2DEPoint \equiv \langle\langle x : \text{Int}; y : \text{Int}; equal : 2DEPoint \to \text{Bool}\rangle\rangle$$
$$2DColEPoint \equiv \langle\langle x : \text{Int}; y : \text{Int}; c : \text{String}; equal : 2DColEPoint \to \text{Bool}\rangle\rangle.$$

The two types are not comparable because of the contravariance of the arrow type in *equal*: since one would expect $2DEPoint$ to be larger, as a record, than $2DColEPoint$, the type at the left of the outer arrow in $2DEPoint$ should be larger, which is impossible by contravariance.[15] Note that this should not be considered a flaw in the system but a desirable property, since a subtyping relation between the two types, in the record based approach, could cause a run-time type error: for example define

$$\begin{array}{lcl} f & \equiv & \lambda p^{2DEPoint}.\lambda q^{2DEPoint}.(q.equal)(p) \\ a & \equiv & \langle x = 3; y = 5; equal = \lambda p^{2DEPoint}.(p.x = 3) \text{ AND } (p.y = 5)\rangle \\ b & \equiv & \langle x = 5; y = 6; c = \text{`}white\text{`}; equal = \lambda p^{2DColEPoint}.(p.c = \text{`}white\text{`})\rangle \end{array}$$

If $2DColEPoint \leq 2DEPoint$ then fab would be well-typed; but the reader can easily verify that this would generate a type-error, since the function would try to select the field c in a.

Hence, there is an apparent incompatibility between the covariant nature of most binary operations and the contravariant subtyping rule of arrow types.

---

encapsulated inside the object; this is a fundamental drawback in wide-area distributed systems where you want objects to navigate carrying their operation with them).

[15] Recursive types should be considered as denotations for their infinite expansion, and an infinite type is a subtype of another one when all the finite approximations of the first one are subtypes of the corresponding finite approximation of the second one; see [AC90].

Our system is essentially more flexible, in this case. Indeed if we set $\mathit{2DColorPoint} \leq \mathit{2DPoint}$ then an equality function, with type:

$$equal: \{\mathit{2DPoint} \rightarrow (\mathit{2DPoint} \rightarrow \mathrm{Bool}),\ \ \mathit{2DColorPoint} \rightarrow (\mathit{2DColorPoint} \rightarrow \mathrm{Bool})\}$$

would not be well-typed in our system either, since $\mathit{2DColorPoint} \leq \mathit{2DPoint}$ while $\mathit{2DPoint} \rightarrow \mathit{Bool} \leq \mathit{2DColorPoint} \rightarrow \mathit{Bool}$. This expresses the fact that a comparison function cannot be chosen only on the basis of the type of the first argument. In our system, on the other hand, we can write an equality function where the code is chosen on the basis of both arguments

$$\begin{aligned} equal \equiv (\ & \lambda(p,q)^{2DPoint \times 2DPoint}.(p.x = q.x) \text{ AND } (p.y = q.y) \\ & \&\ \lambda(p,q)^{2DColorPoint \times 2DColorPoint}.(p.x = q.x) \text{ AND } (p.y = q.y) \text{ AND } (p.c = q.c) \\ )& \end{aligned}$$

the function above has type:

$$\{(\mathit{2DPoint} \times \mathit{2DPoint}) \rightarrow \mathrm{Bool}\ ,\ \ (\mathit{2DColorPoint} \times \mathit{2DColorPoint}) \rightarrow \mathrm{Bool}\}$$

which is well-formed[16].

Thus part of the expressive power of our system derives from the ability to choose one implementation on the basis of the types of many arguments. This ability makes it even possible to decide explicitly how to implement "mixed binary operations". For example, besides implementing "pure" equality between *2DPoint*s and between *2DColorPoint*s, we can also decide how we should compare a *2DPoint* and a *2DColorPoint*, as below[17] :

$$\begin{aligned} equal \equiv (\ & \lambda(p,q)^{2DPoint \times 2DPoint}.\quad \ldots \\ & \&\ \lambda(p,q)^{2DColorPoint \times 2DColorPoint}.\quad \ldots \\ & \&\ \lambda(p,q)^{2DPoint \times 2DColorPoint}.q.c = \text{“}white''\\ & \&\ \lambda(p,q)^{2DColorPoint \times 2DPoint}.(p.x = q.x) \text{ AND } (p.y = q.y) \text{ AND } (p.c = \text{“}white'') \\ )& \end{aligned}$$

The ability to choose a method on the basis of several object parameters is called, in object-oriented jargon, *multiple dispatch*.

2.6.3 Covariance vs. contravariance

In the presence of a subtyping relation, the covariance versus contravariance of the arrow type, w.r.t. the left argument (domain), is a delicate and classical debate. Semantically (categorically) oriented people have no doubt: the hom-functor is contravariant in the first argument. Moreover, this nicely fits with typed models constructed over type-free universes, where types are subsets or subrelations of the type-free structure, and type-free terms model runtime computations. Also the common sense of type-checking forces contravariance: we consider one type a subtype of another if and only if all expressions of the former type can be

[16] This is not surprising as, even if the types of the two versions of *equal* are componentwise isomorphic, in general isomorphisms of types do not preserve subtyping. This is true also for the simply typed lambda calculus: if $A < B$ then $A \times A \rightarrow A \leq A \times B \rightarrow A$ but $A \rightarrow A \rightarrow A \not\leq A \rightarrow B \rightarrow A$

[17] Match the definition of *equal* with the method for `compare` in section 1.1.9

used in the place of expressions of the latter; then a function $g : T \to U$ may be substituted by a function f only if the domain of f is *greater* than T. However, practitioners often have a different attitude. In OOP, in particular, the "overriding" of a method by one, say, with a smaller domain (input type) leads to a smaller codomain (output type), in the spirit of a "preservation of information". Indeed, in our approach, we show that both viewpoints are correct, when adopted in the "right" context.

In fact, our general arrow types (the types of ordinary functions) are contravariant in the first argument, as required by common sense and mathematical meaning. However, the *families* of arrow types which are glued together in overloaded types form covariant collections, by our conditions on the formation of these types (see 2.2.2). Besides the justification of this at the end of section 2.1.1, consider the practice of overriding as shown in chapter 1, section 1.1.5. The implementation of a method in a superclass is substituted by a more specific implementation in a subclass; or, more precisely, overriding methods must return smaller or equal types than the overridden one. For example, the "+" operation, on different types, may be given by two different implementations: one implementation of type $Int \times Int \to Int$, the other of type $Real \times Real \to Real$. In our approach, we can glue these implementations together into one global method, precisely because their types satisfy the required covariance condition. We broadly discuss this issue in section 11.2.

2.6.4 Abstract classes

We now briefly discuss the mechanism of abstract classes[18] which have been omitted in the toy language of chapter 1. An abstract class is a class that can be used only as the base for the definition of some other classes. If a class is abstract it cannot be used as argument of `new`; thus an abstract class has no instance. A class is *abstract* if it associates to one of its message a *virtual* method (*deferred* method in Eiffel terminology). A method is *virtual* when it does not implement any operation but it defers its definition to the subclasses. Take again the example of chess in section 1.1.5: in that example we had defined a fake class `Chessman`, which were the superclass of all the classes that implemented the chessmen, i.e. `King`, `Queen`, `Bishop` and so on. The reason of its definition was twofold: on one hand it allowed to share the code of the methods whose implementation was the same for all the chessmen (as for example the methods `position` or `capture`); on the other hand it permitted to write functions working on all chessmen: just define a function of the form

$$\texttt{fn}(\texttt{x} : \texttt{Chessman}) => \ldots \tag{2.25}$$

and by subtyping it will accept as argument any object instance of the classes `King`, `Queen`, etc. In that same example we also said that a method for a message like `move` for which it is not possible to give a general implementation, would have been defined in each subclass. Note then that

1. In the program, instances of the class `Chessman` will never be used: all the objects will be an instance of some subclass of `Chessman` (you never play with generic chessmen but with two kings, two queens, four bishops...).

[18] *Abstract class* is the name used in C++ [Str84, ES90] and in Dylan [App92] to denote this mechanism; in Eiffel [Mey88] the name used is *deferred class*

2. Every object of a subclass of `Chessmen` can respond to the message `move`.

From this one deduces that all the objects that will be passed to a function like (2.25) will be able to respond to the message `move`. Nevertheless it is not possible in the body of (2.25) to send `move` to `x`. This because there is not syntactical construction asserting that the two properties above are satisfied.

This can be obtained by declaring in the definition of `Chessman` that the implementation of `move` is virtual, and thus it is deferred to the subclasses. A possible syntax is

```
class Chessman
  {
   color: String;
   x: Int;
   y: Int;
       :
  }
  position = (self.x, self.y)
  move = virtual
       :
  [[
    position = Int x Int
    move = Int x Int -> Chessman
       :
  ]]
```

This implies that `Chessman` is an abstract class and thus it will have no instances, and that all the objects instance of a subclass of `Chessman` will be able to respond to the message `move` (if a subclass of `Chessman` does not define a method for `move` then it will be abstract, too).

In λ& an abstract class C is simply an atomic type for which the function $_^C$ (see page 102) is not defined. In this way it is not possible to create objects of class C. A message like `move` which possesses a virtual branch is an overloaded function in which we can put as a virtual branch any expression of the right type: we know that this branch will be never selected. For example we can add to λ& a constant *virtual* which has every type and define move in the following way

$$
\begin{array}{ll}
(& \varepsilon \\
\& & \lambda self^{Chessman}.virtual \\
\& & \lambda self^{King}.\lambda(x,y)^{Int\times Int}.\ \mathrm{IF}\ abs(x-self.x)\leq 1\ \mathrm{AND}\ abs(y-self.y)\leq 1\ \mathrm{THEN}\ \ldots \\
 & \qquad\qquad \vdots \\
) &
\end{array}
$$

Note that the virtual branch will be never selected: it cannot be selected for an object of class *Chessman* because there exists no such an object (*Chessman* is virtual); it cannot be selected for an object of a C subclass of *Chessman* because if such an object exists then C is not virtual and thus a non virtual method for *move* has been defined for C.

In some languages a further distinction is introduced between abstract and partially abstract classes: a class is (totally) *abstract* when all its methods (defined or inherited) are

virtual. A class is *partially abstract* if some of its methods are virtual but not all of them. Thus abstract classes are used just to correlate various classes in order to define functions that accept objects of all these classes, while a partially abstract class is needed as soon as these classes have also to share some code. For example `Chessman` above is a partially abstract class since it has a virtual method `move` and a non virtual method `position` whose code is thus shared by all the subclasses of `Chessman`.

This distinction is introduced in the formalisms that use the objects as records analogy where inheritance is obtained by extension of the generators (see section 2.6.1). In these formalisms it is very easy to have totally abstract classes: they correspond to record types for which no generator has been defined. But it is not completely clear how to model partially abstract classes: indeed to implement these classes one would have to define generators with undefined fields, and to assure that these generators will never be *wrapped*, since a partially abstract class cannot have any instance. Note that on the contrary in our model abstract and partially abstract classes are dealt with in the same (natural) way: a class C is abstract if in all the overloaded functions possessing a branch selected by the input type C that branch is virtual; it is partially abstract if there exist two overloaded functions possessing a branch selected by the input type C such that in one the branch is virtual and in the other not.

Chapter 3

Strong Normalization

Interestingly, according to modern astronomers, space is finite. This is a very comforting thought — particularly for people who can never remember where they have left things.

Woody Allen
Side-effects. (1981)

In this chapter we study the normalization properties of $\lambda\&$. We show that the $\lambda\&$-calculus is not strongly normalizing and that it is possible to define in it a fix-point combinator of type $(T \to T) \to T$ for every well formed type T. This expressiveness derives from the definition of the subtyping relation for overloaded types. We give a sufficient condition to have strong normalization, and we define two expressive systems that satisfy it. These systems are important since they will be used in chapter 6 to study the mathematical meaning of overloading and because they are expressive enough to model object-oriented programming. This chapter is based on a joint work with Giorgio Ghelli and Giuseppe Longo.

3.1 The full calculus is not normalizing

The $\lambda\&$ calculus is not normalizing. Let T be any type; consider the following term, where $\emptyset$ is used instead of $\{\}$ to reduce the parenthesis nesting level, and where $\mathcal{E}_T$ stands for any closed term of type $\{\emptyset \to T\}$, e.g. $\mathcal{E}_T \equiv (\varepsilon \;\&^{\{\emptyset\to T\}}\; \lambda x^{\emptyset}.M)$ with M of type T:

$$\begin{array}{rcl} \omega_T & = & (\mathcal{E}_T \;\&^{\{\emptyset\to T,\{\emptyset\to T\}\to T\}}\; (\lambda x^{\{\emptyset\to T\}}.x \bullet x)) \colon \mathcal{W}_T \\ \mathcal{W}_T & = & \{\emptyset \to T, \{\emptyset \to T\} \to T\} \end{array}$$

ω_T is a $\lambda\&$ version of the untyped λ-term $\omega \equiv \lambda x.xx$, coerced to a type $\mathcal{W}_T$ such that it is possible to apply ω_T to itself. ω_T is well typed; in particular, $x \bullet x$ is well typed and has type T as proved below:

$$[\{\}\text{Elim}] \qquad \frac{\vdash x \colon \{\emptyset \to T\} \quad \vdash x \colon \{\emptyset \to T\} \quad \emptyset = \min_{U \in \{\emptyset\}}\{U \mid \{\emptyset \to T\} \leq U\}}{\vdash x \bullet x \colon T}$$

The term ω_T has the peculiar characteristic that its self application is well-typed and it does not possess a normal form. Define $\Omega_T \equiv \omega_T \bullet \omega_T$. Let first show that Ω_T has type T:

$$[\{\}\text{ELIM}] \quad \frac{\vdash \omega_T : \mathcal{W}_T \quad \vdash \omega_T : \mathcal{W}_T \qquad \{\emptyset \to T\} = \min_{U \in \{\emptyset, \{\emptyset \to T\}\}} \{U | \{\emptyset \to T, \{\emptyset \to T\} \to T\} \leq U\}}{\vdash \omega_T \bullet \omega_T : T}$$

Now we can show that Ω_T is not strongly normalizing as it reduces to itself:

$$\Omega_T \equiv \omega_T \bullet \omega_T \quad \rhd_{\beta_\&} \quad (\lambda x^{\{\emptyset \to \emptyset\}}.x \bullet x) \cdot \omega_T \quad \rhd_\beta \quad \omega_T \bullet \omega_T \equiv \Omega_T$$

More than that, Ω_T has no normal form, since the one used above is the only possible reduction strategy (there may be some reductions in $\mathcal{E}_T$ but they cannot affect the outer reduction).

3.2 Fixed point combinators

The presence of fixed point combinators in a calculus is very important since it allows recursive definitions of functions. In this section we show that there are infinitely many fixed point combinators in $\lambda\&$.

Definition 3.2.1 [Bar84] A *fixed point combinator* is a combinator[1] M such that for all terms N $MN = N(MN)$ $\Box$

In lambda calculus the classical fixed point combinator is Curry's paradoxical combinator $\mathbf{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$. We can follow the technique used for the definition of ω_T to define the typed term equivalent to $\mathbf{Y}$:

$$\mathbf{Y}_T \equiv \lambda f^{T \to T}.((\mathcal{E}_T \&^{\{\emptyset \to T, \{\emptyset \to T\} \to T\}} \lambda x^{\{\emptyset \to T\}}.f(x \bullet x)) \bullet (\mathcal{E}_T \&^{\{\emptyset \to T, \{\emptyset \to T\} \to T\}} \lambda x^{\{\emptyset \to T\}}.f(x \bullet x)))$$

First of all note that $\mathbf{Y}_T : (T \to T) \to T$. Indeed by the same derivations as above we prove that $(x \bullet x) : T$ and thus $f(x \bullet x) : T$. Then we also have that $\mathbf{Y}_T$ is a *fixed point combinator*: indeed let F be a term of type $T \to T$, and define $H \equiv (\mathcal{E}_T \&^{\{\emptyset \to T, \{\emptyset \to T\} \to T\}} \lambda x^{\{\emptyset \to T\}}.F(x \bullet x))$. Then

$$\mathbf{Y}_T F \rhd H \bullet H \rhd^* F(H \bullet H) \lhd F(\mathbf{Y}_T F)$$

Note also that

$$\Omega_T \equiv \omega_T \bullet \omega_T = \mathbf{Y}_T(\lambda x^T.x)$$

as it happens in classical lambda-calculus.

[1] A *combinator* is a closed term not containing constants. Here we make an exception to it defining a combinator as a closed term not containing constants other than ε, since every closed term with an overloaded type must contain it.

We can carry on with this technique to mimic the Turing's fixed point combinator Θ_T for every T [Tur37]. Recall that the Turing's fixed point combinator has the following characteristic:

$$\Theta F \triangleright^* F(\Theta F)$$

which is not true for $\mathbf{Y}$.

Define $\mathcal{S} \equiv (T \to T) \to T$. Then

$$A_T \equiv (\mathcal{E}_{\mathcal{S}} \&^{\{\emptyset \to \mathcal{S}, \{\emptyset \to \mathcal{S}\} \to \mathcal{S}\}} \lambda x^{\{\emptyset \to \mathcal{S}\}}.\lambda y^{T \to T}.y((x \bullet x)y)$$

has type $\{\emptyset \to \mathcal{S}, \{\emptyset \to \mathcal{S}\} \to \mathcal{S}\}$. Finally define $\Theta_T \equiv A_T \bullet A_T : \mathcal{S}$. Then for $F{:}T \to T$ we obtain

$$\Theta_T F \equiv (A \bullet A)F \triangleright^* F((A \bullet A)F) \equiv F(\Theta_T F)$$

Again Θ_T is well typed: indeed $(x \bullet x)$ has type $\mathcal{S}$, i.e. $(T \to T) \to T$; since $y{:}T \to T$ then $y((x \bullet x)y){:}T$. And thus Θ_T has type $\mathcal{S}$, i.e. $(T \to T) \to T$ as expected. This combiantor will be used in section 5.4

3.3 The reasons for non normalization

Simply typed lambda calculus prevents looping, essentially, by imposing a stratification between a function of type $T \to U$ and its argument, whose type T is "simpler" than the whole type $T \to U$; the same thing happens, in a subtler way, with system F.

When we add subtyping, the type T' of the argument of a function with type $T \to U$ is just a subtype of T, and may be, syntactically, much bigger than the whole $T \to U$: consider the case when T' is a record type with more fields that T. However, the *rank* of T' is still strictly smaller than that of $T \to U$, where the rank of an arrow type is at least the rank of its domain part plus one (for a correct definition see below; however as a first approximation define the rank of a type as the depth of its syntax tree). This happens, in short, since in $\lambda_{\leq}$ and in $F_{\leq}$ two types can be related by subtyping only when they have the same rank (or one of the two is atomic). Hence, $\lambda_{\leq}$ and $F_{\leq}$ are strongly normalizing [Ghe90].

$\lambda\&$ typing does not prevent looping, essentially, since it allows one to compare types with a different rank. In our example, we pass a parameter of type $\{\emptyset \to T, \{\emptyset \to T\} \to T\}$ (rank 2) to a function with domain type $\{\emptyset \to \emptyset\}$ (rank 1), and in the $x \bullet x$ case we pass a parameter of type $\{\emptyset \to \emptyset\}$ (rank 1) to a function with domain type $\{\}$ (rank 0). Hence, $\lambda\&$ typing does not prevent looping since it does not stratify functions w.r.t. their arguments.

However, when $\lambda\&$ is used to model object-oriented programming, it is always used in a stratified way. It is then interesting to define a stratified subsystem of $\lambda\&$ which is both strongly normalizing and expressive enough to model object-oriented programming. To this aim, we will prove the following theorem.

Theorem 3.3.1 *Let* $\lambda\&^-$ *be any subsystem of* $\lambda\&$ *closed by reduction and let* rank *be any function associating integers with* $\lambda\&^-$ *types. Assume also that, if* T *(syntactically) occurs in* U*, then* $\mathrm{rank}(T) \leq \mathrm{rank}(U)$*. If in* $\lambda\&^-$*, for any well typed application* $M^T N^U$ *one has* $\mathrm{rank}(U) < \mathrm{rank}(T)$*, then* $\lambda\&^-$ *is Strongly Normalizing.*

Example 3.3.2 We may obtain a system similar to $\lambda\&$ and with the properties of $\lambda\&^{-}$ in 3.3.1 either by restricting the set of types, or by imposing a stricter subtyping relation. We propose here two significant examples based on these restrictions: $(\lambda\&^{-}_{\mathrm{T}})$ and $(\lambda\&^{-}_{\leq})$, respectively. In either case, the *rank* function is defined as follows:

$$
\begin{array}{lcl}
rank(\{\}) & = & 0 \\
rank(A) & = & 0 \\
rank(T \to U) & = & \max\{rank(T)+1, rank(U)\} \\
rank(\{T_i \to U_i\}_{i \in I}) & = & \max_{i\in I}\{rank(T_i \to U_i)\}
\end{array}
$$

Note however that other definitions of *rank* are possible.

The idea is that, by restricting the set of types or the subtyping relation as described below, the types of a function and of its arguments are "stratified", namely the rank of the functional type is strictly greater than the rank of the input type, as required by theorem 3.3.1.

- $\lambda\&^{-}_{\leq}$ is defined by substituting $\leq$ in all $\lambda\&$ rules with a stricter subtyping relation $\leq^{-}$ defined by adding to any subtyping rule which proves $T \leq U$ the further condition $rank(T) \leq rank(U)$. In any well typed $\lambda\&^{-}_{\leq}$ application $M^{\{T_i \to U_i\}_{i\in I}} \bullet N^{T'}$, the rank of T' is then smaller than the rank of some T_i, hence is strictly smaller than the rank of $\{T_i \to U_i\}_{i\in I}$; similarly for functional application. The subject reduction proof for $\lambda\&$ works for $\lambda\&^{-}_{\leq}$ too, thanks to the transitivity of the $\leq^{-}$ relation.[2]

- $\lambda\&^{-}_{\mathrm{T}}$ is defined by imposing, on overloaded types $\{T_i \to U_i\}_{i\in I}$, the restriction that the ranks of all the branch types $T_i \to U_i$ are equal, and by stipulating that $\{\}$ is not a supertype of any non-empty overloaded type (see the previous footnote). Then we can prove inductively that, whenever $T \leq U$, then $rank(T) = rank(U)$, and that $\lambda\&^{-}_{\mathrm{T}}$ is a subsystem of $\lambda\&^{-}_{\leq}$. To prove the closure under reduction (i.e., that $\lambda\&^{-}_{\mathrm{T}}$ terms reduce to $\lambda\&^{-}_{\mathrm{T}}$ terms), observe first that a $\lambda\&$ term is also a $\lambda\&^{-}_{\mathrm{T}}$ term if and only if all the overloaded types appearing in the indexes of variables and of &'s are $\lambda\&^{-}_{\mathrm{T}}$ overloaded types (this is easily shown by induction on typing rules). The closure by reduction follows immediately, since variables and &'s indexes are never created by a reduction step.

Note that $\lambda\&^{-}_{\mathrm{T}}$ is already expressive enough to model object-oriented programming, where all methods always have the same rank (rank 1), and that $\lambda\&^{-}_{\leq}$ is even more expressive than $\lambda\&^{-}_{\mathrm{T}}$.

Note also that $\lambda\&^{-}_{\leq}$ is a subsystem of $\lambda\&$ if and only if the restriction is imposed on the the subtyping relation *on the types* (and not just on the pretypes); thus for example one has to use $\leq$ in the definition of well formed types and $\leq^{-}$ in the type checking rules. In the same way $\lambda\&^{-}_{\mathrm{T}}$ is a subsystem of $\lambda\&$ if and only if the restriction is imposed on the *well formed* overloaded types (not just on pretypes). Indeed if we restrict the subtyping relation on pretypes or we exclude some pretypes it happens that two types that possessed a common

[2] Note that, in this system, $\{\}$ is not a supertype of any non-empty overloaded type; this is not a problem, since the empty overloaded type is only used to type ε, which is only used to start overloaded function construction. However, we may alternatively define a family of empty types $\{\}_{i\in\omega}$, each being the maximum overloaded type of the corresponding rank, and a correspondent family of empty functions $\varepsilon_{i\in\omega}$.

lower bound in the full system may no longer possess it in the restriction. Therefore the condition **(c)** may be more easily satisfied and types that were not well formed may now satisfy all the conditions of good formation. We would have then more types and, thus, more terms. □

Theorem 3.3.1 and the examples show that there exist subsystems of $\lambda\&$ which are strongly normalizing and expressive enough for our purposes. However we preferred to adopt the whole $\lambda\&$ as our target system, since it is easier to establish results such as Subject Reduction and Confluence on the wider system and apply them in subsystems rather than trying to extend restricted versions to more general cases.

In the following subsections we prove Theorem 3.3.1.

3.4 Typed-inductive properties

As is well known, strong normalization cannot be proved by induction on terms, since β reduction potentially increases the size of the reduced term. For this reason we introduce, along the lines of [Mit86], a different notion of induction on typed terms, called *typed induction*, proving that every typed-inductive property is satisfied by any typed $\lambda\&^{-}$ term. This notion is defined to conform the reduction, so that some reduction related properties, such as strong normalization or confluence, can be easily proved to be typed-inductive. Theorem 3.4.7, which proves that every typed-inductive property is satisfied by any typed $\lambda\&^{-}$ term, is the kernel of our proof and is related to the normalization proofs due to Tait, Girard, Mitchell and others. We had to avoid, though, the notions of saturated set and of logical relation, which do not seem to generalize easily to our setting. In this section we define a notion of "typed-inductive property" for $\lambda\&^{-}$ terms and show that every typed-inductive property is satisfied by any (well-typed) $\lambda\&^{-}$ term. Although many of the results and definitions in this section hold or make sense for $\lambda\&$ too, the reader should remember that all the terms, types and judgments in this section refer to a $\lambda\&^{-}$ system satisfying the conditions of Theorem 3.3.1.

Notation 3.4.1 *$M \circ N$ will denote $M \cdot N$ if $M{:}\,T \to U$ and $M{\bullet}N$ if $M{:}\,\{M_i \to N_i\}_{i=1\ldots n}$.*

Notation 3.4.2 *$\vec{M}$ denotes a list $[M_i]_{i=1,\ldots,n}$ of terms, possibly empty, and $N \cdot \vec{M}$ means $N \cdot M_1 \circ \ldots \circ M_n$; the same for $N{\bullet}\vec{M}$; if $\vec{M}$ is empty, $N \circ \vec{M}$ is just N.*

"$\vec{M}$ is well typed" means "each M_i in $\vec{M}$ is well typed"; similarly for other predicates on terms.

Definition 3.4.3 *Let $\{\mathcal{S}^T\}_T$ be a family of sets of $\lambda\&^{-}$ terms, indexed over $\lambda\&^{-}$ types, such that:*

$$M{\in}\mathcal{S}^T \Rightarrow\ \vdash M{:}\,T.$$

$\mathcal{S}$ is typed-inductive *if it satisfies the following conditions*[3] *(where $M{\in}\mathcal{S}^{if}$ means "$M{\in}\mathcal{S}$ if M is well typed"):*

[3] We use $\mathcal{S}$ for $\{\mathcal{S}^T\}_T$. Furthermore, since any term M has a unique type T, we will write without ambiguity $M{\in}\mathcal{S}$ to mean $M{\in}\mathcal{S}^T$.

(x/c) $\forall x, \vec{N}{\in}\mathcal{S}.\ x \circ \vec{N} \in \mathcal{S}^{if}$
and similarly for constants and for ε.

$(\&_1)$ $\forall M_1{\in}\mathcal{S}, M_2{\in}\mathcal{S}, N{\in}\mathcal{S}, \vec{N}{\in}\mathcal{S}.$
$M_1{\bullet}N \circ \vec{N} \in \mathcal{S}^{if} \wedge M_2 \cdot N \circ \vec{N} \in \mathcal{S}^{if} \Rightarrow (M_1\&M_2){\bullet}N \circ \vec{N} \in \mathcal{S}^{if}$

(λ_1) $\forall M{\in}\mathcal{S}, N{\in}\mathcal{S}, \vec{N}{\in}\mathcal{S}.\ M[x{:=}N] \circ \vec{N} \in \mathcal{S}^{if} \Rightarrow (\lambda x{:}T.M) \cdot N \circ \vec{N} \in \mathcal{S}^{if}$

$(\&_2)$ $\forall M_1{\in}\mathcal{S}, M_2{\in}\mathcal{S}.\ M_1\&M_2 \in \mathcal{S}^{if}$

(λ_2) $\forall M{\in}\mathcal{S}.\ \lambda x^T.M \in \mathcal{S}^{if}$

The $\mathcal{S}^{if}$ notation means that all the "$\in\mathcal{S}$" predicates in the above implications must be satisfied only by typed preterms. This is difficult only in case $\&_1$: depending on whether $M_1{\bullet}\ldots$ is well-typed, $M_2 \cdot \ldots$ is well-typed or both are well-typed, the first, the second or both are required to be in $\mathcal{S}$; indeed we want to take into account all the branches that could be selected not only the one that will be actually executed. For this reason we used in $\&_1$ a "$\wedge$" rather than a "$\vee$".

We aim to prove, by induction on terms, that every well-typed $\lambda\&^-$ term N belongs to $\mathcal{S}$. The conditions on typed induction allow an inductive proof of this fact for terms like $\lambda x^T.M$ and $M\&N$, but we have no direct proof that $(M{\in}\mathcal{S}{\wedge}N{\in}\mathcal{S}){\Rightarrow}(M \circ N{\in}\mathcal{S})$. For this reason we derive from $\mathcal{S}$ a stronger predicate $\mathcal{S}^*$ which allows term induction through application. We will then prove that $\mathcal{S}^*$ is not actually stronger than $\mathcal{S}$, since for any typed-inductive property $\mathcal{S}$:

$$M{\in}\mathcal{S}^{*T}{\Leftrightarrow}M{\in}\mathcal{S}^T{\Leftrightarrow}\vdash M{:}T.$$

The definition of $\mathcal{S}^*$ is the only part of the proof where we need the stratification by the rank function.

Notation 3.4.4 ($\widehat{[T_i]}_{i\in I}$) *For any list of types $[T_i]_{i\in I}$, $T'{\in}\widehat{[T_i]}_{i\in I}$ if and only if $\exists i{\in}I.T'{\leq}T_i$. Note that if $\vdash M{:}\{T_i \to U_i\}_{i\in I}$ and $\vdash N{:}T'$ then $M{\bullet}N$ is well typed if and only if $T'{\in}\widehat{[T_i]}_{i\in I}$.*

Definition 3.4.5 *For any typed-inductive property $\{\mathcal{S}^T\}_T$ its application closure on $\lambda\&^-$ terms $\{\mathcal{S}^{*T}\}_T$ is defined, by lexicographic induction on the rank and then on the size of T, as follows:*

$(atomic)$ $M{\in}\mathcal{S}^{*A}{\Leftrightarrow}M{\in}\mathcal{S}^A$

$(\to)$ $M{\in}\mathcal{S}^{*T\to U} \Leftrightarrow M{\in}\mathcal{S}^{T\to U} \wedge \forall T'{\leq}T.\forall N{\in}\mathcal{S}^{*T'}.\ M \cdot N{\in}\mathcal{S}^{*U}$

$(\{\})$ $M{\in}\mathcal{S}^{*\{T_i\to U_i\}_{i=1\ldots n}}$
$\Leftrightarrow M{\in}\mathcal{S}^{\{T_i\to U_i\}_{i=1\ldots n}} \wedge \forall T'{\in}\widehat{[T_i]}_{i=1\ldots n}.\forall N{\in}\mathcal{S}^{*T'}.\exists i{\in}[1..n].M{\bullet}N{\in}\mathcal{S}^{*U_i}$

In short:

$$M{\in}\mathcal{S}^*{\Leftrightarrow}M{\in}\mathcal{S}{\wedge}\forall N{\in}\mathcal{S}^*.M \circ N{\in}\mathcal{S}^{*if}$$

In the definition of $\mathcal{S}^*$, we say that M belongs to $\mathcal{S}^*$ by taking for granted the definition of $\mathcal{S}^*$ over the types of the N's such that $M \circ N$ is well typed and over the type of $M \circ N$ itself. This is consistent with the inductive hypothesis since:

1. The rank of the type of N is strictly smaller than the rank of the type of M in view of the conditions in Theorem 3.3.1.
2. Since the type U of $M \circ N$ strictly occurs in the type W of M, then the rank of U is not greater than the rank of W (by the conditions in Theorem 3.3.1). Hence the definition is well formed either by induction on the rank or, if the ranks of U and W are equal, by secondary induction on the size.

The next lemma shows, informally, that in the condition $M\in\mathcal{S}^*\Leftrightarrow\forall N\in\mathcal{S}^*.M\circ N\in\mathcal{S}^{*if}$ we can trade an $*$ for an $\vec{\ }$, since $\forall N\in\mathcal{S}^*.M\circ N\in\mathcal{S}^{*if}\Leftrightarrow\forall\vec{N}\in\mathcal{S}^*.M\circ\vec{N}\in\mathcal{S}^{if}$.

Lemma 3.4.6 *$M\in\mathcal{S}^*\Leftrightarrow M$ is well typed $\wedge\forall\vec{N}\in\mathcal{S}^*.M\circ\vec{N}\in\mathcal{S}^{if}$*

Proof.

($\Rightarrow$) "M is well typed" is immediate since $M\in\mathcal{S}^{*T}\Rightarrow M\in\mathcal{S}^T\Rightarrow\vdash M{:}T$.

$\forall\vec{N}\in\mathcal{S}^*.M\circ\vec{N}\in\mathcal{S}^{if}$ is proved by proving the stronger property $\forall\vec{N}\in\mathcal{S}^*.M\circ\vec{N}\in\mathcal{S}^{*if}$ by induction on the length of $\vec{N}$. If $\vec{N}$ is empty, the thesis is immediate. If $\vec{N}=N_1\cup\vec{N}'$ then $M\circ N_1\in\mathcal{S}^{*if}$ by definition of $\mathcal{S}^*$, and $(M\circ N_1)\circ\vec{N}'\in\mathcal{S}^{*if}$ by induction.

($\Leftarrow$) By definition, $M\in\mathcal{S}^*\Leftrightarrow M\in\mathcal{S}\wedge\forall N\in\mathcal{S}^*.M\circ N\in\mathcal{S}^{*if}$. $\forall\vec{N}\in\mathcal{S}^*.M\bullet\vec{N}\in\mathcal{S}^{if}$ implies immediately $M\in\mathcal{S}$: just take an empty $\vec{N}$. $M\circ N\in\mathcal{S}^{*if}$ is proved by induction on the type of M.

(atomic) $\vdash M{:}A$: $M\circ N$ is never well typed; $M\in\mathcal{S}^A$ is enough to conclude $M\in\mathcal{S}^{*A}$.

($\{\}$) $\vdash M{:}\{\}$: as above.

($\rightarrow$) $\vdash M{:}T\rightarrow U$: we have to prove that $\forall N\in\mathcal{S}^{*T'},T'\leq T.M\cdot N\in\mathcal{S}^{*U}$. By hypothesis:

$$\forall\vec{N}\in\mathcal{S}^*.M\cdot N\circ\vec{N}\in\mathcal{S}^{if}$$

applying induction to $M\cdot N$, whose type U is smaller than the one of $T\rightarrow U$, we have that $M\cdot N\in\mathcal{S}^{*U}$.

($\{T_i\rightarrow U_i\}$) $\vdash M{:}\{T_i\rightarrow U_i\}_{i=1\ldots n+1}$: as in the previous case.

□

Theorem 3.4.7 *If $\mathcal{S}$ is typed-inductive, then every term $\vdash N{:}T$ is in $\mathcal{S}^{*T}$.*

Proof. We prove the following stronger property: if N is well-typed and $\sigma\equiv[\vec{x}^{\vec{T}}:=\vec{N}]$ is a well-typed $\mathcal{S}^*$-substitution (i.e. for $i\in[1..n]$. $N_i\in\mathcal{S}^{*T'_i}$ and $T'_i\leq T_i$), then $N\sigma\in\mathcal{S}^*$; $\vec{x}^{\vec{T}}$ is called the domain of $\sigma\equiv[\vec{x}^{\vec{T}}:=\vec{N}]$, and is denoted as $dom(\sigma)$.

It is proved by induction on the size of N. In any induction step, we prove $\forall\sigma.N\sigma\in\mathcal{S}^*$, supposing that, for any N' smaller than N, $\forall\sigma'.N'\sigma'\in\mathcal{S}^*$ (which implies $N'\sigma'\in\mathcal{S}$ and $N'\in\mathcal{S}$).

(c) $c\sigma\equiv c$. We apply lemma 3.4.6, and prove that $\forall\vec{N}\in\mathcal{S}^*.c\circ\vec{N}\in\mathcal{S}^{if}$. Since $\vec{N}\in\mathcal{S}^*\Rightarrow\vec{N}\in\mathcal{S}$ then $c\circ\vec{N}\in\mathcal{S}^{if}$ follows immediately from property (c) of $\mathcal{S}$.

(x) If $x{\in}dom(\sigma)$ then $x\sigma{\in}\mathcal{S}^*$ since σ is an $\mathcal{S}^*$-substitution. Otherwise, reason as in case (c).

($M_1\&M_2$) By applying lemma 3.4.6 we prove that $\forall\sigma.\forall\vec{N}{\in}\mathcal{S}^*.(M_1\&M_2)\sigma\circ\vec{N}{\in}\mathcal{S}^{if}$.

We have two cases. If $\vec{N}$ is not empty then $\vec{N}{\equiv}\,N_1\cup\vec{N}'$. For any σ, $M_1\sigma{\bullet}N_1\circ\vec{N}'{\in}\mathcal{S}^{if}$ and $M_2\sigma{\cdot}N_1{\circ}\vec{N}'{\in}\mathcal{S}^{if}$ by induction (M_1 and M_2 are smaller than $M_1\&M_2$). Then $(M_1\&M_2)\sigma{\bullet}N_1\circ\vec{N}'{\in}\mathcal{S}^{if}$ by property ($\&_1$) of $\mathcal{S}$.

If $\vec{N}$ is empty then $(M_1\&M_2)\sigma{\in}\mathcal{S}$ follows, by property ($\&_2$) of $\mathcal{S}$, from the inductive hypothesis $M_1\sigma{\in}\mathcal{S}$ and $M_2\sigma{\in}\mathcal{S}$.

($\lambda x^T.M$) We will prove that $\forall\sigma.\forall\vec{N}{\in}\mathcal{S}^*.\ (\lambda x^T.M)\sigma\circ\vec{N}{\in}\mathcal{S}^{if}$, supposing, w.l.o.g., that x is not in $dom(\sigma)$.

We have two cases. If $\vec{N}$ is not empty and $(\lambda x^T.M)\sigma\circ\vec{N}$ is well typed then $\vec{N}{\equiv}\,N_1\cup\vec{N}'$ and the type of N_1 is a subtype of T. Then for any $\mathcal{S}^*$-substitution σ, $\sigma[x^T:=N_1]$ is a well-typed $\mathcal{S}^*$-substitution, since $N_1{\in}\mathcal{S}^*$ by hypothesis, and then $M(\sigma[x{:=}N_1])\circ\vec{N}'{\in}\mathcal{S}^{if}$ by induction, which implies $(M\sigma)[x{:=}N_1]\circ\vec{N}'{\in}\mathcal{S}^{if}$. Then $(\lambda x^T.M\sigma)\cdot N_1\circ\vec{N}'{\equiv}(\lambda x^T.M)\sigma\circ\vec{N}{\in}\mathcal{S}^{if}$ by property (λ_1) of $\mathcal{S}$.

If $\vec{N}$ is empty, $(\lambda x^T.M)\sigma{\in}\mathcal{S}$ follows, by property (λ_2), from the inductive hypothesis $M\sigma{\in}\mathcal{S}$.

($M\circ N$) By induction $M\sigma{\in}\mathcal{S}^*$ and $N\sigma{\in}\mathcal{S}^*$; then $(M\circ N)\sigma{\in}\mathcal{S}^*$ by definition of $\mathcal{S}^*$.

This property implies the theorem since, as can be argued by case (x) of this proof, the identity substitution is a well-typed $\mathcal{S}^*$-substitution. □

Corollary 3.4.8 *If $\mathcal{S}$ is a typed-inductive property, every well-typed term satisfies $\mathcal{S}$ and its application closure and viceversa:*

$$M{\in}\mathcal{S}^{*T} \;\Leftrightarrow\; M{\in}\mathcal{S}^T \;\Leftrightarrow\; \vdash M{:}T$$

Proof.

$M{\in}\mathcal{S}^{*T}$	$\Rightarrow$	$M{\in}\mathcal{S}^T$	by definition of $\mathcal{S}^*$.
$M{\in}\mathcal{S}^T$	$\Rightarrow$	$\vdash M{:}T$	by definition of typed induction.
$\vdash M{:}T$	$\Rightarrow$	$M{\in}\mathcal{S}^{*T}$	by theorem 3.4.7.

□

3.5 Strong Normalization is typed-inductive

In this section we prove the strong normalization of $\lambda\&^-$ by proving that strong normalization is a typed-inductive property of $\lambda\&^-$ terms.

Consider the following term rewriting system *unconditional*-$\beta_\&\cup\beta$, which differs from $\beta_\&\cup\beta$ since unconditional-$\beta_\&$ reduction steps are allowed even if N is not normal or not closed and the selected branch can be any of those whose input type is compatible with the type of the argument:

β) $(\lambda x^S.M)N\ \triangleright M[x^S{:=}N]$

uncond.-$\beta_{\&}$) If $N{:}U \leq U_j$ then

$$((M_1 \&^{\{U_i \to V_i\}_{i=1..n}} M_2) \bullet N) \rhd \begin{cases} M_1 \bullet N & \text{for } j < n \\ M_2 \cdot N & \text{for } j = n \end{cases}$$

Instead of proving Strong Normalization for $\lambda\&^{-}$ reduction, we prove Strong Normalization for unconditional-$\beta_{\&} \cup \beta$. Since any $\beta_{\&} \cup \beta$ reduction is also an unconditional-$\beta_{\&} \cup \beta$ reduction, Strong Normalization of the unconditional system implies Strong Normalization for the original one. Note that the proof of subject reduction is valid also when using the uncond.-$\beta_{\&}$ reduction (the proof results even simpler), but that even if the $\beta_{\&}$ conditions are not necessary to obtain strong termination or subject reduction, they are still needed to obtain confluence. We prove the strong normalization for this general case since it will be used in section 4.2 to prove the strong normalization for the calculus with explicit coercions.

Notation 3.5.1 *If M is strongly normalizing, $\nu(M)$ is the length of the longest reduction chain starting from M. $\nu(\vec{M})$ is equal to $\nu(M_1) + \ldots + \nu(M_n)$.*

Theorem 3.5.2 *$\mathcal{SN}^T$, the property of being strongly normalizing terms of type T (according to the unconditional relation) is typed-inductive.*

Proof.

(x/c) $\forall \vec{N} \in \mathcal{SN}.\ x^U \circ \vec{N} \in \mathcal{SN}^{if}$

By induction on $\nu(\vec{N})$: if $x \circ \vec{N} \rhd P$ then $P = x \circ N_1' \circ \ldots \circ N_n'$ where just one of the primed terms is a one-step reduct of the corresponding non-primed one, while the other ones are equal. So $P \in \mathcal{SN}$ by induction on $\nu(\vec{N})$.

($\&_1$) $\forall M_1 \in \mathcal{SN}, M_2 \in \mathcal{SN}, N \in \mathcal{SN}, \vec{N} \in \mathcal{SN}.$
$M_1 \bullet N \circ \vec{N} \in \mathcal{SN}^{if} \wedge M_2 \cdot N \circ \vec{N} \in \mathcal{SN}^{if} \Rightarrow (M_1 \& M_2) \bullet N \circ \vec{N} \in \mathcal{SN}^{if}$

By induction on $\nu(M_1) + \nu(M_2) + \nu(N) + \nu(\vec{N})$.

If $(M_1 \& M_2) \bullet N \circ \vec{N} \rhd P$ then we have the following cases:

($\beta\&_1$) $P = M_1 \bullet N \circ \vec{N}$: since P is well-typed by subject-reduction, then $P \in \mathcal{SN}$ by hypothesis.

($\beta\&_2$) $P = M_2 \cdot N \circ \vec{N}$: as above.

(congr.) $P = (M_1' \& M_2') \bullet N' \circ \vec{N}'$: $P \in \mathcal{SN}$ by induction on ν.

So $(M_1 \& M_2) \bullet N \circ \vec{N} \in \mathcal{SN}$ since it one-step reduces only to strongly normalizing terms.

(λ_1) $\forall M \in \mathcal{SN}, N \in \mathcal{SN}, \vec{N} \in \mathcal{SN}.\ M[x{:=}N] \circ \vec{N} \in \mathcal{SN} \Rightarrow (\lambda x^T.M) \cdot N \circ \vec{N} \in \mathcal{SN}^{if}$

By induction on $\nu(M) + \nu(N) + \nu(\vec{N})$. If $(\lambda x^T.M) \cdot N \circ \vec{N} \rhd P$ we have the following cases:

(β) $P = M[x{:=}N] \circ \vec{N}$: $P \in \mathcal{SN}$ by hypothesis.

(congr.) $P = (\lambda x^T.M') \cdot N' \circ \vec{N'}$ where just one of the primed terms is a one-step reduct of the corresponding one, while the other ones are equal: $P{\in}\mathcal{SN}$ by induction on ν.

($\&_2$) $\forall M_1{\in}\mathcal{SN}, M_2{\in}\mathcal{SN}.\ M_1\&M_2 \in \mathcal{SN}^{if}$

By induction on $\nu(M_1) + \nu(M_2)$. If $M_1\&M_2 \rhd P$ then $P{\equiv}M_1'\&M_2'$ where one of the primed terms is a one-step reduct of the corresponding one, while the other one is equal; then $P{\in}\mathcal{SN}$ by induction.

(λ_2) $\forall M{\in}\mathcal{SN}.\vdash \lambda x^T.M{:}T \to U \Rightarrow \lambda x^T.M \in \mathcal{SN}$

If $\lambda x^T.M \rhd \lambda x^T.M'$ then, since $\nu(M') < \nu(M)$, $\lambda x^T.M'{\in}\mathcal{SN}$ by induction on $\nu(M)$. So $\lambda x^T.M{\in}\mathcal{SN}$.

□

The last proof can be easily extended to show that the reduction system remains strongly normalizing if we add the following extensionality rules:

(η) $\quad \lambda x^T.M \cdot x \rhd M \quad$ if x is not free in M

($\eta_\&$) $\quad M\&(\lambda x^T.M{\bullet}x) \rhd M \quad$ if x is not free in M

Theorem 3.3.1 is now a corollary of Theorem 3.5.2 and of Corollary 3.4.8.

Chapter 4

Three variations on the theme

I have called this principle, by which each slight variation, if useful, is preserved, by the term of Natural Selection

CHARLES DARWIN
The Origin of Species (1859)

In this chapter we present three different systems directly issued from $\lambda\&$

1. A modification of $\lambda\&$ with less contrived restrictions on the formation of types and terms, and on the application of the reduction rules.

2. The extension of $\lambda\&$ by the addition of explicit coercions.

3. A calculus in which regular functions and overloaded ones are introduced by the same abstraction operator, which therefore unifies them.

The first two systems are introduced mainly by pragmatical reasons, since they will be extensively used when translating object-oriented languages into the formal system (see chapter 5). The last calculus has mainly a theoretical interest and is somewhat detached from the leitmotiv of the thesis, i.e. the typing of object-oriented languages.

The three modifications are independent one from each other, thus can be introduced separately.

4.1 More freedom to the system: $\lambda\&^+$

We present in this thesis a unique variant of $\lambda\&$, but a very appealing one. We will introduce three modifications in the definition of $\lambda\&$: the first two were suggested when trying to translate object-oriented languages into $\lambda\&$ [Cas93b] (see also chapter 5) and thus they have a very pragmatical reason. The last one has been introduced to give full generality to the implementation of late binding, during the study of a formalism in which a unique abstraction could be used both for overloaded and normal functions; thus it is originally motivated by

theoretical reasons; however, it is extremely important for code optimization during the compilation of the programs; it also serves as an introduction to the calculus in 4.3.

4.1.1 Modifying the good formation of types

Up to now we said that an overloaded type $\{U_i \to T_i\}_{i\in I}$ is well-formed if the set of its input types $\{U_i\}_{i\in I}$ satisfies the "multiple inheritance" condition (see section 2.2.2:

$$\textbf{(c)} \qquad \forall U_i, U_j \in \{U_i\}_{i\in I}.U \Downarrow V \Rightarrow \exists! h \in I \,.\, U_h = \inf\{U_i, U_j\}$$

This restriction is too strong to model an object-oriented language in which the methods that are in common to more than one direct ancestor must be redefined. Consider the following example: take two unrelated classes A and B whose objects can respond to the same message m. Suppose you define a new class C by multiple inheritance from A and B: the method for m must be redefined for C, and the domains of the type of m becomes as in Figure 1. Then define a new class D by multiple inheritance from A and B where you redefine the method for m. Then the domains of the type of m get as in Figure 2. Note that there may not be any ambiguity in the selection of the branch, though this set of input types does not respect the condition **(c)** since there exists no *inf* for A and B.

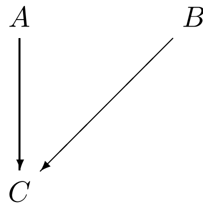


Fig. 1

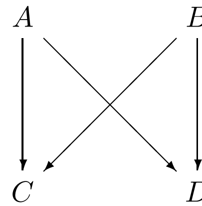


Fig. 2

Indeed the condition of having semi-lattices is very comfortable to work with, especially when dealing with the semantics of the system (see chapter 6). Though it happens to be too strict for the practice of multiple inheritance: as shown by the example above, the user-defined hierarchy may not be a semilattice but just an order.

Thus the first modification we apport is to weaken the **(c)** restriction in order to capture the general behavior of multiple inheritance. Given two types U, V denote by $LB(U, V)$ the set of common lower bounds of U and V. Then replace in the definition of type good formation the condition **(c)** by the following one:

$$\textbf{(c+)} \qquad \forall U_i, U_j \in \{U_i\}_{i\in I} \,.\, \forall U \text{ maximal element of } LB(U_i, U_j)\ \exists! h \in I.\, U_h = U$$ [1]

This condition corresponds to the requirement of redefining the methods in common to some direct ancestors of a class defined by multiple inheritance: when a class is defined by multiple inheritance from, say, two others then it becomes not the inf of the two classes in the subtyping

[1] Once more this definition is not precise since $\leq$ is not a p.o.: indeed we have not to consider the maximal elements of $LB(U, V)$ but rather the equivalences classes of maximal elements: thus the right formulation is: for every U maximal in $LB(U_i, U_j)$ there exists a unique $h \in I$ s.t. $U_h \in [U]_\sim$

hierarchy (this is true only for the first common descendent) but rather a maximal element of the set of the subclasses of these two classes; this because the practice of object-oriented programming does not allow to declare a class *greater* than another: only refinement is allowed. Thus the freshly defined class will always be a maximal element since no other class can be inserted between it and and its ancestors. Therefore condition **(c+)** exactly requires the redefinition of methods common to direct ancestors and ... nothing more.

Recall that we introduced the condition **(c)** in order to assure the existence of the minimum in the selection of the branch. The condition **(c+)** is weaker than **(c)** though it still assures the existence of the minimum in the selection of the branch. Actually the following theorem prove that **(c+)** is the weakest condition to assure it:

Theorem 4.1.1 *Let* $(Y, \leqslant)$ *be a partially ordered set,* $X \subseteq Y$*. Define:*
(1) $\forall a, b \in X. \forall c \in Y. (c \text{ maximal in } LB(a, b) \Rightarrow c \in X)$
(2) $\forall c \in Y. \{x \in X | c \leqslant x\}$ *either is empty or it has a least element*
then (1) $\iff$ (2)

Proof.
Sufficiency ($\Rightarrow$). Suppose that $\mathcal{S} = \{x \in X | c \leqslant x\}$ is not empty, and by contradiction that there exist two distinct minimal elements of $\mathcal{S}$, say, a and b. From the hypothesis it follows that there is no maximal element of $LB(a, b)$ greater than or equal to c otherwise it would be contained in X and thus in $\mathcal{S}$ contradicting the minimality of a and b. Contradiction since $c \in LB(a, b)$.

Necessity ($\Leftarrow$) Let c be a maximal element of $LB(a, b)$ with $a, b \in X$. Then $\{x \in X | c \leqslant x\}$ is not empty since it contains at least a and b. By hypothesis $\min_{x \in X}\{x | c \leqslant x\}$ exists and belongs both to X and to $LB(a, b)$. By definition one has $c \leqslant \min_{x \in X}\{x | c \leqslant x\}$; by the maximality of c in $LB(a, b)$ one also has $\min_{x \in X}\{x | c \leqslant x\} \leqslant c$. By antisymmetry $c = \min_{x \in X}\{x | c \leqslant x\} \in X$. □

If you take Y equal to **Types**, "$\leqslant$" equal to "$\leq$" and X equal to the domain of an overloaded type, then you obtain the desired result[2], i.e. that in a well-typed overloaded application there always exists a least compatible branch to select *if and only if* the input types of the overloaded function satisfy the condition **(c+)**.

4.1.2 Modifying the formation of the terms

The second modification that we want to make to $\lambda\&$ is to allow the replacement of a branch in an overloaded function. This is an operation that is allowed in some object-oriented languages, where it corresponds to the redefinition of a certain method (see section 1.1.7).

In our calculus what we would like to obtain is that when we append to an overloaded function of type $\{U_i \to V_i\}_{i \in I}$ a branch of type $U \to V$ if $U \in \{U_i\}_{i \in I}$ then the new branch

[2] For $\leqslant$ preorder modify the theorem in the following way:
Theorem 4.1.2 *Let* $(Y, \leqslant)$ *be a preordered set,* $X \subseteq Y$*. Define:*
(1) $\forall a, b \in X. \forall c \in Y. ([c]_\sim \text{ maximal in } LB(a, b) \Rightarrow \exists!\, d \in [c]_\sim . d \in X)$
(2) $\forall c \in Y. \{a \in X | c \leqslant a\}$ *either is empty of it a a least element*
then (1) $\iff$ (2)
The proof of theorem 4.1.1 can be easily adapted to this

replaces the old one whose input type was U. We already showed how to do it in section 2.5.3 to encode updatable records. However that technique does not work in general but only for overloaded types with isolated input types. For the general case it can be obtained by a slight modification of the [{}INTRO] rule. To make it more readable it is useful to introduce the following meta-notation

Notation 4.1.3
$\{U_1 \to V_1, \ldots, U_n \to V_n\} \oplus U \to V =$

$$= \begin{cases} \{U_1 \to V_1, \ldots, U_{i-1} \to V_{i-1}, U_{i+1} \to V_{i+1}, \ldots U_n \to V_n, U \to V\} & \textit{if } U = U_i \\ \{U_1 \to V_1, \ldots, \ldots U_n \to V_n, U \to V\} & \textit{otherwise} \end{cases}$$

Roughly speaking the meta-notation "$\oplus$" denotes the append of a new arrow type to an overloaded type and eventually remove from it an existing arrow type with the same input type[3] (note that by the uniqueness of the input types there may be at most one such a branch). The new formulation of the [{}INTRO] rule is

$$[\{\}\textsc{Intro}+] \qquad \frac{\vdash M\colon W_1 \leq \{U_i \to V_i\}_{i\in I} \quad \vdash N\colon W_2 \leq U \to V}{\vdash (M \&^{\{U_i \to V_i\}_{i\in I} \oplus (U \to V)} N)\colon \{U_i \to V_i\}_{i\in I} \oplus (U \to V)}$$

If M does not contain any branch with input type U then this rule behaves as the old rule. Otherwise the old branch remains there but will be never selected again; note that in that case, contrary to what happens in the old rule, the overloaded type at the premises and the one at the conclusion have the same cardinality.

Note that this effect can be obtained also with the old rule

$$[\{\}\textsc{Intro}] \qquad \frac{\vdash M\colon W_1 \leq \{U_i \to V_i\}_{i\leq(n-1)} \quad \vdash N\colon W_2 \leq U_n \to V_n}{\vdash (M \&^{\{U_i \to V_i\}_{i\leq n}} N)\colon \{U_i \to V_i\}_{i\leq n}}$$

if we don't require $\{U_i \to V_i\}_{i\leq(n-1)}$ to be a type but just a pretype.

Finally and most important, remark that neither of these modifications affect any of the results of $\lambda\&$. It is indeed easy to verify that the very same proofs of the theorems of subject-reduction, Church-Rosser and strong normalization for $\lambda\&$ hold also if we replace [{}INTRO] by [{}INTRO+].

4.1.3 Modifying the notion of reduction

In subsection 4.1.1 we defined the weakest condition that assured the existence of a least branch for the selection. We would like to do the same with late binding and find out the weakest condition that assures the correct implementation of late binding. Unfortunately one cannot do it: we recall that with late binding we refer to the mechanism of selection of the branch according to the most precise type of the argument. Thus, to put it in another way, we would like to execute the branch with the least input type compatible with (i.e. greater than) the run-time type of the argument. Now a reduction rule such as

[3] $\{U_i \to V_i\}_{i\in I} \oplus (U \to V)$ corresponds to $(\{U_i \to V_i\}_{i\in I} \setminus U) \cup \{U \to V\}$ of section 2.5.3

$$\beta_{\&}) \quad ((M_1\&^{\{U_i\to V_i\}_{i=1..n}}M_2)\bullet N \rhd \begin{cases} M_1\bullet N & \text{for } j<n \\ M_2\cdot N & \text{for } j=n \end{cases} \quad \begin{matrix} U_j \text{ least type compatible} \\ \text{with the run-time type of } N \end{matrix}$$

is surely intractable if not even undecidable. In general it will be necessary at least to compute a good deal of the program this redex appears in, in order to discover the right U_j. In $\lambda\&$ we adopted the simplest solution choosing to allow the reduction only after that this computation had taken place, when the argument had reached its run-time type. This solution was inspired by what happens in object-oriented programming in which a message is bound to a method only when the receiver of the message is a fully evaluated object. Though there are some reasonable improvements. For example one can always safely perform the reduction when the involved overloaded function has only one branch, or when the type of the argument is a leaf of the type hierarchy and thus cannot decrease any further.

We think that a good trade-off between the tractability of the reduction and its generality is to allow reductions also when we are sure that however the computation evolves the selected branch is always the same. This is precisely stated by the following notion of reduction which in $\lambda\&^{+}$replaces ($\beta_{\&}$):

$\beta^{+}_{\&}$) Let $U_j = min\{U_i | U \leq U_i\}$; if $N{:}U$ is closed and in normal form or $\{U_i | U_i \leq U_j\} = \{U_j\}$ then

$$((M_1\&^{\{U_i\to V_i\}_{i=1..n}}M_2)\bullet N) \rhd \begin{cases} M_1\bullet N & \text{for } j<n \\ M_2\cdot N & \text{for } j=n \end{cases}$$

In other terms when we select one branch we check whether there are other branches with a smaller input type. If not, we know that the selected branch can no longer change, and thus we can reduce.

This variant is interesting from a proof theoretical point of view since, as we will show in section 4.3, it permits us to consider lambda abstractions as the special case of overloaded functions with just one branch. However it has not only a theoretical interest. At run-time this rule would never be used; indeed one does not want to early select or partially evaluate methods but rather to apply them to concrete objects. On the contrary at compile time a rule that permits the early resolution of the dispatching is essential for the production of efficient object code. A preliminary study on early implementations of Dylan showed that on non optimized code about 30% of the time of computation is spent to perform the method dispatching (source: Dylan group, Eastern Research and Technology, personal communication). It is then clear that a mechanism which permits us to solve the dispatching (branch selection) at compile time is one of the main tasks in designing a compiler producing code comparable for speed to the code produced by a C++ compiler. ($\beta^{+}_{\&}$) goes in that direction. We think that this rule and a definition of values that captures most of the terms whose type can no longer decrease (see the "tagged values" of next chapter) are the basic mechanisms that allow a significative amount of resolution at compile time of method dispatching and, thus, the production of efficient object code.

In conclusion the $\lambda\&^{+}$-calculus is obtained from $\lambda\&$ by replacing **(c)**, [{}INTRO] and ($\beta_{\&}$) by **(c+)**, [{}INTRO+] and ($\beta^{+}_{\&}$) respectively. We now pass to study the properties of $\lambda\&^{+}$.

4.1.4 Conservativity

The first observation is that $\lambda\&^+$ is a conservative extension of $\lambda\&$ w.r.t. the subtyping relation. Strictly speaking, let **Types** and **Types**$^+$ denote the set of well-formed types of $\lambda\&$ and $\lambda\&^+$ respectively; let $\mathfrak{L} \equiv \mathbf{Types} \times \mathbf{Types}$ and $\mathfrak{L}^+ \equiv \mathbf{Types}^+ \times \mathbf{Types}^+$, and denote by $\mathcal{T}$ and $\mathcal{T}^+$ the the subtyping relations (as set of pairs) of $\lambda\&$ and $\lambda\&^+$, obtained from a same axiom set for subtyping relation on atomic types. Then we have to show that $\mathcal{T}^+ \cap \mathfrak{L} = \mathcal{T}$. In other words, we have to prove that two types in **Types** are in the subtyping relation of $\lambda\&$ if and only if they are in subtyping relation of $\lambda\&^+$. This is a consequence of the following theorem:

Theorem 4.1.4 (transitivity elimination) *If $\lambda\&^+ \vdash S \leq T$ then there exists a derivation of this judgement that do not use the transitivity rule.*

Proof. The proof of the theorem for $\lambda\&$ (theorem 2.2.1) works also in this case. □

Corollary 4.1.5 (conservativity) *If $S, T \in$ **Types** then*

$$\lambda\& \vdash S \leq T \qquad \Longleftrightarrow \qquad \lambda\&^+ \vdash S \leq T$$

Proof. A trivial induction on the subtyping rules (without transitivity of course). In other terms if the theorem would not hold, this could only happen if and only if we had used the transitivity rule on three types $U \leq V \leq W$ with U and W types of $\lambda\&$ and V type of $\lambda\&^+$but not of $\lambda\&$. The transitivity elimination ensures that this is not possible. □

Observe that if one just replaces **(c)** by **(c+)** the proofs of subject reduction, Church-Rosser, and strong normalization for $\lambda\&$ are still valid: indeed in these proofs there is no assumption on the good formation of types; only the existence of a least feasible input type in the overloaded application must be assured. Thus theorem 4.1.1 assures the validity of all these results.

We already said that these same proofs work also with [{}Intro+]. Therefore to prove these theorems for the whole $\lambda\&^+$ it just suffices to show how to modify the proofs written for $\lambda\&$ to take into account $(\beta_\&^+)$. This is the subject of the next three subsections.

4.1.5 Subject Reduction

Lemma 4.1.6 (Substitution Lemma) *Let $M{:}U$, $N{:}T'$ and $T' \leq T$. Then $M[x^T{:=}N]{:}U'$, where $U' \leq U$.*

Proof. The same as lemma 2.3.1. □

Theorem 4.1.7 (Generalized Subject Reduction) *Let $M{:}U$. If $M \; \triangleright^* N$ then $N{:}U'$, where $U' \leq U$.*

Proof. In the case $M \equiv (N_1 \& N_2) \bullet M_2$ of theorem 2.3.2 the argument M_2 may not be in normal form. The rest of the proof is unchanged □

4.1.6 Church-Rosser

Lemma 4.1.8 *If* $N \triangleright^*_{\beta^+_\&} N'$ *and* $N{:}T'{\leq}T$ *then* $M[x^T := N] \triangleright^*_{\beta^+_\&} M[x^T := N']$

Proof. The same as lemma 2.4.2 □

Lemma 4.1.9 *If* $M \triangleright_{\beta^+_\&} M'$ *and* $N{:}T'{\leq}T$ *then* $M[x^T := N] \triangleright_{\beta^+_\&} M'[x^T := N]$

Proof. We have to add the following case to the proof of lemma 2.4.3

CASE 4 $(P_1\&^S P_2)Q \triangleright P_j Q$ where Q is not closed and in normal form. Then

$$M[x := N] \equiv (P_1[x := N]\&P_2[x := N])Q[x := N]$$

Let U be the type of Q, $\{U_i\}_{i\in I}$ be the set of input types of S and $U_j = \min_{i\in I}\{U_i \mid U \leq U_i\}$. Since Q is not closed and in normal form then by the definition of $(\beta^+_\&)$ we have that $\{U_i \mid U_i \leq U_j\} = \{U_j\}$. By lemma 4.1.6 one has $Q[x := N]{:}U' \leq U$. This implies that $\min_{i\in I}\{U_i \mid U' \leq U_i\} \leq \min_{i\in I}\{U_i \mid U \leq U_i\}$. But since $\{U_i \mid U_i \leq U_j\} = \{U_j\}$ then $\min_{i\in I}\{U_i \mid U' \leq U_i\} = U_j$. Whereas substitutions do not change the type of $(P_1\&^S P_2)$ (recall that it is fixed by the index), the selected branch will be the same for both $(P_1\&P_2)Q$ and $(P_1[x := N]\&P_2[x := N])Q$, thus:

$$\begin{aligned} M[x := N] &\triangleright P_j[x := N](Q[x := N)) \\ &\equiv M'[x := N] \end{aligned}$$

□

Theorem 4.1.10 (Weak commutativity) *If* $M \triangleright_\beta N_1$ *and* $M \triangleright_{\beta^+_\&} N_2$ *then there exists* N_3 *such that* $N_1 \triangleright^*_{\beta^+_\&} N_3$ *and* $N_2 \triangleright^*_\beta N_3$

Proof. As the proof of theorem 2.4.5. The only modifications are to be done in case 3 which, since M_2 may be not closed and in normal form, needs two more subcases:
Subcase 6: $N_1{\equiv}(P\&^S Q)M_2'$ and $N_2{\equiv}P{\bullet}M_2$ Thus M_2 is not in normal form. By the definition of $(\beta^+_\&)$ then the input type of the selected branch is a minimal element of the set of input types of S. By the subject reduction theorem 4.1.7, the type of M_2' is smaller than or equal to the type of M_2. Thus, by the minimality, the selected branch we be the same both for $(P\&^S Q)M_2$ and for $(P\&^S Q)M_2'$. Therefore $N_3{\equiv}PM_2'$.
Subcase 7: $N_1{\equiv}(P'\&^S Q)M_2'$ and $N_2{\equiv}QM_2$: as the case above. □

4.1.7 Strong Normalization

Finally it is very easy to check that all the results stated in the chapter 3 on strong normalization hold for $\lambda\&^+$, too. Indeed, as we already said, the definition of good type formation and the rule [{}INTRO+] do not play any role in the proof of strong normalization. Furthermore the proof of strong normalization in section 3.5 is done for the notion of reduction *uncond*-$\beta_\&$ which is weaker than $(\beta^+_\&)$. Therefore that proof implies the strong normalization of the restrictions of $\lambda\&^+$.

4.2 Adding explicit coercions

One of the most interesting extensions of $\lambda\&$ (or of $\lambda\&^{+}$: see the remark 4.2.4) is the one obtained by adding explicit coercions. An explicit coercion informally is a term that changes the type of its argument into a super-type. For example, the term $\mathbf{coerce}^{T}(M)$ behaves in the same way as the term M but with type T. The precise meaning of "behaving in the same way" is given by the following notion of reduction[4]:

$$\text{(coerce)} \qquad \mathbf{coerce}^{T}(M) \circ N \ \triangleright M \circ N$$

In order to preserve the subject reduction property we must require T to be a supertype of the type of M. More formally we add to the $\lambda\&$-terms the following term

$$\mathbf{coerce}^{T}(M)$$

we add to the typing rules the following rule

$$[\text{COERCE}] \qquad \frac{\vdash M{:}\,S \leq T}{\vdash \mathbf{coerce}^{T}(M){:}\,T}$$

and we add (coerce) to the notions of reduction.

The interest of this extension is that in $\lambda\&$ the computation depends directly on types. Since the coercions affect the types then they also affect the computation. More precisely it is possible to drive the selection in an overloaded application on a particular branch by applying a coercion to the argument. This is for example what we will do later in the thesis to deal with the "early binding" (see section 6.3). In object-oriented programming coercions corresponds to the constructs that change the class of an object: For example in Dylan the command **as** $< C > M$ corresponds to our $\mathbf{coerce}^{C}(M)$; though in Dylan this operation is not type safe because it does not require C to be a supertype of the class of M. Of similar behavior is `change-class` in CLOS. There however the operation is type safe since when one changes the class of an object M to the class C, the instance variables specified in C that are not in M are initialized in the coerced term (as a limit if all the instance variables of M are different from those specified in C then `change-class` corresponds to the creation of a new instance).

Note that a coerced term keeps all its functionalities, since the coercion disappears as soon as we have to "use" the term (i.e. to apply it). Instead the coercion is maintained as long as the term is an argument of a function, since in that case it is its type that matters.

The rule (coerce) is the simplest rule we have add to the system to have expressive coercions. Others rules could be added for example

$$\mathbf{coerce}^{T}(\mathbf{coerce}^{V}(M)) \ \triangleright \mathbf{coerce}^{T}(M)$$

A different set of rules, stemming from the semantics of the coercions are proposed in [BL90]. However all of them would not bring any interesting modification to the system so we prefer to limit our study to this minimal extension.

[4] As usual we use $\circ$ to denote either $\cdot$ or $\bullet$. Strictly speaking (coerce) is the union of two different notions of reduction, one for functional application and the other for the overloaded one.

This extension does not introduce any modification at the level of types; therefore the property of transitivity elimination still holds. Another thing that is of immediate verification is that this is a conservative extension of the theory of the terms of $\lambda\&$. Indeed it is trivial to check that if M and N are two terms without coercions then $\lambda\& \vdash M \triangleright N$ if and only if $\lambda\& + coerce \vdash M \triangleright N$. Let us verify also the other properties.

4.2.1 Subject Reduction

It is very easy to prove the subject reduction theorem for this extension: it suffice to apport some slight modifications to the proof in section 2.3. More precisely:

Lemma 4.2.1 (Substitution Lemma) *Let $M{:}U$, $N{:}T'$ and $T' \leq T$. Then $M[x^T{:=}N]{:}U'$, where $U' \leq U$.*

Proof. The proof is much the same as the one of lemma 2.3.1. The case for $M \equiv \mathbf{coerce}^U(M')$ is solved by a straightforward use of the induction hypothesis. □

Theorem 4.2.2 (Generalized Subject Reduction) *Let $M{:}U$. If $M \triangleright^* N$ then $N{:}U'$, where $U' \leq U$.*

Proof. Use the proof of theorem 2.3.2: in the cases $M \equiv M_1 \cdot M_2$ and $M \equiv M_1 \bullet M_2$ add the subcase $M_1 \equiv \mathbf{coerce}^S(M_1')$ (for a suitable S) and $M \triangleright M_1' \circ M_2$. These are solved in the same way as the respective first subcases. □

4.2.2 Church Rosser

To prove that $\lambda\&$+*coerce* is **CR** we use once more the Hindley-Rosen lemma. In section 2.4 we proved that $\beta \cup \beta_\&$ is **CR**. It is very easy to check that (coerce) satisfies the diamond property. Therefore it just remains to prove that $\beta \cup \beta_\&$ and (coerce) commute, and apply once more the Hindley-Rosen lemma to obtain **CR** for the whole extension.

Lemma 4.2.3 *If $M \triangleright_{coerce} M'$ then $M[x := N] \triangleright_{coerce} M'[x := N]$*

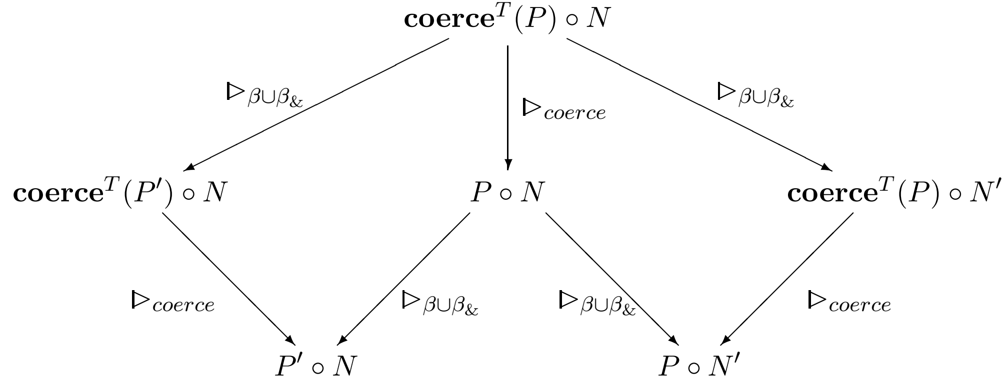
Proof. As in 2.4.3; just add to the table the case for the coercion. □

Lemma 4.2.4 *For all contexts $C[\ ]$ if $M \triangleright^*_{coerce} N$ then $C[M] \triangleright^*_{coerce} C[N]$*

Proof. As in 2.4.4: replace the case 3 by the one dealing with $\triangleright_{coerce}$ whose proof is straightforward □

Theorem 4.2.5 (Weak commutativity) *If $M \triangleright_{coerce} N_1$ and $M \triangleright_{\beta\cup\beta_\&} N_2$ then there exists N_3 such that $N_1 \triangleright^*_{\beta\cup\beta_\&} N_3$ and $N_2 \triangleright^*_{coerce} N_3$*

Proof. Once more the proof is the same as the corresponding one of theorem 2.4.5. We have only to add the subcase $M \equiv \mathbf{coerce}^T(P) \circ N$ whenever M is an application. Its solution is given by the following diagram chase:



□

As usual **CR** follows from the Hindley-Rosen lemma.

4.2.3 Strong Normalization

To prove the strong normalization of $\lambda\&^-$ with coercions we associate a well-founded complexity measure to each term and we show that each notion of reduction strictly decreases this measure. To define this measure we define a translation from $\lambda\&^-$+coercions to $\lambda\&^-$ with unconditional-$\beta_\&$ (see section 3.5). This translation simply erases all the explicit coercions from a term.

Definition 4.2.6

$$
\begin{array}{lcl}
[\![\varepsilon]\!] & = & \varepsilon \\
[\![x]\!] & = & x \\
[\![\lambda x^T.M]\!] & = & \lambda x^T.[\![M]\!] \\
[\![M\&^T N]\!] & = & [\![M]\!]\&^T[\![N]\!] \\
[\![M \circ N]\!] & = & [\![M]\!] \circ [\![N]\!] \\
[\![\mathbf{coerce}^T(M)]\!] & = & [\![M]\!]
\end{array}
$$

□

Given a term M of $\lambda\&^-$+coercions its complexity measure is given by the lexicographical order of $(\mathcal{N},\mathcal{C})$ where $\mathcal{C}$ is the number of coercions appearing in M and $\mathcal{N}$ is the sum of lengths of all the reductions starting from $[\![M]\!]$ (note that this is a finite number: $\lambda\&^-$ with uncond-$\beta_\&$ is strongly normalizing; thus this is the sum of the lengths of the paths of a finitely branching, bounded tree, which is finite by the König's Lemma). It is very easy to verify that the rule (coerce) decreases this measure since it decreases the $\mathcal{C}$ component leaving $\mathcal{N}$ unchanged. To prove that also β and $\beta_\&$ decrease this measure is a little more difficult.

Lemma 4.2.7 $M{:}T \quad \Rightarrow \quad [\![M]\!]{:}T' \leq T$

Proof. A trivial induction on M. □

Lemma 4.2.8 $M \rhd_{\beta_{\&}} N \qquad \Rightarrow \qquad [\![M]\!] \rhd_{\beta_{\&}} [\![N]\!]$

Proof. When M is the redex then the result follows from the definition of $uncond - \beta_{\&}$ and the lemma above. In all the other cases is follows by induction on M. □

The lemma above proves that $\beta_{\&}$ strictly decreases the $\mathcal{N}$ component. The same can be done for β:

Lemma 4.2.9 $[\![M]\!][x := [\![N]\!]] = [\![M[x := N]]\!]$

Proof. A straightforward induction on M □

Lemma 4.2.10 $M \rhd_{\beta} N \qquad \Rightarrow \qquad [\![M]\!] \rhd_{\beta} [\![N]\!]$

Proof. When M is the redex then the result follows from lemma 4.2.9. In all the other cases is follows by induction on M. □

4.2.4 More on updatable records

With the introduction of explicit coercions we are now able to encode the record values defined by Cardelli and Mitchell [CM91][5] . Their records are constructed starting from an empty record value (denoted by $\langle\,\rangle$, as usual) by three elementary operations:

- *Extension* $\langle r|\ell_i = M\rangle$; adds a field of label ℓ_i and a value M to a record r provided that a field of label ℓ_i is not already present.
- *Extraction* $r.\ell_i$; extracts the value corresponding the label ℓ_i provided that a field having that label is present.
- *Restriction* $r\backslash\ell_i$; removes the field of label ℓ_i, if any, from the record r.

The encoding is defined as follows:

$$
\begin{array}{lcll}
\langle\,\rangle & \equiv & \varepsilon & \\
\langle r|\ell_i = M\rangle & \equiv & (r\&\lambda x^{L_i}.M) & \text{where } x \notin FV(M) \\
r.\ell_i & \equiv & r\bullet\ell_i & \\
r\backslash\ell & \equiv & \mathbf{coerce}^{\{L_j \to V_j\}_{j\in J} \setminus L}(r) & \text{where } r\colon \{L_j \to V_j\}_{j\in J}
\end{array}
$$

Where $\setminus$ is defined as in definition 2.5.2

As usual both conditions in *Extension* and *Extraction* are statically enforced in the encoding. And the remark done in section 2.5.3 on the polymorphism of the encoding is (unfortunately) valid also in this case.

The extension of $\lambda\&$ with explicit coercions will be widely used in the rest of this part of the thesis. See for example chapters 6 and 5.

Remark The whole section 4.2 can be paraphrased, to define the extension of $\lambda\&^{+}$ by explicit coercions, instead of $\lambda\&$. Just replace the references to the proofs of $\lambda\&$ by the corresponding ones of the section of $\lambda\&^{+}$.

[5] We are not able to encode record types defined in the cited paper since we have no (linguistic) operation on types.

4.3 Unifying overloading and λ-abstraction: $\lambda^{\{\}}$

In this section we define a minimal system implementing overloading with late binding. The goal is to use as few operators as possible. Therefore we renounce having "extensible" overloaded functions (i.e. functions to which one can add new branches by the & operator). Terms are are built from variables by an operator of abstraction and one of application. Types are built from a set of basic types by the constructor for overloaded types. The key idea is to consider standard functions (λ-abstractions) as overloaded functions with just one branch. We use a rule similar to $\beta^+_\&$ in order not to have to use call-by-value when there is a unique branch (i.e. when we perform β-reductions).

$$\begin{array}{lcll} T & ::= & A \mid \{S_1 \to T_1, \cdots, S_n \to T_n\} & n \geq 1 \\ M & ::= & x \mid \lambda x(M_1 \colon S_1 \Rightarrow T_1, \cdots, M_n \colon S_n \Rightarrow T_n) \mid MM & n \geq 1 \end{array}$$

Since there is only one type constructor, there is also only one subtyping rule:

$$\text{(subtype)} \qquad \frac{\forall i \in I, \exists j \in J \; U''_i \leq U'_j \text{ and } V'_j \leq V''_i}{\{U'_j \to V'_j\}_{j \in J} \leq \{U''_i \to V''_i\}_{i \in I}}$$

Types

As usual we have the rules of type good formation: every atomic type belongs to **Types** and if for all $i, j \in I$

a. ($U_i, V_i \in$ **Types**)

b. ($U_i \leq U_j \Rightarrow V_i \leq V_j$)

c. ($U_i \Downarrow U_j \Rightarrow$ there is a unique $h \in I$ such that $U_h = \inf\{U_i, U_j\}$)
then $\{U_i \to V_i\}_{i \in I} \in$ **Types**[6]

Note that variables are no longer indexed by their type. This because in the term $\lambda x(M_1 \colon S_1 \Rightarrow T_1, \cdots, M_n \colon S_n \Rightarrow T_n)$ the variable x should be indexed in each branch by a different type (i.e. the corresponding S_i). Thus we prefer to avoid indexing and introduce in the typing rules type contexts (denoted by Γ). We suppose to work modulo α-conversion so that the order in Γ is not significant:

Type-checking

$$[\text{TAUT}] \qquad \Gamma \vdash x \colon \Gamma(x)$$

$$[\text{INTRO}] \qquad \frac{\forall i \in I \qquad \Gamma, (x \colon S_i) \vdash M_i \colon U_i \leq T_i}{\Gamma \vdash \lambda x(M_i : S_i \Rightarrow T_i)_{i \in I} \colon \{S_i \to T_i\}_{i \in I}}$$

[6] The restriction c+ works as well

$$[\text{ELIM}] \qquad \frac{\Gamma \vdash M\colon \{S_i \to T_i\}_{i\in I} \quad \Gamma \vdash N\colon S}{\Gamma \vdash MN\colon T_j} \qquad S_j = \min_{i\in I}\{S_i | S \leq S_i\}$$

Reduction

The selection of the branch of an overloaded function needs the type of its argument. Since this argument may be an open term (and variables are no longer indexed by their type) reduction will depend on a typing context Γ. Thus we define a family of reductions, subscripted by typing contexts $\rhd_\Gamma \subseteq \textbf{Terms} \times \textbf{Terms}$, such that if $M \rhd_\Gamma N$ then $FV(M) \subseteq dom(\Gamma)$. (Avoid confusion between this notation and $\rhd_R$ where R is a notion of reduction)

We have the following notion of reduction:

ζ) Let $S_j = \min_{i\in I}\{S_i | U \leq S_i\}$ and $\Gamma \vdash N\colon U$; if N is closed and in normal form or $\{S_i | S_i \leq S_j\} = \{S_j\}$ then

$$\lambda x(M_i : S_i \Rightarrow U_i)_{i\in I} N \;\; \rhd_\Gamma \;\; M_j[x := N]$$

Then there are the rules for the context closure: the change of the context must be taken into account when reducing inside λ-abstractions:

$$\frac{M \rhd_\Gamma M'}{MN \rhd_\Gamma M'N} \qquad\qquad \frac{N \rhd_\Gamma N'}{MN \rhd_\Gamma MN'}$$

$$\frac{M_i \;\; \rhd_{\Gamma,(x:S_i)} M_i'}{\lambda x(\cdots M_i\colon S_i \Rightarrow T_i \cdots) \rhd_\Gamma \lambda x(\cdots M_i'\colon S_i \Rightarrow T_i \cdots)}$$

Note that if $M \rhd_\Gamma N$ then $FV(N) \subseteq FV(M)$ thus the transitivity closure of $\rhd_\Gamma$ is well-defined.

4.3.1 Subject Reduction

To prove that $\lambda^{\{\}}$ satisfies the subject reduction property, we define a translation $[\![\,]\!]_\Gamma$ from $\lambda^{\{\}}$ to $\lambda\&^+$ with the following properties:

1. $\Gamma \vdash M\colon T \qquad \Leftrightarrow \qquad \vdash [\![M]\!]_\Gamma\colon T$
2. $M \rhd_\Gamma N \qquad \Rightarrow \qquad [\![M]\!]_\Gamma \rhd^* [\![N]\!]_\Gamma$

It is then clear that the subject reduction of $\lambda^{\{\}}$ follows by the subject reduction of $\lambda\&^+$.

Define an arbitrary *total* order $\preceq$ on **Types** with the following property: if $S \leq T$ then $S \preceq T$.[7] Given an overloaded type $\{S_i \to T_i\}_{i=i..n}$ we denote by σ the permutation that orders the S_i's according to $\preceq$. Thus $S_i \leq S_j$ implies $\sigma(i) \leq \sigma(j)$. This permutation is used to translate $\lambda^{\{\}}$ into $\lambda\&^+$.

$$\begin{array}{lcl}
[\![x]\!]_\Gamma & = & x^{\Gamma(x)} \\
[\![MN]\!]_\Gamma & = & [\![M]\!]_\Gamma \bullet [\![N]\!]_\Gamma \\
[\![\lambda x(M_i : S_i \Rightarrow U_i)_{i=1..n}]\!]_\Gamma & = & (\cdots(\varepsilon \;\&^{\{S_{\sigma(1)} \to T_{\sigma(1)}\}} \lambda x^{S_{\sigma(1)}}.[\![M_{\sigma(1)}]\!]_{\Gamma,(x:S_{\sigma(1)})}) \\
 & & \qquad \cdots \&^{\{S_{\sigma(i)} \to T_{\sigma(i)}\}_{i=1..n}} \lambda x^{S_{\sigma(n)}}.[\![M_{\sigma(n)}]\!]_{\Gamma,(x:S_{\sigma(n)})})
\end{array}$$

[7]Remember that $\leq$ is just a preorder. Therefore strictly speaking $\preceq$ is defined on **Types**/$\sim$ where $S \sim T$ iff $S \leq T \leq S$. This however does not affect the substance of what follows

Lemma 4.3.1 $\Gamma \vdash M{:}T \Leftrightarrow \vdash [\![M]\!]_\Gamma{:}T$

Proof. By a straightforward induction on the structure of M. Just remark in the case of $M \equiv \lambda x(M_i : S_i \Rightarrow U_i)_{i=1..n}$ that by the definition of σ every subterm of the translation has a well-formed type. □

Lemma 4.3.2 $[\![M]\!]_{\Gamma,(x:T)}[x^T := [\![N]\!]_\Gamma] = [\![M[x{:}= N]]\!]_\Gamma$

Proof. A straightforward induction on the structure of M □

Theorem 4.3.3 *If* $\Gamma \vdash M{:}T$ *and* $M \ \triangleright_\Gamma N$ *then* $[\![M]\!]_\Gamma \ \triangleright^* [\![N]\!]_\Gamma$

Proof. We first prove the case for $M \equiv \lambda x(M_i{:}S_i \Rightarrow U_i)_{i=1..n}P$ and $N \equiv M_j[x{:}= P]$. By induction on the number n of branches of M:

$$
\begin{array}{llll}
n=1 & [\![M]\!]_\Gamma & \equiv & (\varepsilon \& \lambda x_1^S.[\![M_1]\!]_{\Gamma,(x:S_1)}) \bullet [\![P]\!]_\Gamma \\
 & & \triangleright_{\beta^+_\&} & (\lambda x_1^S.[\![M_1]\!]_{\Gamma,(x:S_1)}) \cdot [\![P]\!]_\Gamma \\
 & & \triangleright_\beta & [\![M_1]\!]_{\Gamma,(x:S_1)}[x^{S_1}{:}= [\![P]\!]_\Gamma] \\
 & & = & [\![M_1[x{:}= P]]\!]_\Gamma \qquad \text{by lemma 4.3.2} \\
 & & \equiv & [\![N]\!]_\Gamma
\end{array}
$$

$n > 1$ Then there are two possible subcases:

1. $j = \sigma(n)$. By lemma 4.3.1 the last branch is selected thus $[\![M]\!]_\Gamma \ \triangleright_{\beta^+_\&} (\lambda x^{S_{\sigma(n)}}.[\![M_{\sigma(n)}]\!]_{\Gamma,(x:S_{\sigma(n)})}) \cdot [\![P]\!]_\Gamma$ which is proved as in the previous case.
2. $j \neq \sigma(n)$. Again by lemma 4.3.1 the first branch is selected thus

$$
\begin{array}{llll}
[\![M]\!]_\Gamma & \triangleright_{\beta^+_\&} & (\varepsilon \& \cdots \& \lambda x^{S_{\sigma(n-1)}}.[\![M_{\sigma(n-1)}]\!]_{\Gamma,(x:S_{\sigma(n-1)})}) \bullet [\![P]\!]_\Gamma & \\
 & = & [\![\lambda x(M_i : S_i \Rightarrow U_i)_{i \in [1..n] \setminus \{\sigma(n)\}} P]\!]_\Gamma & \text{by the definition of } \sigma \\
 & \triangleright^* & [\![M_j[x{:}= P]]\!]_\Gamma & \text{by induction hypothesis}
\end{array}
$$

The proof of the theorem is then easily obtained by induction on the structure of M, performing a case analysis on the definition of $\triangleright$. □

Corollary 4.3.4 (subject reduction) *If* $\Gamma \vdash M{:}T$ *and* $M \ \triangleright^*_\Gamma N$ *then* $\Gamma \vdash N{:}S \leq T$

Proof. We prove the theorem for one step reductions. The result follows by induction on the number of steps. If $\Gamma \vdash M{:}T$ then by lemma 4.3.1 $\vdash [\![M]\!]_\Gamma{:}T$; by lemma 4.3.3 $[\![M]\!]_\Gamma \ \triangleright^* [\![N]\!]_\Gamma$; from the subject reduction theorem for $\lambda\&^+$ we obtain $\vdash [\![N]\!]_\Gamma{:}S \leq T$; thus applying once more lemma 4.3.1 we obtain the result. □

4.3.2 Church-Rosser

In this section we provide a proof that for all Γ the relation is $\triangleright_\Gamma$ is Church-Rosser. We follow a technique due to W. Tait and P. Martin-Löf (see [Bar84]).

Lemma 4.3.5 *If a binary relation satisfies the diamond property, so it does its transitive closure*

Proof. See Lemma 3.2.2 of [Bar84]. □

We now define a relation $\blacktriangleright_\Gamma$ that satisfies the diamond property and whose transitive closure is $\triangleright^*_\Gamma$. Then it follows by lemma 4.3.5 that $\triangleright^*_\Gamma$ satisfies the diamond property, i.e. $\triangleright_\Gamma$ is **CR**.

Definition 4.3.6 [parallel reduction]

1. $M \blacktriangleright_\Gamma M$
2. $\forall i \in I \ \ M_i \blacktriangleright_{\Gamma,(x:S_i)} M'_i \quad\Rightarrow\quad \lambda x(M_i\colon S_i \Rightarrow T_i)_{i\in I} \blacktriangleright_\Gamma \lambda x(M'_i\colon S_i \Rightarrow T_i)_{i\in I}$
3. $M \blacktriangleright_\Gamma M' \ \ N \blacktriangleright_\Gamma N' \quad\Rightarrow\quad MN \blacktriangleright_\Gamma M'N'$
4. $N \blacktriangleright_\Gamma N' \ \ \forall i \in I \ \ M_i \blacktriangleright_{\Gamma,(x:S_i)} M'_i \quad\Rightarrow\quad (\lambda x(M_i\colon S_i \Rightarrow T_i)_{i\in I})N \blacktriangleright_\Gamma M'_j[x\colon= N']$ (*)

(*) if N' is closed and in normal form or $\{S_i | S_i \leq S_j\} = \{S_j\}$, where $\Gamma \vdash N'\colon U$ and $S_j = \min\{S_i | U \leq S_i\}$. □

Lemma 4.3.7 *For all* Γ, $\triangleright^*_\Gamma$ *is the transitive closure of* $\blacktriangleright_\Gamma$

Proof. Note that $\triangleright^=_\Gamma \subseteq \blacktriangleright_\Gamma \subseteq \triangleright^*_\Gamma$. Since $\triangleright^*_\Gamma$ is the transitive closure of $\triangleright^=_\Gamma$, so it is of $\blacktriangleright_\Gamma$. □

Lemma 4.3.8 *If* $M \blacktriangleright_{\Gamma,(x:T)} M'$, $N \blacktriangleright_\Gamma N'$ *and* $\Gamma \vdash N\colon S \leq T$ *then* $M[x\colon= N] \blacktriangleright_\Gamma M'[x\colon= N']$

Proof. For notational convenience set $\bar\Gamma \equiv \Gamma, (x\colon T)$. The result follows by induction on the definition of $M \blacktriangleright_{\bar\Gamma} M'$:

CASE 1 $M' \equiv M$. Then we have to show that $M[x\colon= N] \blacktriangleright_\Gamma M[x\colon= N']$. This follows by induction on the structure of M, as shown in the following table:

M	LHS	RHS	comment
ε	ε	ε	OK
x	N	N'	OK
y	y	y	OK
PQ	$P[\ \ ]Q[\ \ ]$	$P[\ \ ']Q[\ \ ']$	use the induction hyp.
$\lambda y.(P_i\colon S_i \Rightarrow T_i)$	$\lambda y.(P_i[\ \ ]\colon S_i \Rightarrow T_i)$	$\lambda y.(P_i[\ \ ']\colon S_i \Rightarrow T_i)$	use the induction hyp.

CASE 2 $M \equiv \lambda y(P_i\colon S_i \Rightarrow T_i)_{i\in I} \blacktriangleright_{\bar\Gamma} \lambda y(P'_i\colon S_i \Rightarrow T_i)_{i\in I} \equiv M'$ By induction hypothesis one has $\forall i \in I \ P_i[x\colon= N] \blacktriangleright_{\Gamma,(y:S_i)} P'_i[x\colon= N']$, whence

$$\lambda y(P_i[x\colon= N]\colon S_i \Rightarrow T_i)_{i\in I} \blacktriangleright_\Gamma \lambda y(P'_i[x\colon= N']\colon S_i \Rightarrow T_i)_{i\in I}$$

CASE 3 $M \equiv PQ \blacktriangleright_{\bar\Gamma} P'Q' \equiv M'$; as in the case above, the result follows from a straightforward use of the induction hypothesis.

CASE 4 $M \equiv (\lambda y(P_i\colon S_i \Rightarrow T_i)_{i\in I})Q \blacktriangleright_{\bar\Gamma} P'_j[y\colon= Q'] \equiv M'$ where $Q \blacktriangleright_{\bar\Gamma} Q'$, $\forall i \in I \ P_i \blacktriangleright_{\bar\Gamma,(y:S_i)} P'_i$, $\bar\Gamma \vdash Q'\colon U$ and $S_j = \min_{i\in I}\{S_i | U \leq S_i\}$. Without loss of generality we can suppose that $y \notin FV(N)$ (and thus $y \notin FV(N')$). There are two subcases:

1. Q' is closed and in normal form:

$$M[x\colon= N] \quad = \quad (\lambda y(P_i[x\colon= N]\colon S_i \Rightarrow T_i)_{i\in I})(Q[x\colon= N])$$

By induction hypothesis $P_i[x{:=}N] \blacktriangleright_{\Gamma,(y:S_i)} P'_i[x{:=}N']$ and $Q[x{:=}N] \blacktriangleright_\Gamma Q'[x{:=}N']$. Since Q' is closed then $Q'[x := N'] \equiv Q'$, thus the j-th branch will be again selected:

$$\begin{array}{lll} \blacktriangleright_\Gamma & P'_j[x{:=}N'][y{:=}Q'[x{:=}N']] & \\ \blacktriangleright_\Gamma & P'_j[y{:=}Q'][x{:=}N'] & y \notin FV(N') \\ = & M'[x{:=}N'] & \end{array}$$

2. $\{S_i|S_i \leq S_j\} = \{S_j\}$. Observe that $\blacktriangleright_\Gamma \subseteq \rhd^*_\Gamma$ (see lemma 4.3.7); then the subject reduction for $\rhd^*_\Gamma$ (corollary 4.3.4) implies the subject reduction for $\blacktriangleright_\Gamma$. Let $\bar{\Gamma} \vdash Q'{:}U$ and consider the term
$$\lambda x(Q'{:}T \Rightarrow U)N$$
It is easy to verify that this term is well-typed and $\Gamma \vdash \lambda x(Q'{:}T \Rightarrow U)N : U$. Note also that $\lambda x(Q'{:}T \Rightarrow U)N \blacktriangleright_\Gamma Q'[x{:=}N']$. Thus by the subject reduction theorem for $\blacktriangleright_\Gamma$ we deduce that $\Gamma \vdash Q'[x{:=}N']{:}U' {\leq} U$. Then we have:

$$M[x{:=}N] \quad = \quad (\lambda y(P_i[x{:=}N]{:}S_i \Rightarrow T_i)_{i\in I})(Q[x{:=}N])$$

By induction hypothesis $P_i[x{:=}N] \blacktriangleright_{\Gamma,(y:S_i)} P'_i[x{:=}N']$ and $Q[x{:=}N] \blacktriangleright_\Gamma Q'[x{:=}N']$. Furthermore $\bar{\Gamma} \vdash Q'[x{:=}N']{:}U' {\leq} U {\leq} S_j$. By hypothesis $\{S_i|S_i \leq S_j\} = \{S_j\}$ thus the j-th branch is again selected:

$$\begin{array}{lll} \blacktriangleright_\Gamma & P'_j[x{:=}N'][y{:=}Q'[x{:=}N']] & \\ \blacktriangleright_\Gamma & P'_j[y{:=}Q'][x{:=}N'] & y \notin FV(N') \\ = & M'[x{:=}N'] & \end{array}$$

□

Lemma 4.3.9 $\blacktriangleright_\Gamma$ *satisfies the diamond property*

Proof. We write " $M, N \blacktriangleright_\Gamma P$ " for " $M \blacktriangleright_\Gamma P$ *and* $N \blacktriangleright_\Gamma P$ " and " $M \blacktriangleright_\Gamma P, Q$ " for " $M \blacktriangleright_\Gamma P$ *and* $M \blacktriangleright_\Gamma Q$ ".

By induction on the definition of $M \blacktriangleright_\Gamma M'$ we show that for all $M \blacktriangleright_\Gamma M''$ there exists M''' such that $M', M'' \blacktriangleright_\Gamma M'''$.

The only interesting case is when $M \equiv (\lambda y(P_i{:}S_i \Rightarrow T_i)_{i\in I})Q$, $M' \equiv P'_j[y{:=}Q']$ and $M'' \equiv (\lambda y(P''_i{:}S_i \Rightarrow T_i)_{i\in I})Q''$ where $Q \blacktriangleright_\Gamma Q', Q''$ and $\forall i \in I\ P_i \blacktriangleright_{\Gamma,(y:S_i)} P'_i, P''_i$ and $\Gamma \vdash Q'{:}U$ and $S_j = \min_{i\in I}\{S_i|U \leq S_i\}$:[8]

By induction hypothesis for all $i \in I$ there exist P'''_i and Q''' such that $P'_i, P''_i \blacktriangleright_{\Gamma,(y:S_i)} P'''_i$ and $Q', Q'' \blacktriangleright_\Gamma Q'''$. By lemma 4.3.8 one has $P'_j[y{:=}Q'] \blacktriangleright_\Gamma P'''_j[y{:=}Q''']$. To obtain the result it just remains to prove that $(\lambda y(P''_i{:}S_i \Rightarrow T_i)_{i\in I})Q'' \blacktriangleright_\Gamma P'''_j[y{:=}Q''']$. This is obtained by showing that $\min_{i\in I}\{S_i|U \leq S_i\} = \min_{i\in I}\{S_i|U' \leq S_i\}$ where $\Gamma \vdash Q'''{:}U'$. There are two subcases:

1. Q' is closed and in normal form: but then $Q''' \equiv Q'$ therefore $U' \equiv U$.

[8] Sorry for all those "and" but commas were too confusing

2. $\{S_i | S_i \leq S_j\} = \{S_j\}$. By the subject reduction for $\blacktriangleright_\Gamma$ (see the proof of lemma 4.3.8) one has that $U' \leq U$ this implies that $S_j \in \{S_i | U' \leq S_i, i \in I\}$. From $\{S_i | S_i \leq S_j\} = \{S_j\}$ it follows that $S_j = \min_{i \in I}\{S_i | U' \leq S_i\}$.

All the other cases are either trivial (case $M \equiv M'$) or they follow from a straightforward use of the induction hypothesis. □

Corollary 4.3.10 *For every* Γ, $\blacktriangleright_\Gamma$ *is* **CR**

Proof. It follows from lemmas 4.3.5, 4.3.7 and 4.3.9 □

We want to end this section, by remarking that $\lambda^{\{\}}$ is not completely deprived of interest for the modeling of object-oriented programming. Indeed it constitutes a first step toward the definition of a pure calculus of methods and generic functions (i.e. of branches and overloaded functions); and it is important to stress that this is exactly the way Dylan works.

4.4 Reference to other work

At the end of this chapter devoted to the variations of $\lambda\&$ we have to cite two modifications proposed by Hideki Tsuiki [Tsu94] that he calls $\lambda\&C$ and $\lambda\&C^*$. These calculi are essentially $\lambda\&$+coerce but in which the standard definition of substitution is modified so that a variable is always substituted by a term of the same type; this is obtained by explicitly coercing the type of the argument of the substitution to the type of the variable; thus the β-reduction becomes:

$$(\beta C) \qquad (\lambda x^T.M)N \rhd M[x^T := \mathbf{coerce}^T(N)]$$

The two subcalculi then differ for the implicit meaning of the overloaded types, which corresponds to different reduction rules for the coercions in the application.

The main motivation of these modifications is to define calculi that strictly satisfy the subject reduction property, that is in which the reductions preserve the type of a term (and do not reduce it as for $\lambda\&$). Of course in this way there is no possible late binding since types do not evolve during computation. Though in the section dedicated to the future work Tsuiki makes an interesting proposition to use a peculiar form of implicit bounded polymorphism to mimic late binding, and that surely deserves much attention.

Chapter 5

A meta-language from $\lambda\&$

In chapter 2 we have introduced the $\lambda\&$-calculus and we have showed how this calculus could be intuitively used to model some features of object-oriented programming.

However, $\lambda\&$ is not adequate to a formal study of the properties of real object-oriented languages, and it was not meant for this: it is a calculus not a meta-language; thus, even if it possesses the key mechanisms to model object-oriented features, it cannot be used to "reason about" (i.e. to prove properties of) an object-oriented language.

For these reasons in this chapter we define a meta-language (i.e. a language to reason about —object-oriented— languages)[1] that we call λ_object. This language is still based on the key mechanisms of $\lambda\&$ (essentially, overloading with late binding) but it is enriched by some features (like commands to define new types, to work on their representations, to handle the subtyping hierarchy, to change the type of a term or to modify a discipline of dispatching etc.) that are necessary to reproduce the constructs of a programming language and which $\lambda\&$ lacks.

However this passage is not smooth since the meta-language has to be formed by very few constructs (in order to keep to a reasonable size the proofs of the properties of the studied languages) and it must be proved that it meets the subject reduction property.

We also show how to use λ_object to prove properties of an object-oriented language. For this purpose we give the formal definition of the toy object-oriented language and of the type checking algorithm we informally described in chapter 1 and we translate the programs of the toy object-oriented language into this meta-language. We prove that every well typed program of the former is translated into a well typed program of the latter; since this last one enjoys the subject reduction property, the reduction of the translated program does not go wrong on a type error; in particular this proves the correction of the type-checker for the toy-language.

Consequently, the chapter is organized as follows: section 5.1 gives the formal description of the toy-language and of its type discipline. We do not give any reduction rule since the formal operational behavior of the language is given by the translation that follows. In section 5.2 we describe λ_object: we give its operational semantics, a type-checker and we prove for it the subject-reduction theorem. In section 5.3 (technically, the most difficult one) we define the translation and we prove the correction of the type discipline for the toy

[1] In this case the prefix "meta" is used w.r.t. the object-oriented languages

language. The reader can find in Appendix A the implementation in CAML LIGHT of an interpreter for λ_object as well as some examples of its use.

5.1 The formal presentation of the toy language

In this section we give the formal presentation of the toy language and of its type discipline. We restrict our analysis to the main constructions of the language, omitting what is not strictly necessary to the comprehension of the object-oriented part like conditionals, natural numbers and their operations, and so on.

The formal description is given, as usual, in BNF. We use `courier` font for terminal symbols and *italics* for non terminals; parenthesis and brackets that are metasymbols of BNF are written in *italics*, too. Thus confusion must be avoided between `()[]` which belong to the syntax and *()[]* which belong to the BNF notation. We have the following nonterminal symbols :

NON TERMINAL	MEANING
classname	names for classes
x	variables
method	
message	
r	record expressions
exp	expressions
instanceVariables	
p	programs
interface	
A	atomic types
D	input types
R	record types
T	raw types
V	interface types

The first two nonterminals denote strings of characters. For the others we give the formal grammars.

This section is organized in two main subsections: one describes the terms (expressions and programs) of the language; the other describes the types of the language and the type-checker which is formally defined by means of syntax-driven rules.

5.1.1 The terms of the language

In chapter 1 we have presented the constructs that form our kernel language. Roughly speaking these constructs can be divided in two groups: those which introduce new types (this group contains only the class definitions) and those which can be evaluated and return a value (all the others). In programs we keep separated the elements of these groups: a program of the toy language is formed by a suite of class definitions followed by an expression where

these definitions are used and no other class definition appears. This separation is necessary to have static type-checking: if we allowed a class definition inside, say, a function then the type hierarchy would vary according to whether this function is called or not; thus also the typing of expressions would vary; for example an overloaded function which satisfied the multiple inheritance condition might no longer satisfy it because a new class has been dynamically created. Since the execution of a function is undecidable, dynamic type checking would be necessary. [2]

Thus we impose that all types and the subtyping relation are known *before* any step of calculation. Programs are formed by an expression (a term that does not contain class definitions) preceded by (possibly zero) class definitions. Among them the type-checker statically picks up those which are well-typed.

Expressions and Programs

We start by defining record expressions. Record expressions possess a peculiar relevance in our system, where they are used to represent the internal state (the instance variables) of objects.

$$r ::= \{\ell_1 = exp_1; \ldots; \ell_n = exp_n\}$$

We next give the productions for expressions *exp* and programs *p* which are defined in terms of the non-terminals *method*, *instanceVariables* and *interface*:

$$
\begin{array}{rcl}
exp & ::= & x \\
 & | & \texttt{fn}(x_1 : T_1, \ldots, x_n : T_n) \texttt{ => } exp \\
 & | & exp_1\texttt{(}exp_2\texttt{)} \\
 & | & \texttt{(}exp, \ldots, exp\texttt{)} \\
 & | & \texttt{fst(}exp\texttt{)} \mid \texttt{snd(}exp\texttt{)} \\
 & | & \texttt{let } x{:}T \texttt{ = } exp \texttt{ in } exp \\
 & | & \texttt{extend } classname \\
 & & \quad (message \texttt{ = } method\texttt{;})^+ \\
 & & \quad interface \\
 & & \texttt{in } exp \\
 & | & \texttt{new(}classname\texttt{)} \\
 & | & \texttt{self} \\
 & | & \texttt{(self.}\ell\texttt{)} \\
 & | & \texttt{(update } r\texttt{)} \\
 & | & \texttt{super[}A\texttt{](}exp\texttt{)} \\
 & | & \texttt{coerce[}A\texttt{](}exp\texttt{)}
\end{array}
$$

[2] Further study is required to find a trade off between static type checking and dynamic definition of classes

$$
\begin{array}{rcll}
 & | & \texttt{\&}\ \texttt{fn(}x_1{:}A_1,\ldots,x_{n_1}:A_{n_1}\texttt{)}\ \texttt{=>}\ exp_1 & \\
 & & \texttt{\&}\ \texttt{fn(}x_1{:}A_1,\ldots,x_{n_2}:A_{n_2}\texttt{)}\ \texttt{=>}\ exp_2 & \\
 & & \vdots & \\
 & & \texttt{\&}\ \texttt{fn(}x_1:A_1,\ldots,x_{n_m}:A_{n_m}\texttt{)}\ \texttt{=>}\ exp_m & (m\geq 1) \\
 & | & \texttt{[}\ exp_0 exp\ exp_1\ldots exp_n\ \texttt{]} & (n\geq 0)
\end{array}
$$

$$
\begin{array}{rcl}
p & ::= & exp \\
 & | & \texttt{class}\ classname\ [\ \texttt{is}\ classname\ (\ ,\ classname\ )^*] \\
 & & \quad instanceVariables \\
 & & \quad (message\ \texttt{=}\ method;)^* \\
 & & \quad interface \\
 & & \texttt{in}\ p
\end{array}
$$

$$
\begin{array}{rcl}
method & ::= & exp \\
message & ::= & x \\
interface & ::= & \texttt{[[}message : V;\ldots;message : V\texttt{]]} \\
instanceVariables & ::= & \{\ell_1 : T_1\texttt{=}exp_1;\ldots;\ell_n : T_n\texttt{=}exp_n\}
\end{array}
$$

The use of these constructs has already been explained. Just note, *en passant*, that the dot selection and record updating are allowed only on `self`. In this way we have the encapsulation of the internal state: instance variables are implemented by a record value, each variable being a label of the record. Thus objects behave at type level as record values, but the selection and the updating of a field is allowed only when the object is denoted by `self`, i.e. when the object is processed inside one of its methods. Note also that the non-terminals *method* and *message* are not strictly necessary since they are special cases of terms and variables (the type checking algorithm will require them to be respectively functions and variables with an overloaded type) but they make the rules more readable. Finally note that the branches of the overloaded functions are in a functional form (a branch cannot be, say, an application).

5.1.2 The types of the language

The types of a language are usually defined starting from a set of *atomic types* and applying type constructors (such as *list*, →, ...). A programming language always possesses some built-in atomic types (typically *Int*, *Bool*, *String*, etc...) on which some basic operations are

defined. Many languages, besides these basic types, offer the ability to the programmer to define its own atomic types. Object-oriented languages do it by means of class definitions; nay, this mechanism is the corner-stone of this programming style. As a matter of fact, class definitions are richer and more complex than the simple definition of an atomic type. Roughly speaking a class definition is composed by three distinct definitions: the definition of a new atomic type (the class-name), the definition of some operations for that atomic type (the methods), the definition of a partial order on the atomic types (subtyping relation) or, better, the definition of the type constraints for the newly introduced atomic type.

In brief, when a programmer defines a new class he declares by it a new atomic type, that is the class-name of that class definition. Besides the atomic types defined by the programmer, there are also some predefined atomic types. Usually also this built-in types are considered class-names: just consider them as predefined classes whose methods are the predefined operations (this is for example what is done in Smalltalk [GR83] and in Dylan [App92]).

Thus, without loss of generality, we can take as atomic types for our language only *class-names*. From these atomic types, by applying some type constructors, we build the whole set of types. We also define the *interface types* which are the types possibly preceded by the symbol # to pinpoint multi-methods.

Raw Types

$$R \quad ::= \quad \langle\!\langle \ell_1 : T_1; \ldots ; \ell_n : T_n \rangle\!\rangle \qquad \text{(record types)}$$

$$A \quad ::= \quad \textit{classname} \qquad \text{(atomic types)}$$

$$\begin{array}{lcll} T & ::= & A & \text{(raw types)} \\ & | & T \to T & \\ & | & (T \times \ldots \times T) & \\ & | & \{(A_1 \times \ldots \times A_{m_1}) \to T_1, \ldots, (A'_1 \times \ldots \times A'_{m_n}) \to T_n\} & {\scriptstyle (m_i \geq 1)} \end{array}$$

$$V \quad ::= \quad T \mid \#\{(A_1 \times \ldots \times A_{m_1}) \to T_1, \ldots, (A'_1 \times \ldots \times A'_{m_n}) \to T_n\} \qquad \text{(interface types)}$$

We use the metavariables T, U and W to range over raw types A to range over atomic types and D to range over atomic types or products of atomic types (i.e. the domains of the branches of an overloaded function). In the following if T denotes the type $\{U_i \to T_i\}_{i=1..n-1}$, the meta-notation $T \cup \{U_n \to T_n\}$ denotes the type $\{U_i \to T_i\}_{i=1..n}$ if $U_n \to T_n$ is different from all the arrow types in T, and it denotes T itself otherwise. In other terms $\cup$ denotes the usual set-theoretic union.

Subtyping

We give the formal rules that extend a subtyping relation defined on atomic types, to higher types. Since this extension depends on the specific constraints defined on the atomic types, we use in the rules a *type constraint system* which records these constraints:

Definition 5.1.1 A type constraint system (tcs) C is inductively defined by:

1. $\emptyset$ is a type constraint system

2. If C is a type constraint system and A_1, A_2 are atomic types then $C \cup (A_1 \leq A_2)$ is a type constraint system.

□

Next we give the definition of the rules informally described in section 1.2.2.

$$C \cup (A_1 \leq A_2) \vdash A_1 \leq A_2$$

$$\frac{C \vdash T_2 \leq T_1 \quad C \vdash U_1 \leq U_2}{C \vdash T_1 \to U_1 \leq T_2 \to U_2}$$

$$\frac{C \vdash U_1 \leq T_1 \ \ldots C \vdash U_n \leq T_n}{C \vdash (U_1 \times ... \times U_n) \leq (T_1 \times ... \times T_n)}$$

$$\frac{\text{for all } i \in I, \text{ there exists } j \in J \text{ such that } C \vdash D''_i \leq D'_j \text{ and } C \vdash U'_j \leq U''_i}{C \vdash \{D'_j \to U'_j\}_{j \in J} \leq \{D''_i \to U''_i\}_{i \in I}}$$

$$\frac{C \vdash U_1 \leq T_1 \ \ldots C \vdash U_k \leq T_k}{C \vdash \langle\!\langle \ell_1 \colon U_1; \ldots; \ell_k \colon U_k; \ldots; \ell_{k+j} \colon U_{k+j} \rangle\!\rangle \leq \langle\!\langle \ell_1 \colon T_1; \ldots; \ell_k \colon T_k \rangle\!\rangle}$$

The (pre)order for all types is given by the reflexive transitive closure of the rules above (as usual transitivity can be eliminted at higher types: transitivity is required just on the atomic types).

Finally we end this section by two definitions that are not strictly necessary, but which result very useful, when checking the state coherence condition and the updating, making the corresponding type-checking rules more readable. We define $\leq_{strict}$ as:

$$C \vdash \langle\!\langle \ell_1 \colon T_1; \ldots; \ell_k \colon T_k; \ldots; \ell_{k+j} \colon T_{k+j} \rangle\!\rangle \leq_{strict} \langle\!\langle \ell_1 \colon T_1; \ldots; \ell_k \colon T_k \rangle\!\rangle$$

In words, strict subtyping on records corresponds to field extension. Then we define the notation $\Subset$ (which used in the updating) as follows:

$$\frac{C \vdash U_1 \leq T_1 \ \ldots C \vdash U_k \leq T_k}{C \vdash \langle\!\langle \ell_1 \colon U_1; \ldots; \ell_k \colon U_k \rangle\!\rangle \Subset \langle\!\langle \ell_1 \colon T_1; \ldots; \ell_k \colon T_k; \ldots; \ell_{k+j} \colon T_{k+j} \rangle\!\rangle}$$

Two types are equivalent if they are syntactically equivalent modulo the order of the arrow types in overloaded types.

Well-formed types

We select among the raw types those that satisfy the conditions of covariance and multiple inheritance of section 1.2.2 and we call them *well-formed types*; the condition of state coherence concerns the definition of a class and will be checked directly on terms.

We denote the set of well-formed types by **Types**. Since the membership to **Types** depends on the definition of the subtyping relation on the atomic types, we index the symbol of membership by a type constraint system.

Notation 5.1.2 *Let $S \subseteq$ **Types**. we denote by $LB_C(S)$ the set $\{T \in_C \textbf{Types} \,|\, \forall T' \in S \;\; C \vdash T \leq T'\}$ of lower bounds of S with respect to the subtyping relation defined by C.*□

Definition 5.1.3 [well-formed types]

1. $A \in_C$ **Types** for each A atomic
2. if $T_1, T_2 \in_C$ **Types** then $T_1 \to T_2 \in_C$ **Types** and $T_1 \times T_2 \in_C$ **Types**
3. if for all $i, j \in I$
   - **(a)** $D_i, T_i \in_C$ **Types**
   - **(b)** if $C \vdash D_i \leq D_j$ then $C \vdash T_i \leq T_j$
   - **(c)** for all maximal type D in $LB_C(\{D_i, D_j\})$ there exists $h \in I$ such that $D_h = D$
   - **(d)** if $i \neq j$ then $D_i \neq D_j$

   then $\{D_i \to T_i\}_{i \in I} \in_C$ **Types**

□

By analogy we will denote by **AtomicTypes** the set of atomic types (the names of all classes) and by **RecordTypes** the set of the record types whose fields are associated to well-formed types.

Type checking rules

In this subsection we give the definition of the type checker. This is done by defining a typing rule for each construct described in the formal grammars above. We use a *type environment* Γ to record the type of the identifiers (parameters of functions and `self`) and a *state environment* S to record the type of the internal states of the classes defined up to that point; the domain of S (i.e. the values for which S is defined) is thus the set of all the names of the classes that have been defined.

More formally we define the relation $C; S; \Gamma \vdash p\colon T$, where C is a type constraint system, p a program, T a well-formed type and Γ and S are partial functions between the following sets:

$$\Gamma\colon (Vars \cup \{\texttt{self}\}) \to \textbf{Types}$$
$$S\colon \textbf{AtomicTypes} \to \textbf{RecordTypes}$$

The function Γ records the types of the various identifiers: $Vars$ are the identifiers for expressions and `self` is the identifier for the current object. The function S records the type of the internal state of the previously defined classes; since $\Gamma(\texttt{self})$ is the class-name of the current object (i.e. the current class) then $S(\Gamma(\texttt{self}))$ is the type of the internal state of the current object. Let X be any of these two functions; then we denote by $dom(X)$ the domain of X and by $X[x \leftarrow T]$ the function that for an argument a returns T if $a \equiv x$ and $X(a)$ otherwise; $X[x_i \leftarrow T_i]_{i=1\ldots n}$ is a shorthand for $X[x_1 \leftarrow T_1]\ldots[x_n \leftarrow T_n]$.

In order to simplify the treatment we suppose to have translated all the expressions of the form $\texttt{fn(}x_1\texttt{:}T_1\texttt{,}\ldots\texttt{,}x_n\texttt{:}T_n\texttt{) => } v$ into an equivalent unary function $\texttt{fn(}x\texttt{:}T_1 \times \ldots \times T_n\texttt{)} \texttt{ => } v[x_i := \texttt{fst snd}^{i-1}(x)]$ and all the declaration of instance variables $\texttt{\{}\ell_1 : T_1\texttt{=}v_1; \ldots; \ell_n : T_n\texttt{=}v_n\texttt{\}}$ into $\texttt{\{}\ell_1\texttt{=}v_1; \ldots; \ell_n\texttt{=}v_n\texttt{\}}\colon \langle\!\langle \ell_1 : T_1; \ldots; \ell_n : T_n \rangle\!\rangle$. The definition of the relation is inductively given by cases on the program p. Each rule is followed by an explanation, when it deserves one. We start with p formed by just an expression.

[TAUT] $$C;S;\Gamma \vdash x : \Gamma(x) \qquad x \in (\mathit{Vars} \cup \{\mathtt{self}\})$$

[FUNCT] $$\frac{C;S;\Gamma[x \leftarrow T] \vdash \mathit{exp}{:}\,U}{C;S;\Gamma \vdash \mathtt{fn(}x\mathtt{:}T\mathtt{)\ =>}\ \mathit{exp} : T \rightarrow U} \qquad T \in_C \mathbf{Types}$$

[APPL] $$\frac{C;S;\Gamma \vdash \mathit{exp}_1{:}\,T \rightarrow U \qquad C;S;\Gamma \vdash \mathit{exp}_2{:}\,W}{C;S;\Gamma \vdash \mathit{exp}_1(\mathit{exp}_2) : U} \qquad C \vdash W \leq T$$

[PROD] $$\frac{C;S;\Gamma \vdash \mathit{exp}_1{:}\,T_1 \qquad \ldots \qquad C;S;\Gamma \vdash \mathit{exp}_n{:}\,T_n}{C;S;\Gamma \vdash \mathtt{(}\mathit{exp}_1\mathtt{,}\ldots\mathtt{,}\mathit{exp}_n\mathtt{)}{:}\ (T_1 \times \ldots \times T_n)}$$

[RECORD] $$\frac{C;S;\Gamma \vdash \mathit{exp}_1{:}\,T_1 \qquad \ldots \qquad C;S;\Gamma \vdash \mathit{exp}_n{:}\,T_n}{C;S;\Gamma \vdash \{\ell_1 = \mathit{exp}_1\mathtt{;}\ldots\mathtt{;}\ell_n = \mathit{exp}_n\} : \langle\langle \ell_1 : T_1; \ldots; \ell_n : T_n \rangle\rangle}$$

[LET] $$\frac{C;S;\Gamma \vdash \mathit{exp}'{:}\,W \qquad C;S;\Gamma[x \leftarrow T] \vdash \mathit{exp}{:}\,U}{C;S;\Gamma \vdash \mathtt{let}\ x : T\ \mathtt{=}\ \mathit{exp}'\ \mathtt{in}\ \mathit{exp} : U} \qquad C \vdash W \leq T$$

These were the rules of the functional core and deserve very few explanations: just note that in the rule [APPL] a function accepts as argument every expression whose type is smaller than or equal to its domain. The rules for the object-oriented constructs are more interesting:

[NEW] $$C;S;\Gamma \vdash \mathtt{new(}A\mathtt{)}{:}\,A \qquad A \in \mathit{dom}(S)$$

The type of a new object is the name of its class. Of course this class must have been previously defined, and thus we check that $A \in \mathit{dom}(S)$.

[READ] $$C;S;\Gamma \vdash \mathtt{self}.\ell{:}\,T \qquad S(\Gamma(\mathtt{self})) = \langle\langle \ldots\ell{:}\,T\ldots \rangle\rangle$$

The expression $\mathtt{self}.\ell$ reads the value of an instance variable of an object and thus it must be contained inside the body of a method. Then $\Gamma(\mathtt{self})$ is the type (i.e., the class-name) of the current object and $S(\Gamma(\mathtt{self}))$ is the record type of its internal state.

[WRITE] $$\frac{C;S;\Gamma \vdash r{:}\,R}{C;S;\Gamma \vdash \mathtt{(update}\ r\mathtt{)} : \Gamma(\mathtt{self})} \qquad C \vdash R \Subset S(\Gamma(\mathtt{self}))$$

As in the previous rule this expression must be contained in a method. When by $(\mathtt{update}\ r)$ we update some instance variables, we have to check that the fields specified belong to the instance variables of the current class ($R \Subset S(\Gamma(\mathtt{self}))$); note that we need to specify only the instance variables we want to modify. In that case we return a value whose type is the current class (which is recorded in $\Gamma(\mathtt{self})$).

[OVABST] $$\frac{C;S;\Gamma \vdash \mathit{exp}_1{:}\,T_1 \ldots C;S;\Gamma \vdash \mathit{exp}_n{:}\,T_n}{C;S;\Gamma \vdash\ \mathtt{\&}\mathit{exp}_1\mathtt{\&}\ldots\mathtt{\&}\ \mathit{exp}_n{:}\,\{T_1\mathtt{,}\ \ldots\ \mathtt{,}T_n\}} \qquad \{T_1\mathtt{,}\ \ldots\ \mathtt{,}T_n\} \in_C \mathbf{Types}$$

The type of an overloaded function is the set of the types of its branches; by the production for overloaded function, the T_i's are arrow types. Also one has to check that the obtained type is well-formed.

$$[\textsc{OvAppl}] \qquad \frac{C;S;\Gamma \vdash exp\colon \{D_i \to T_i\}_{i\in I} \qquad C;S;\Gamma \vdash exp_j\colon A_j \ \ (j{=}0..n)}{C;S;\Gamma \vdash \texttt{[}exp_0\ exp\ \ exp_1,\ldots,exp_n\ \texttt{]:}T_h}$$

$$\text{if } D_h = \min_{i\in I}\{D_i \mid C \vdash A_0 \times A_1 \times \ldots \times A_n \leq D_i\}.$$

When we pass a message or, more generally, we perform an overloaded application we look at the type of the function, *exp*, and we select the branch whose input type best approximates the type of the argument. The argument is $\texttt{(}exp_0\texttt{,}expr_1\texttt{,}\ldots\texttt{,}expr_n\texttt{)}$ and the selected branch is the branch h such that $D_h = \min_{i\in I}\{D_i \mid C \vdash A_0 \times A_1 \times \ldots \times A_n \leq D_i\}$. Note that if the set $\{D_i \mid C \vdash A_0 \times A_1 \times \ldots \times A_n \leq D_i, i \in I\}$ is not empty the the *min* exists thanks to the condition of multiple inheritance in the type of *exp*. If it is empty then the expression is not well-typed.

$$[\textsc{Coerce}] \qquad \frac{C;S;\Gamma \vdash exp\colon A}{C;S;\Gamma \vdash \texttt{coerce[}A'\texttt{](}exp\texttt{)}\texttt{:}A'} \qquad C \vdash A \leq A'$$

The construct $\texttt{coerce[}A'\texttt{](}exp\texttt{)}$ says to consider *exp* (whose type is A) as if it were of type A'. This is a type safe operation if and only if $A \leq A'$.

$$[\textsc{Super}] \qquad \frac{C;S;\Gamma \vdash exp\colon A}{C;S;\Gamma \vdash \texttt{super[}A'\texttt{](}exp\texttt{)}\texttt{:}A'} \qquad C \vdash A \leq A'$$

At type level super and coerce have exactly the same behavior.

Finally we have a special rule for multi-methods

$$[\textsc{Multi}] \qquad \frac{C;S;\Gamma \vdash exp_1\colon T_1 \ldots C;S;\Gamma \vdash exp_n\colon T_n}{C;S;\Gamma \vdash\ \texttt{\&}exp_1\texttt{\&}\ldots\texttt{\&}\ exp_n\texttt{:\#}\{T_1\texttt{,}\ \ldots\ \texttt{,}T_n\}} \qquad \{T_1\texttt{,}\ \ldots\ \texttt{,}T_n\} \in_C \mathbf{Types}$$

Note that this rule and [OvAbst] assign two different types to the same expression $\texttt{\&}exp_1\texttt{\&}\ldots\texttt{\&}\ exp_n$; however this ambiguity is solved by the use of # in the interfaces: If that expression is to be used as an overloaded function (and thus it is applied to an argument) then it must be typed by [OvAbst]. Instead [Multi] is used just for typing multi-methods; then the # desappears thanks to the definition of "$\leadsto$" (see the rule [Class]) since the branches are "distributed" on the more general type of the message (which has no #).

Now we show how to type class definitions; this is by far the hardest case. We want to type the program

$$\texttt{class}\ A\ \texttt{is}\ A_1\texttt{,}\ldots\texttt{,}A_n\ \ r\colon R\ \ m_1\texttt{=}exp_1\texttt{;}\ldots\texttt{;}m_k\texttt{=}exp_k\ \texttt{[[}\ m_1 : V_1\texttt{,}\ldots\texttt{,}m_k : V_k\ \texttt{]]}\ \texttt{in}\ p$$

Unfortunately the definition of a class on one line loses the clarity due to the vertical formatting: the class A is a subtype of $A_1, \ldots\ A_n$; it defines k new methods $m_1 \ldots\ m_k$ of type $V_1 \ldots V_k$; these methods add to those that A inherits from its supertypes; its instance variables are defined by the record type R with initial values given by r. The whole program is well-typed if:

1. The definitions in the class are well-typed
2. The program p with the new definitions introduced by the class is well-typed.

Thus the steps to achieve are:

[1.1] Check whether a class with the same name has not already been defined, i.e. $A \notin dom(S)$.

[1.2] Check whether the initial values are well-typed, i.e. $r : R$

[1.3] Check whether the state coherence condition is satisfied, i.e. $R \leq_{strict} S(A_1) \ldots R \leq_{strict} S(A_n)$

[1.4] Record the new type constraints by $C \cup (A \leq A_1) \cup \ldots \cup (A \leq A_n)$ and record the internal state of A by $S[A \leftarrow R]$.

[1.5] Update the type of the messages by adding the new branches defined in the class. We have to distinguish the case of a simple method from that of multi-method. For every message m_i in the interface such that V_i is a raw type we must update its current type $\Gamma(m_i)$ in the following way: $\Gamma(m_i) := \Gamma(m_i) \cup \{A \to V_i\}$ (where we use the convention that $\Gamma(m_i) = \emptyset$ if $m_i \notin dom(\Gamma)$). If the type of a message in the interface is preceded by a `#`, then the associated method is a multi-method; recall that the type of its argument is the cartesian product of the type of the current class with the types the dispatch is performed on (see section 1.2.2 and the rule [OvAppl]). Thus for example if in the interface m_i:$\#\{D \to U, D' \to T\}$ then we have the following updating: $\Gamma(m_i) := \Gamma(m_i) \cup \{(A \times D) \to U, (A \times D') \to T\}$. More generally we define

$$\{A \leadsto V\} = \begin{cases} \{(A \times D_i) \to U_i\}_{i \in I} & \text{if } V \equiv \#\{D_i \to U_i\}_{i \in I} \\ \{A \to V\} & \text{otherwise} \end{cases}$$

thus the updating of Γ becomes: $\Gamma(m_i) := \Gamma(m_i) \cup \{A \leadsto T_i\}$, where the same convention as before applies.

Also check if this yields well-formed overloaded types; this corresponds to verifying that the message redefined satisfies the conditions of *covariance*, *multiple inheritance* and input type uniqueness.

[1.6] Check whether the types given in the interface correspond to those of the methods[3], i.e. $exp_i : V_i$. This check must be performed in an environment where the current class is A (and thus `self`: A) and messages have the updated types of step [1.5] since methods can call one each other (they are *mutually recursive*).

[2.] Type-check the program p considering the newly introduced definitions, i.e. the type constraints and the internal state of step [1.4] and the new types for messages in step [1.5].

We next give the precise rule that includes all of these steps. In order to shorten the definition we use the following abbreviations:

- $S' \equiv S[A \leftarrow R]$; the function S where to the class A is associated the type of its internal state R.

[3]Subtyping would have sufficed in that case

- $C' \equiv C \cup (\bigcup_{i=1..n} A \leq A_i)$; the set C extended by the type constraints generated by the definition,
- $I \equiv$ `[[` $m_1 : V_1$`,...,`$m_k : V_k$ `]]`; the interface of the class
- $\Gamma' \equiv \Gamma[m_i \leftarrow \Gamma(m_i) \cup \{A \rightsquigarrow V_i\}]_{i=1..k}$; the environment Γ where the (overloaded) type of the messages is updated with the type of the new methods (branches) added by the class-definition

Then the rule [Class] is

$$\frac{C;S;\Gamma \vdash r{:}R \qquad C';S';\Gamma'[\mathtt{self} \leftarrow A] \vdash exp_j{:}V_j \ \ (j{=}1..k) \qquad C';S';\Gamma' \vdash p:T}{C;S;\Gamma \vdash \mathtt{class}\ A\ \mathtt{is}\ A_1\mathtt{,...,}A_n\ r{:}R\ \ m_1\mathtt{=}exp_1\mathtt{;...;}m_k\mathtt{=}exp_k\ \ I\ \mathtt{in}\ p:T}$$

if $A \notin dom(S)$, for $i = 1..n$ $C \vdash R \leq_{strict} S(A_i)$ and for $i = 1..k$ $\Gamma(m_i) \cup \{A \rightsquigarrow V_i\} \in_{C'}$ **Types**

Let us examine the single parts of this rule more in details: first we assure that a class with this name does not already exist ($A \notin dom(S)$) [1.1], we check the type of the initial values of the instance variables ($C;S;\Gamma \vdash r{:}R$) [1.2] and we verifythat the type of the internal state of the class is compatible with (i.e. it is an extension of) the states of its ancestors ($C \vdash R \leq_{strict} S(A_i)$ for $i = 1..n$ [4]) [1.3] (see page 81 for the motivations). Then we check that the defined messages possess well-formed overloaded types ($\Gamma(m_i) \cup \{A \rightsquigarrow V_i\} \in_{C'}$ **Types**), i.e. that they satisfy the conditions of covariance, multiple subtyping and input type uniqueness [1.5]; we also check that the methods have the type declared in the interface ($\vdash exp_j : V_j$) [1.6], and this check is performed in an environment where we have recorded in C' the newly introduced type constraints, in S' the type of the internal state of the current object [1.4] and in Γ' also the types of the new methods for a possible mutual recursion. Finally we type the rest of the program where the class is declared. In order to implement the protection mechanisms we restore in the environment the old values for `self` [2.].

We left for last the rule for [Extend] as, even if it type-checks a single expression, it is a special case of the rule [Class] where there are no type constraints and no instance variables to check; we have just to check that the class in the `extend` expression has been already defined (i.e. $A \in dom(S)$) :

$$[\textsc{Extend}] \quad \frac{C;S;\Gamma'[\mathtt{self} \leftarrow A] \vdash exp_j : V_j \ \ (j{=}1..k) \qquad C;S;\Gamma' \vdash exp : T}{C;S;\Gamma \vdash \mathtt{extend}\ A\ \ m_1\mathtt{=}exp_1\mathtt{;...;}m_k\mathtt{=}exp_k\ \ \mathtt{[[}m_1{:}V_1\mathtt{,...,}m_k{:}V_k\mathtt{]]}\ \mathtt{in}\ exp{:}T}$$

$A \in dom(S)$ and for $i = 1..k$ $\Gamma(m_i) \cup \{A \rightsquigarrow V_i\} \in_C$ **Types**

Note finally that because of the definition of $\cup$ and the condition of input type uniqueness, if m_i has already been defined for the class A then $\Gamma(m_i) \cup \{A \rightsquigarrow V_i\} \in$ **Types** if and only if $\{A \rightsquigarrow V_i\} \in \Gamma(m_i)$. In other terms if we redefine a method the new definition has to possess the same type as the old one. It is possible to use the technique of section 4.1.2 and weaken this condition so that to allow the new method to have any type which replacing the old one yields to a well-formed overloaded type; but we prefer this stricter type discipline since, in our opinion, redefinition should not completely upset the previous definition.

[4]the ancestor must have been already declared, otherwise S is not defined

5.2 λ_object

In this section we define the meta-language λ_object. We pass from a calculus, which possesses an equational presentation, to a language, which thus is associated to a reduction strategy and a set of values. It is as if we had the λ-calculus and we wanted to define the SECD machine. The analogy is quite suggestive since, as in the case of the SECD machine, we do not want an exact correspondence with the λ-calculus (e.g. as the one between the SECD machine and the λ_V: see [Plo75]); rather we aim to define a language that implements the "general" behavior of the λ&-calculus, and that constitutes a meta-language for object-oriented languages. A meta-language is conceived to "speak about", to describe a language. Thus it must possess the syntactic structures to reproduce the constructs of that language, structures that are not generally present in a calculus. To this end we provide λ_object with constructs to define new atomic types, to define a subtyping hierarchy on them, to work on the implementation of a value of atomic type, to define recursive terms, to change the type of a term and to deal with `super`. We give an operational semantics for the untyped terms, we define a notion of run-time type error and a type-checking algorithm. Finally we prove the subject reduction theorem (thus the correctness of the type-checker) which plays a key role, being λ_object intended for typed object-oriented languages.

The main decision in the definition of λ_object is how to represent objects. This decision will drive the rest of the definition of the language. Running languages usually implement objects by records formed by three kinds of fields: fields containing the values of the instance variables, fields used by the system (for example for garbage collection) and a special field containing a reference to the class of the object. Quoting from the book of Meyer about the implementation of Eiffel:

> "Apart from their normal fields, representing class attributes [instance variables], objects must be equipped with some supplementary information. In particular, every object must carry its dynamic type, which is needed for the implementation of the dynamic binding. Some further fields are needed by the garbage collector.
>
> Note that this self-referent aspect of objects (every object includes information about its own type) is not just one possible implementation technique; it is necessary in any implementation of object-oriented concepts if dynamic binding is to work properly. This idea is reminiscent of *tagged architectures* in hardware."[pag. 336]

Obviously in this theoretical account we are not interested in the fields for the system, hence an object in λ_object will be formed only by the values of its instance variables (the so-called *internal state*) and by a *tag* indicating the class of the object. The tag of an object must uniquely determine the type of the object, for in our approach the selection of a method is based on the type of the object. There are two reasonable ways to do it, and in both of them the name of the class is considered an atomic type:

(a) An object is a record whose fields are the instance variables plus a special empty field whose type is the name of the class

(b) An object is a record whose fields are the instance variables and which is given a tag, say A, by applying it to a special constructor in^A. In other terms, in^{tag} is the constructor

for the values of (atomic) type *tag* whose internal representation is given by the record of the instance variables.

For instance, consider the class `2DPoint` of example 1.1.3. A new instance of this class would be implemented according to (a) as

$$\langle is_a = *; x = 0; y = 0\rangle : \langle\!\langle is_a\colon 2DPoint; x\colon Int; y\colon Int\rangle\!\rangle$$

(where $*$ is a value possessing every type) and according to (b) as

$$in^{2DPoint}(\langle x = 0; y = 0\rangle) : 2DPoint$$

Let us compare the two approaches. The former possesses the advantage that it can be completely encoded in λ& since records can be encoded by overloaded functions. The latter instead needs the introduction in the language of what in ML are called *constructor functions*: in standard ML the definition of `2DPoint` would correspond to:

$$\textbf{datatype}\ \mathit{2DPoint} = in^{2DPoint}\ \textbf{of}\ \{x : int, y : int\}$$

Consider now subtyping. Suppose that you have to encode two classes whose names are A and B in the case (a) the encoding of their objects will have type $\langle\!\langle is_a\colon A;\ x_1\colon T_1; \dots;\ x_n\colon T_n\rangle\!\rangle$ and $\langle\!\langle is_a\colon B; y_1\colon U_1; \dots; y_m\colon U_m\rangle\!\rangle$ respectively. Now if $\langle\!\langle is_a\colon A;\ x_1\colon T_1; \dots;\ x_n\colon T_n\rangle\!\rangle \leq \langle\!\langle is_a\colon B; y_1\colon U_1; \dots; y_m\colon U_m\rangle\!\rangle$ this automatically implies that the condition for refinement on the instance variables is respected (i.e. $\langle\!\langle x_1\colon T_1; \dots;\ x_n\colon T_n\rangle\!\rangle \leq \langle\!\langle y_1\colon U_1; \dots; y_m\colon U_m\rangle\!\rangle$). In the case (b) instead if one set $A \leq B$ then it has to check separately that the condition is respected; in other terms $in^A\colon \langle\!\langle x_1\colon T_1; \dots;\ x_n\colon T_n\rangle\!\rangle \to A$ and $in^B\colon \langle\!\langle y_1\colon U_1; \dots; y_m\colon U_m\rangle\!\rangle \to B$ are well-typed only if $\langle\!\langle x_1\colon T_1; \dots;\ x_n\colon T_n\rangle\!\rangle \leq \langle\!\langle y_1\colon U_1; \dots; y_m\colon U_m\rangle\!\rangle$.

For λ_object we choose to use the solution (b) for, even if it needs the introduction of new operations and new typing rules, it has the advantage that, as in our toy language, the type of an object is its class. Thus types will be conserved during the translation from the toy language to λ_object. Furthermore the operational semantics of λ_object will be simplified. Henceforth we will not distinguish among the terms "tag", "atomic type" and "class-name" since in λ_object they coincide.

To resume, in λ_object objects are "tagged terms" of the form in^A(M) where A is the tag and M represents the internal state. When we have an overloaded application $M \bullet N$ we first reduce M to a term $(M_1 \& M_2)$ and N to a tagged term, and then we perform the branch selection according to the obtained tag, that is the name of the class of the object. The selected method must be able to access the instance variables of the object, i.e. to get inside the *in* construct. To this purpose we use a function denoted *out* that composed with *in* gives the identity.

Pretypes

We use A and B (possibly subscripted) to denote atomic types.

$$T \ ::= \ A \mid T \times T \mid T \to T \mid \{(A_1 \times \dots \times A_{m_1}) \to T_1, \dots, (B_1 \times \dots \times B_{m_n}) \to T_n\} \qquad {}_{n,m_i \geq 1}$$

Terms

Here we define the raw terms of the language, i.e. terms that have not been type checked yet. Terms are composed by an expression preceded by a (possibly empty) suite of declarations. We use the metavariable M to range over expressions and P to range over terms:

$$\begin{array}{rcl} M & ::= & x^T \mid \lambda x^T.M \mid M \cdot M \mid \varepsilon \mid M\&^T M \mid M \bullet M \\ & \mid & <M\,,M> \mid \pi_1(M) \mid \pi_2(M) \mid \mu x^T.M \\ & \mid & \mathbf{coerce}^A(M) \mid \mathbf{super}^A(M) \mid in^A(M) \mid out^A(M) \end{array}$$

$$P \ ::= \ M \mid \mathbf{let}\ A \leq A_1, \ldots, A_n\ \mathbf{in}\ P \mid \mathbf{let}\ A\ \mathbf{hide}\ T\ \mathbf{in}\ P$$

Declarations cope with atomic types: they can be used to define the subtyping relation on atomic types and to declare a new atomic type by associating to it a representation type (i.e. the type of the internal state). More precisely the declaration **let** A **hide** T **in** P declares the atomic type A and associates it to the type T used for its representation. This declaration defines two constructors $in^A{:}T \to A$ and $out^A{:}A \to T$ which form a retraction pair from T to A.

Tagged values

We have to be a little more precise about tagged values: a tagged value is everything an overloaded function can perform its selection on. Thus it can be an object of the form in^A(M) but also the coercion of an object, the super of an object and, since we have multiple dispatch, a tuple of objects. Thus a tag is either an atomic type or a product of atomic types. We use the metavariable D to range over tags; tagged values are ranged over by G^D where D is the tag.

$$G^D ::= \ in^D(M) \mid \mathbf{coerce}^D(M) \mid \mathbf{super}^D(M) \mid <G_1^{A_1}, G_2^{A_2}, \ldots, G_n^{A_n}>$$

In the last case of the production above D is $(A_1 \times \ldots \times A_n)$

Operational Semantics

We define the *values* of λ_object, i.e. those terms which are considered as results; values are ranged over by G.

$$G \ ::= \ x \mid (\lambda x^T.M) \mid \varepsilon \mid (M_1 \&^T M_2) \mid <G_1\,, G_2> \mid \mathbf{coerce}^A(M) \mid \mathbf{super}^A(M) \mid in^A(M)$$

The operational semantics for λ_object is given by the reduction $\Rightarrow$; this reduction includes a type constraint system[5] C that is built along the reduction by the declarations (**let** $A \leq A_1 \ldots A_n$ **in** P) and that is used in the rule(s) for the selection of the branch. In the following, we use $\circ$ to denote either $\cdot$ or $\bullet$ and $\overline{D}$ to denote the $\min_{i=1..n}\{D_i | C \vdash D \leq D_i\}$

[5] At this stage it would be more correct to call it a "tag constraint system"

Axioms

$$(C\ ,\ \pi_i(<G_1,G_2>))\ \Rightarrow\ (C\ ,\ G_i) \qquad \text{i=1,2}$$
$$(C\ ,\ out^{A_1}(in^{A_2}(M)))\ \Rightarrow\ (C\ ,\ M)$$
$$(C\ ,\ out^{A_1}(\mathbf{coerce}^{A_2}(M)))\ \Rightarrow\ (C\ ,\ out^{A_1}(M))$$
$$(C\ ,\ out^{A_1}(\mathbf{super}^{A_2}(M)))\ \Rightarrow\ (C\ ,\ out^{A_1}(M))$$
$$(C\ ,\ \mu x.M)\ \Rightarrow\ (C\ ,\ M[x:=\mu x.M])$$
$$(C\ ,\ (\lambda x.M)\cdot N)\ \Rightarrow\ (C\ ,\ M[x:=N])$$
$$(C\ ,\ (M_1\&^{\{D_1\to T_1,\ldots,D_n\to T_n\}}M_2)\bullet G^D)\ \Rightarrow\ (C\ ,\ M_1\bullet G^D) \qquad \text{if } D_n\neq\overline{D}$$
$$(C\ ,\ (M_1\&^{\{D_1\to T_1,\ldots,D_n\to T_n\}}M_2)\bullet G^D)\ \Rightarrow\ (C\ ,\ M_2\cdot G^D) \qquad \text{if } D_n=\overline{D} \text{ and } G^D\not\equiv\mathbf{super}^D(M)$$
$$(C\ ,\ (M_1\&^{\{D_1\to T_1,\ldots,D_n\to T_n\}}M_2)\bullet G^D)\ \Rightarrow\ (C\ ,\ M_2\cdot M) \qquad \text{if } D_n=\overline{D} \text{ and } G^D\equiv\mathbf{super}^D(M)$$
$$(C\ ,\ \mathbf{let}\ A\leq A_1\ldots A_n\ \mathbf{in}\ P)\ \Rightarrow\ (C\cup(A\leq A_1)\cup\ldots\cup(A\leq A_n)\ ,\ P)$$
$$(C\ ,\ \mathbf{let}\ A\ \mathbf{hide}\ T\ \mathbf{in}\ P)\ \Rightarrow\ (C\ ,\ P)$$

Context Rules

$$\frac{(C\ ,\ M)\Rightarrow(C\ ,\ M')}{(C\ ,\ <M,N>)\Rightarrow(C\ ,\ <M',N>)} \qquad \frac{(C\ ,\ M)\Rightarrow(C\ ,\ M')}{(C\ ,\ <G,M>)\Rightarrow(C\ ,\ <G,M'>)}$$

$$\frac{(C\ ,\ M)\Rightarrow(C\ ,\ M')}{(C\ ,\ \pi_i(M))\Rightarrow(C\ ,\ \pi_i(M'))} \qquad \frac{(C\ ,\ M)\Rightarrow(C\ ,\ M')}{(C\ ,\ out^A(M))\Rightarrow(C\ ,\ out^A(M'))}$$

$$\frac{(C\ ,\ M)\Rightarrow(C\ ,\ M')}{(C\ ,\ M\circ N)\Rightarrow(C\ ,\ M'\circ N)} \qquad \frac{(C\ ,\ M)\Rightarrow(C\ ,\ M')}{(C\ ,\ (N_1\&N_2)\bullet M)\Rightarrow(C\ ,\ (N_1\&N_2)\bullet M')}$$

The semantics for pairs is the standard one. Three axioms and a rule describe the behavior of *out* and give it access to the internal state of an object. Functional application is implemented by call-by-name; anyway, this is not a necessary condition and the call-by-value would fit as well.

The three axioms and two rules for overloaded functions deserve more attention: in an overloaded application we first reduce the function (the term on the left) to an &-term and then its argument to a tagged value; then the reduction is performed according to the index of the &-term. In a sense, we perform a "call-by-tagged-value" (but for well-typed programs this notion coincides with the usual call-by-value: see corollary 5.2.5). It is worth noting that this selection does not use types: no type checking is performed, only a match of tags and some constraints is done; indeed, we still do not have any "type" here, but some tags indexing the terms. Note the difference when the tagged value is a super: in that case the argument of the super is passed to the selected branch instead of the whole tagged value.

Finally, the declaration ($\mathbf{let}\ \ A\leq A_1\ldots A_n\ \ \mathbf{in}\ \ P$) modifies the type constraints in which to evaluate the body P, while ($\mathbf{let}\ \ A\ \mathbf{hide}\ T\ \mathbf{in}\ P$) serves only to the type checker, and thus, operationally, is simply discarded.

Programs and type errors

The operational semantics above is given for untyped terms. Now we define which terms are the programs of λ_object and when a reduction ends by a type error.

Definition 5.2.1 A program in λ_object is a closed term P different from ε. □

We use the notation $P \Rightarrow P'$ to say that $(C, P) \Rightarrow (C', P')$ for some C and C' and we denote by $\overset{*}{\Rightarrow}$ the reflexive and transitive closure of $\Rightarrow$. Given a term M, we say that it is in *normal form* iff it does not exist N such that $M \Rightarrow N$. Let P be a closed term in normal form. If P is not a value then it is always possible to use the context rules of the operational semantics to decompose P to find the *least* subterm which is not a value and where the reduction is stuck. Let call this subterm the *critical subterm* of P. For example consider the following term:

$$(\,(M_1\&M_2) \bullet ((\mathbf{super}^A(M)) \cdot (N))\,) \quad \cdot \quad (\,M'\,)$$

This term is in normal form. Indeed, since it is an application we first try to reduce $((M_1\&M_2) \bullet ((\mathbf{super}^A(M)) \cdot (N)))$; then for the sixth context rule we try to reduce $(\mathbf{super}^A(M)) \cdot (N)$; again for the fifth context rule we try to reduce $(\mathbf{super}^A(M))$; but it is a value different from a λ-abstraction and we are stuck. Thus, in this case, the critical subterm is $(\mathbf{super}^A(M)) \cdot (N)$. Note that the critical subterm (of a closed normal non-value term) always exists and is unique, since it is found by an algorithm which is deterministic (since the operational semantics is deterministic) and terminating (since the size of the term at issue always decreases).

Definition 5.2.2 [type-error] Let P be a program. If $P \overset{*}{\Rightarrow} P'$, P' is in normal form and it is not a value then we say that P *produces a type error.* Furthermore if the critical subterm of P' is of the form $((M_1\&^T M_2) \bullet G^D)$ then we say that P *produces an "undefined method" type error.* □

The "undefined method" error is raised when we try to reduce an overloaded application of a &-term to a tagged value, and $\overline{D}$ (i.e. $\min_{i=1..n}\{D_i | C \vdash D \leq D_i\}$) is not defined. This means that it is not possible to select a branch for the object passed to the function. This can be due either because the set $\{D_i | D \leq D_i\,,\; i = 1..n\}$ is empty or because it has no minimum. In object-oriented terms the former case means that a wrong message has been sent to the object and in the latter that the conditions on multiple inheritance have not been respected.

5.2.1 The type system

We have defined programs and how to compute them; then we have singled out those computation that produce a "type error". Now we have to justify the use of the adjective *type* in front of the word "error". To this purpose we define a type system for the raw terms, so that the well-typed programs will not produce these errors. The complete definitions of this section are summarized in appendix B.

Types

As in the case of $\lambda\&$-calculus and of our toy language we first define an order on the pretypes and then we select among them those that satisfy the conditions for covariance, multiple inheritance and input type uniqueness. The subtyping relation on pretypes and the good formation for types are exactly the same as those defined for our toy language in section 1.2 and by the definition 5.1.3, with the only modification that the set of atomic types is relative to a program and it is formed by all the pretypes that have been declared by a **let** ... **hide** definition

Definition 5.2.3

1. $A \in_{C,S}$ **Types** for each $A \in dom(S)$
2. if $T_1, T_2 \in_{C,S}$ **Types** then $T_1 \to T_2 \in_{C,S}$ **Types** and $T_1 \times T_2 \in_{C,S}$ **Types**
3. if for all $i, j \in I$
   - **(a)** $(D_i, T_i \in_{C,S}$ **Types**$)$
   - **(b)** if $C \vdash D_i \leq D_j$ then $C \vdash T_i \leq T_j)$
   - **(c)** for each maximal type D in $LB_C(\{D_i, D_j\})$ there exists $h \in I$ such that $D_h = D$
   - **(d)** if $i \neq j$ then $D_i \neq D_j$

   then $\{D_i \to T_i\}_{i \in I} \in_{C,S}$ **Types**

□

Type checking rules

The type checking rules are parametric in a type constraint system C and a function S from atomic types to types. These are used respectively to store the type constraints and the implementation types defined in the declarations; this is performed by the following rules

[NewType]
$$\frac{C\,,\ S[A \leftarrow T] \vdash P\colon U}{C, S \vdash \textbf{let } A \textbf{ hide } T \textbf{ in } P\colon U}$$
$A \notin dom(S), T \in_{C,S}$ **Types** and T not atomic

[Constraint]
$$\frac{C \cup (A \leq A_i)_{1=1..n}, S \vdash P\colon T}{C, S \vdash \textbf{let } A \leq A_1, \ldots, A_n \textbf{ in } P\colon T}$$
if $C \vdash S(A) \leq S(A_i)$ and A do not appear in C

In the [NewType] rule we require that the representation type of a class is not another class; this is very reasonable, for the new atomic type would be completely equivalent to the one of its representation, but it would require a further *in* and *out* to access the internal state. In the last rule we require that A does not appear in any type constraint. In this way the ordering on atomic types is defined stepwise in the top-down sense. In this way the subtyping relation forms a dag. [6] In this case we do not require a partial lattice for atomic types as for λ&: in section 4.1.1 we showed that this would be too strong a condition for multiple inheritance; thus we allow every order.

Once more, even if we have an order on atomic types, what we obtain at higher types is only a preorder: indeed consider two atomic types A and B with $B \leq A$. Then $\{A \to A\} \leq \{A \to A, A \to B\} \leq \{A \to A\}$; thus antisymmetry does not hold; however we do not need to redefine the various notions of *inf*, *min* etc as we did in the footnotes of chapters 2 and 4 since these notions are used only in the selection of the branch which is performed on tags; and tags form an order (antisymmetry holds).

We want to interpret the construct `extend`; in λ& we can only add a new branch to an overloaded function but we cannot replace an existing branch by another with the same

[6] Equivalently we could have defined C so that to satisfy this property.

input type. However in section 4.1.2 we showed how modify λ& to obtain it. Therefore we adopt the same discipline for λ_object : the rules [TAUT], [→ INTRO], [→ ELIM$_{(\leq)}$], [TAUT$_\varepsilon$], [{}ELIM] for λ_object are the same as in λ& (with the obvious modifications to consider C and S). The only rule we have to change is [{}INTRO] that we replace by [{}INTRO+] of section 4.1.2

$$[\{\}\textrm{INTRO+}]\quad \frac{C,S \vdash M\colon W_1 \leq \{U_i \to V_i\}_{i\in I} \quad C,S\vdash N\colon W_2 \leq U \to V}{C,S \vdash (M\&^{\{U_i\to V_i\}_{i\in I}\oplus(U\to V)}N)\colon \{U_i \to V_i\}_{i\in I} \oplus (U \to V)} \quad \{U_i \to V_i\}_{i\in I} \oplus (U\to V) \in_{C,S} \textbf{Types}$$

The rules for the expressions that do not belong to the syntax of λ& are:

$$[\textrm{COERCE}]\quad \frac{C,S\vdash M\colon B}{C,S\vdash \textbf{coerce}^A(M)\colon A} \quad C\vdash B\leq A \text{ and } A\in_{C,S} \textbf{Types}$$

$$[\textrm{SUPER}]\quad \frac{C,S\vdash M\colon B}{C,S\vdash \textbf{super}^A(M)\colon A} \quad C\vdash B\leq A \text{ and } A\in_{C,S} \textbf{Types}$$

$$[\textrm{IN}]\quad \frac{C,S\vdash M\colon T}{C,S\vdash \mathit{in}^A(M)\colon A} \quad C\vdash T\leq S(A) \text{ and } A\in_{C,S} \textbf{Types}$$

$$[\textrm{OUT}]\quad \frac{C,S\vdash M\colon B}{C,S\vdash \mathit{out}^A(M)\colon S(A)} \quad C\vdash B\leq A \text{ and } A\in_{C,S} \textbf{Types}$$

Note that an atomic type A can be used in an expression like $\textbf{coerce}^A$, $\textbf{super}^A$ and so on, only if $A \in_{C,S}$ **Types**, i.e. it has been previously defined by a **let_hide** declaration.

5.2.2 Some results

Proposition 5.2.4 *Let $M\colon T$; if M is closed and in normal form then M is a value.*

Proof. The proof is obtained by induction on M. □

A consequence of this proposition is the following corollary which justifies the rules for the overloaded application in the operational semantics:

Corollary 5.2.5 *If a program is in normal form and it is typed by a (possibly unary) product of atomic types, then it is a tagged value.*

Recall that it is not possible to reduce inside a λ-abstraction. Therefore if in the evaluation of a program we reduce a term of the form $M \bullet N$, then in particular N must be closed. To perform the selection of a branch (the $\beta_{\&}$-reduction) N must also be a value; thus, by the corollary above it must be a tagged value. Therefore in a well-typed program overloaded application is implemented by the usual call-by-value, since the only values allowed as arguments by the type checker are tagged values.

Lemma 5.2.6 *(substitution lemma) Let $C,S \vdash M\colon T$, $C,S\vdash N\colon T'$ and $C\vdash T'\leq T$; then $C,S\vdash M[x^T := N]\colon U'$, where $C\vdash U'\leq U$*

Proof. By induction on M. The only difficult case is $M \equiv M_1 \bullet M_2$, whose proof follows the pattern of the corresponding case in the next theorem. □

Theorem 5.2.7 (Subject Reduction) *Let $C, S \vdash P{:}T$; if $(C,P) \overset{*}{\Rightarrow} (C',P')$ then $C', S \vdash P'{:}T'$ and $C' \vdash T' \leq T$.*

Proof. The proof is a generalization of the one for the λ&-calculus in chapter 2. It consists in an induction on P where we use the substitution lemma above. It suffices to prove the theorem for ⇒; the thesis follows by a simple induction on the number of steps of the reduction. Thus, we proceed by induction on the structure of P. When P is a value then the thesis is trivially satisfied. When P is of the form (**let** ... **in** P') or of the form $\pi_i(M)$, then the proof is a straightforward use of the induction hypothesis. The remaining cases are (in the rest of the proof we omit C and S since they do not change):

$P \equiv out^A(M)$. Where $M{:}A' \leq A$. The only case of reduction is that $M \Rightarrow M'$ and $P' \equiv out^A(M')$; but from the induction hypothesis it follows that $M'{:}B \leq A' \leq A$; thus also P' is well-typed and possess the same type as P.

$P \equiv M_1 \cdot M_2$ where $M_1{:}U \to T$ and $M_2{:}W \leq U$. We have two subcases:

1. $M_1 \Rightarrow M_1'$, then by induction hypothesis $M_1'{:}U' \to T'$ with $U \leq U'$ and $T' \leq T$. Since $W \leq U \leq U'$, then by rule $[\to\text{ELIM}_{(\leq)}]$ we obtain $M_1' M_2{:}T' \leq T$
2. $M_1 \equiv \lambda x^U.M_3$ and $M \Rightarrow M_3[x := M_2]$, with $M_3{:}T$. Thus, by Lemma 5.2.6, $M_3[x{:=}M_2]{:}T'$ with $T' \leq T$.

$P \equiv M_1 \bullet M_2$ where $M_1{:}\{D_i \to U_i\}_{i \in I}$ and $M_2{:}D$.

Let $D_h = \min_{i \in I}\{D_i | D \leq D_i\}$. Thus $T = U_h$. We have three subcases:

1. $M_1 \Rightarrow M_1'$ then by induction $M_1'{:}\{D_j' \to U_j'\}_{j \in J}$ with $\{D_j' \to U_j'\}_{j \in J} \leq \{D_i \to U_i\}_{i \in I}$ Let $D_k' = \min_{j \in J}\{D_j' | D \leq D_j'\}$. Thus $M_1' \bullet M_2{:}U_k'$. Therefore we have to prove that $U_k' \leq U_h$
   Since $\{D_j' \to U_j'\}_{j \in J} \leq \{D_i \to U_i\}_{i \in I}$, then for all $i \in I$ there exists $j \in J$ such that $D_j' \to U_j' \leq D_i \to U_i$. For $i = h$ we chose a certain $\tilde{h} \in J$ which satisfies this condition that is:
   $$D_{\tilde{h}}' \to U_{\tilde{h}}' \leq D_h \to U_h \tag{5.1}$$
   We now have the following inequalities:
   $$D \leq D_h \tag{5.2}$$
   by hypothesis, since $D_h = \min_{i \in I}\{D_i | D \leq D_i\}$;
   $$D_h \leq D_{\tilde{h}}' \tag{5.3}$$
   follows from (5.1);
   $$D \leq D_{\tilde{h}}' \tag{5.4}$$

follows from (5.2) and (5.3);

$$U'_{\tilde{h}} \leq U_h \tag{5.5}$$

follows from (5.1);

$$D'_k \leq D'_{\tilde{h}} \tag{5.6}$$

by (5.4), since $D'_{\tilde{h}}$ belongs to a set with D'_k as least element;

$$U'_k \leq U'_{\tilde{h}} \tag{5.7}$$

follows from (5.6) and the covariance rule on $\{D'_j \to U'_j\}_{j\in J}$
Finally, by (5.5) and (5.7), one has that $U'_k \leq U_h$

2. $M_2 \Rightarrow M'_2$ then by induction hypothesis $M'_2 \colon D'$ with $D' \leq D$. Let $D_k = \min_{i\in I}\{D_i | D' \leq D_i\}$. Thus $M_1 \bullet M'_2 \colon U_k$. Since $D' \leq D \leq D_h$ then $D_k \leq D_h$; thus, by the covariance rule in $\{D_i \to U_i\}_{i\in I}$, we obtain $U_k \leq U_h$.
3. $M_1 \equiv (N_1 \& N_2)$ and M_2 is a tagged value. Then we have three cases, that is $M \Rightarrow (N_1 \bullet M_2)$ (case $D_h \neq D_n$) or $M \Rightarrow (N_2 \cdot M_2)$ (case $D_h = D_n$ and M_2 different form super) or $M \Rightarrow (N_2 \cdot M_3)$ (case $D_h = D_n$ and $M_3 \equiv \mathbf{super}^D(M_2)$). According to the case it easy to use [{}ELIM] or [→ELIM$_{(\leq)}$] or [→ELIM$_{(\leq)}$] and [SUPER] to show that the terms have type U_h or smaller.

□

Proposition 5.2.8 *If $P \Rightarrow P'$ and P is closed then also P' is closed*

Proof. A simple induction on the rules of the operational semantics. □

Corollary 5.2.9 *Let P be a well-typed program. If $P \overset{*}{\Rightarrow} P'$ and P' is in normal form then P' is a value*

Proof. By theorem 5.2.7 P' is well-typed and by proposition 5.2.8 it is closed. The thesis follows from proposition 5.2.4. □

This corollary states that well-typed programs reduce to values, and thus do not produce type errors.

5.3 Translation

As we already said, we do not give a direct semantics to the toy language. Instead we translate its programs into λ_object.

The key theorem of this section states that a well typed program is translated in a well typed term of λ_object ; this result validates the algorithm of type-checking we have given for the toy object-oriented language in section 5.1.2, since it assures that type errors can never occur during the computation of well typed programs.

We split the definition of the interpreter in three parts: we first translate programs where methods are neither mutually recursive nor multi-methods; then by slight modifications we introduce also multi-methods and finally, in the third subsection, recursion too.

5.3.1 Simple methods without recursion

We first give the intuitive translation of the object-oriented commands of the language:

- A message is an identifier of an overloaded function; thus it is translated in a variable possessing a (raw) overloaded type; i.e. $[\![m]\!] = m^{\{A_i \to T_i\}_{i \in I}}$ where $\{A_i | i \in I\}$ is the set of the classes where the message `m` has been defined, and the T_i's are the corresponding types appearing in the interfaces.
- Message passing is the application of an overloaded function; i.e. $[\![$ `[`exp_0 exp $exp_1, \ldots, exp_n$ `]`$]\!] = [\![exp]\!] \bullet [\![(exp_0, exp_1, \ldots, exp_n)]\!]$
- In the definition of a method, `self` represents the receiver of the message which invoked the method. Thus we translate a method *msg*=*exp* into $\lambda self^A.[\![exp]\!]$, where A is the current class. This will form a branch of the overloaded function denoted by the (translation of the) message *msg*.
- `new(`A`)` defines a value of type A. More exactly it defines $in^A(r)$ where r is the record value containing the initial values of the instance variables of the class A.
- `update` unpacks *self* in its representation (record) type, modifies its value (i.e. the internal state) and packs it again in its original type. Thus for example $[\![$`(update {`$x = 3$`})`$]\!] = in^A(\langle out^A(self^A) \leftarrow x = 3\rangle)$; again A is the current class.
- `super[`A`](`exp`)` and `coerce[`A`](`exp`)` are respectively translated into $\mathbf{super}^A([\![exp]\!])$ and $\mathbf{coerce}^A([\![exp]\!])$.
- The operation `extend` corresponds to adding a branch to an overloaded function. It has the following intuitive interpretation
  $[\![$ `extend` A $m = exp$ `[[...]] in` $exp']\!] = (\mathbf{let}\ m = (m \& \lambda self^A.[\![exp]\!])\ \mathbf{in}\ [\![exp']\!])$.
- Finally we have the most complex construct: the class definition. By a class definition one defines a new atomic type, a set of type constraints on this atomic type and some branches of overloaded function. The intuitive interpretation of, say, (`class` A `is` A_1`,`A_2 $\{x$`:Int=3`$\}$ m `=` exp `[[` $m : T$ `]] in` p) is:

  $\mathbf{let}\ A\ \mathbf{hide}\ \langle\!\langle x : Int \rangle\!\rangle\ \mathbf{in}$

  $\mathbf{let}\ A \leq A_1, A_2\ \mathbf{in}$

  $\mathbf{let}\ m = (m \& \lambda self^A.[\![exp]\!])\ \mathbf{in}\ [\![p]\!]$

  Of course the initial value 3 of x must be recorded during the translation so that this value could be used in the translation of `new(`A`)`.

Unfortunately the formal interpretation is not so smooth. Most problems derive from the fact that in λ_object the variables are typed. So when we translate a set of methods into an overloaded function, we have to concatenate branches so that the resulting term has the required overloaded type.

Formally let $\mathcal{L}$ be the set of the programs of the toy-language; we define the translation from $\mathcal{L}$ to **Terms** (the set of the *raw* terms of λ_object) using three functions. The first is the function which describes the translation itself:

$$\Im[\![_]\!] : \mathcal{L} \to Envs \to InitState \to \mathbf{AtomicTypes} \to \mathbf{Terms}$$

Where:

$Envs = Vars \to \mathbf{RawTypes}$	This parameter records the type of the identifiers. It is ranged over by the metavariable Γ.
$InitState = ClassNames \to RecordValues$	This parameter stores the initial value of the instance variables of each class: it is used in the interpretation of `new`. It is ranged over by the metavariable I.
AtomicTypes	This parameter is the *current class* which is used in the translation of a method.

Therefore $\Im[\![p]\!]_{\Gamma\, I\, A}$ is the term of λ_object that translates the program p.

The definition of $\Im$ is given in term of two auxiliary functions $\mathcal{M}$ and $\mathcal{T}$: $\mathcal{M}[\![p]\!](m)$ returns the (overloaded) term associated to the message m by the definitions in p; $\mathcal{T}[\![p]\!](m)$ returns the (raw) type that indexes the variable (translation of) m. Of course, if p is well typed we expect that $\mathcal{M}[\![p]\!](m)\colon \mathcal{T}[\![p]\!](m)$.

It is necessary to introduce these auxiliary functions in order to overcome one of the major drawbacks of λ&. Suppose we have three classes A, B and C with C defined by multiple inheritance from A and B ($C \leq A, B$). Suppose also that A and B can respond to the same message m; then by the condition of multiple inheritance one has also to define a branch for m with input type C. In object oriented languages, as in our toy language, the logical order is to define first the branches for A and B and then at the moment of the definition of C to append the new branch for C. Thus the definition of m would be of the form

$$m \equiv (\varepsilon \,\&^{\{A\to T_1\}} \lambda self^A.M_1 \,\&^{\{A\to T_1, B\to T_2\}} \lambda self^B.M_2 \,\&^{\{A\to T_1, B\to T_2, C\to T_3\}} \lambda self^C.M_3) \tag{5.8}$$

This is very reasonable but unfortunately the term above is not well typed, since the second index $\{A \to T_1, B \to T_2\}$ is not a well formed type. In λ& the branch written to solve the ambiguity of multiple inheritance must always precede at least one of the branches of its direct ancestors. In the case above for example the following definition is well typed

$$m \equiv (\varepsilon \,\&^{\{A\to T_1\}} \lambda self^A.M_1 \,\&^{\{A\to T_1, C\to T_3\}} \lambda self^C.M_3 \,\&^{\{A\to T_1, C\to T_3, B\to T_2\}} \lambda self^B.M_2)$$

This problem can be framed in the more general problem of the definition of dynamic types. λ& completely lacks the notion of time, or better the order of the definition of types. Atomic types are given all at once, and there is no perception of the temporal dependence of type

definitions. Thus dynamic types cannot be modeled, and for this reason in our toy language all the class definitions have to precede the expression to execute. Actually we are working on the definition of a type system in which the types use time stamps, so that the definition of m as in (5.8) is well typed. The idea is that an expression with type $\{A \to T_1, B \to T_2\}$ has a well-formed type if all its sub-expressions use types that are older than the definition of C.

However for the moment we do not have time stamps; thus to translate our toy language we have to use the functions $\mathcal{M}$ and $\mathcal{T}$ that pre-scan the program to translate, and build the messages in the reverse way, from the latest method defined to the first one. Thus to translate a program we are obliged to scan it twice: once to construct methods by reading the definition in the reverse way, the other to translate the whole program.

Since the function $\mathcal{M}[\![_]\!]$ uses in its definition the function $\Im[\![_]\!]$, it needs the same parameters of $\Im$ in order to pass them to it; for $\mathcal{T}[\![_]\!]$ no parameter is needed. Therefore we have these formal definitions

Definition 5.3.1

$$\mathcal{T}[\![_]\!] : \mathcal{L} \to Vars \to \textbf{Types}$$

1. $\mathcal{T}[\![\texttt{class } B \texttt{ is } A_1\texttt{,}\dots\texttt{,}A_p\;\; r:R\;\; \texttt{m}_1\texttt{=}exp_1 \dots \texttt{m}_n\texttt{=}exp_n \texttt{[[m}_1\texttt{:}T_1 \dots \texttt{m}_n : T_n\texttt{]] in } p]\!](m) =$
   $= \begin{cases} \mathcal{T}[\![p]\!](m_j) \oplus \{B \to T_j\} & \text{for } m = m_j \\ \mathcal{T}[\![p]\!](m) & \text{else} \end{cases}$
2. $\mathcal{T}[\![_]\!]$ is the function which returns $\{\}$ in all the other cases.

□

Definition 5.3.2

$$\mathcal{M}[\![_]\!] : \mathcal{L} \to Envs \to InitState \to \textbf{AtomicTypes} \to Vars \to \textbf{Terms}$$

1. $\mathcal{M}[\![\texttt{class } B \texttt{ is } A_1\texttt{,}\dots\texttt{,}A_q\;\; r\texttt{:}R\;\; \texttt{m}_1\texttt{=}exp_1 \dots \texttt{m}_n\texttt{=}exp_n \texttt{[[m}_1\texttt{:}T_1 \dots \texttt{m}_n : T_n\texttt{]] in } p]\!]_{\Gamma\, I\, A}(m) =$
   $= \begin{cases} ((\mathcal{M}[\![p]\!]_{\Gamma'\, I\, A}(m_j))\&^{\mathcal{T}[\![p]\!](m_j)\oplus\{B\to T_j\}}\lambda self^B.\Im[\![exp_j]\!]_{\Gamma[self\leftarrow B]\, I[B\leftarrow r]\, B}) & \text{for } m = m_j \\ \mathcal{M}[\![p]\!]_{\Gamma\, I\, A}(m) & \text{else} \end{cases}$
2. $\mathcal{M}[\![_]\!]$ is the function which returns ε in all the other cases.

Where $\Gamma' = \Gamma[m_i \leftarrow \Gamma(m_i) \oplus \{B \to T_i\}]_{i=1..n}$. □

Note that in $\Im[\![exp_j]\!]_{\Gamma\, I[B\leftarrow r]\, B}$ we have used Γ instead of Γ' since we stated that the methods are not mutually recursive.

Definition 5.3.3 [Translation]

1. $[\![x]\!]_{\Gamma\, I\, A} = x^{\Gamma(x)}$
2. $[\![exp_1\texttt{(}exp_2\texttt{)}]\!]_{\Gamma\, I\, A} = [\![exp_1]\!]_{\Gamma\, I\, A}[\![exp_2]\!]_{\Gamma\, I\, A}$
3. $[\![\texttt{fn(}x\texttt{:}T\texttt{) => } exp]\!]_{\Gamma\, I\, A} = \lambda x^T.\Im[\![exp]\!]_{\Gamma[x\leftarrow T]\, I\, A}$
4. $[\![\texttt{let } x\texttt{:}T = exp_1 \texttt{ in } exp_2]\!]_{\Gamma\, I\, A} = (\lambda x^T.\Im[\![exp_2]\!]_{\Gamma[x\leftarrow T]\, I\, A})([\![exp_1]\!]_{\Gamma\, I\, A})$
5. $[\![\texttt{(}exp_1\texttt{,}\dots\texttt{,}exp_n\texttt{)}]\!]_{\Gamma\, I\, A} = < [\![exp_1]\!]_{\Gamma\, I\, A}, \dots, [\![exp_n]\!]_{\Gamma\, I\, A} >$
6. $[\![\texttt{fst(}exp\texttt{)}]\!]_{\Gamma\, I\, A} = \pi_1([\![exp]\!]_{\Gamma\, I\, A})$

7. $[\![\texttt{snd(}exp\texttt{)}]\!]_{\Gamma\, I\, A} = \pi_2([\![exp]\!]_{\Gamma\, I\, A})$
8. $[\![\texttt{new(}B\texttt{)}]\!]_{\Gamma\, I\, A} = in^B(I(B))$
9. $[\![\texttt{[}exp_0\ \ exp\ \ exp_1\texttt{,}\ \ldots\texttt{,}\ \ exp_n\texttt{]}]\!]_{\Gamma\, I\, A} = [\![exp]\!]_{\Gamma\, I\, A} \bullet [\![\texttt{(}exp_0\texttt{,}exp_1\texttt{,}\ldots\texttt{,}exp_n\texttt{)}]\!]_{\Gamma\, I\, A}$
10. $[\![\texttt{super[}B\texttt{](}exp\texttt{)}]\!]_{\Gamma\, I\, A} = \mathbf{super}^B([\![exp]\!]_{\Gamma\, I\, A})$
11. $[\![\texttt{coerce[}B\texttt{](}exp\texttt{)}]\!]_{\Gamma\, I\, A} = \mathbf{coerce}^B([\![exp]\!]_{\Gamma\, I\, A})$
12. $[\![\texttt{self}]\!]_{\Gamma\, I\, A} = self^A$
13. $[\![\texttt{self.}\ell]\!]_{\Gamma\, I\, A} = (out^A(self^A)).\ell$
14. $[\![\texttt{(update}\ r\texttt{)}]\!]_{\Gamma\, I\, A} = in^A(\langle out^A(self^A) \leftarrow \ell_1 = [\![exp_1]\!]_{\Gamma\, I\, A} \ldots \leftarrow \ell_n = [\![exp_n]\!]_{\Gamma\, I\, A}\rangle)$
    where $r \equiv \{\ell_1 = exp_1; \ldots; \ell_n = exp_n\}$
15. $\Im[\![\ \texttt{extend}\ B\ \texttt{m}_1\texttt{=}exp_1\texttt{...;m}_n\texttt{=}exp_n\ \texttt{[[m}_1\texttt{:}T_1\texttt{;...;m}_n\texttt{:}T_n\texttt{]]}\ \texttt{in}\ exp]\!]_{\Gamma\ I\ A} =$
    $(\lambda m_1^{\Gamma(m_1)\oplus\{B\to T_1\}} \ldots \lambda m_n^{\Gamma(m_n)\oplus\{B\to T_n\}}.\Im[\![exp]\!]_{\Gamma'\, I\, A})$
    $(m_1^{\Gamma(m_1)} \&^{\Gamma(m_1)\oplus\{B\to T_1\}} \lambda self^B.\Im[\![exp_1]\!]_{\Gamma\, I\, B}) \cdots (m_n^{\Gamma(m_n)} \&^{\Gamma(m_n)\oplus\{B\to T_n\}} \lambda self^B.\Im[\![exp_n]\!]_{\Gamma\, I\, B})$

In the last rule $\Gamma' = \Gamma[m_i \leftarrow \Gamma(m_i) \oplus \{B \to T_i\}]$. □

It still remains to give the semantics of programs.
Let p the program `class` B `is` A_1`,...,`A_q r:R $\texttt{m}_1$=$exp_1 \ldots \texttt{m}_n$=$exp_n$`[[` $\texttt{m}_1$:$T_1 \ldots \texttt{m}_n : T_n$`]]` `in` p' then

$\Im[\![p]\!]_{\Gamma\ I\ A} =$
  let B **hide** R **in**
  let $B \leq A_1 \ldots A_q$ **in**
    $\Im[\![p']\!]_{\Gamma\, I[B\leftarrow r]\, A}[m_i^{(\mathcal{T}[\![p]\!](m_i))} := \mathcal{M}[\![p]\!]_{\Gamma\, I\, A}(m_i)]_{i=1..n}$

5.3.2 With multi-methods

Let us now add multi-methods. Intuitively we have to change only three things:

- The type of a message must take into account also the multi-methods, thus $[\![\ \texttt{m}\ ]\!] = m^{\{A_i \leadsto T_i\}_{i\in I}}$—note the use of $\leadsto$ in the place of $\to$—where again $\{A_i | i \in I\}$ is the set of the classes where the message `m` has been defined, and the T_i's are the corresponding types appearing in the interfaces.

- The method *msg*=*exp* is translated as before into $\lambda self^A.[\![exp]\!]$, if it is a normal method (A is the current class). If it is a multi-method then *exp* must be of the form `&`...`&`.... For example *exp* may be:

```
mesg = & fn(x1:C1; x2:C2) => exp1
       & fn(y1:C1; y2:C3) => exp2
       & fn(z:C2)         => exp3
```

  Then, using some pattern matching in the lambda calculus, the multi-method is translated into

  let *mesg* = (*mesg*
    $\&\lambda(self^A, x_1^{C_1}, x_2^{C_2}).[\![\texttt{exp1}]\!]$
    $\&\lambda(self^A, y_1^{C_1}, y_2^{C_3}).[\![\texttt{exp2}]\!]$
    $\&\lambda(self^A, z^{C_2}).[\![\texttt{exp3}]\!]$
)

- In the translation of `extend` we have to translate the multi-methods in the same way as above.

From a formal point of view this comes to modify the definitions of the previous section in this way:

Definition 5.3.4

$$\mathcal{T}[\![_]\!] : \mathcal{L} \to Vars \to \mathbf{Types}$$

1. $\mathcal{T}[\![$ `class` B `is` A_1`,...,`A_q $r:R$ $\texttt{m}_1$`=`$exp_1 \ldots \texttt{m}_n$`=`$exp_n$`[[` $\texttt{m}_1 : V_1 \ldots \texttt{m}_n : V_n$`]]` `in` $p]\!](m) =$
$= \begin{cases} \mathcal{T}[\![p]\!](m_j) \oplus \{B \leadsto V_j\} & \text{for } m = m_j \\ \mathcal{T}[\![p]\!](m) & \text{else} \end{cases}$
2. $\mathcal{T}[\![_]\!]$ is the function which returns $\{\}$ in all the other cases.

□

The definition of $\mathcal{M}[\![.]\!] : \mathcal{L} \to Envs \to InitState \to \mathbf{AtomicTypes} \to Vars \to \mathbf{Terms}$ gets quite harder:

Definition 5.3.5

- $\mathcal{M}[\![$ `class` B `is` A_1`,...,`A_q $r{:}R$ $\texttt{m}_1$`=`$exp_1 \ldots \texttt{m}_n$`=`$exp_n$`[[` $\texttt{m}_1{:}V_1 \ldots \texttt{m}_n{:}V_n$`]]` `in` $p]\!]_{\Gamma\, I\, A}(m) =$
  1. If $m \equiv m_j$ for some $j \in [1..n]$ and V_j is a raw type, then the definition is as before
  $$((\mathcal{M}[\![p]\!]_{\Gamma'\, I\, A}(m_j)) \&^{\mathcal{T}[\![p]\!](m_j) \oplus \{B \to V_j\}} \lambda self^B . \Im[\![exp_j]\!]_{\Gamma[self \leftarrow B]\, I[B \leftarrow r]\, B})$$
  2. If $m \equiv m_j$ for some $j \in [1..n]$ and $V_j \equiv \#\{D_i \to T_i\}_{i=1..h}$, then we are in the case of multi-method and exp_j must be of the following form:
  `& fn(`$x_1{:}D_1$`) =>` exp_{j_1}
  $\vdots$
  `& fn(`$x_h{:}D_h$`) =>` exp_{j_h}
  then $\mathcal{M}$ is defined in the following way:
  $(\cdots((\mathcal{M}[\![p]\!]_{\Gamma'\, I\, A}(m_j)$
  $\&^{\mathcal{T}[\![p]\!](m_j) \oplus \{B \times D_{\sigma(1)} \to T_{\sigma(1)}\}} \lambda(self^B, x_{\sigma(1)}^{D_{\sigma(1)}}).\Im[\![exp_{j_{\sigma(1)}}]\!]_{\Gamma[self \leftarrow B]\, I[B \leftarrow r]\, B})$
  $\vdots$
  $\&^{(\mathcal{T}[\![p]\!](m_j) \oplus \ldots \oplus \{B \times D_{\sigma(h-1)} \to T_{\sigma(h-1)}\}) \oplus \{B \times D_{\sigma(h)} \to T_{\sigma(h)}\}}$
  $\lambda(self^B, x_{\sigma(h)}^{D_{\sigma(h)}}).\Im[\![exp_{j_{\sigma(h)}}]\!]_{\Gamma[self \leftarrow B]\, I[B \leftarrow r]\, B})$
  Where $\Gamma' = \Gamma[m_i \leftarrow \Gamma(m_i) \oplus \{B \leadsto T_i\}]_{i=1..n}$ and σ is the permutation described in section 4.3.1.
  3. Else the definition of $\mathcal{M}$ is $\mathcal{M}[\![p]\!]_{\Gamma\, I\, A}(m)$
- $\mathcal{M}[\![.]\!]$ is the function which returns ε in all the other cases.

□

Finally we have to modify the definition of $\Im$ for `extend`. We do not write it here since the modifications follow the pattern of those we have done in the definition above. This definition, however can be found in the case 14 of appendix D.

5.3.3 With recursive methods

We now give the translation in the case that methods can be defined mutually recursively. The only thing we have to change is the interpretation of the methods, and then apply it both to the translation of the class definition and the one of `extend`. Intuitively without multi-methods if we have:

`extend` B `with`
$m_1 = exp_1$
$m_2 = exp_2$
$\vdots$
$m_n = exp_n$
`[[ ...]] in` exp

this is translated into

let $(m_1, m_2, \ldots, m_n) = \mu(m_1, m2, \ldots, m_n).$
$((m_1 \& \lambda self^B.[\![exp_1]\!]) , (m_2 \& \lambda self^B.[\![exp_2]\!]) , \ldots , (m_n \& \lambda self^B.[\![exp_n]\!]))$
in $[\![exp]\!]$

Of course we have to put the right types to the variables and the ampersands, and to deal with multi-methods. Thus we have to change the definition of $\mathcal{M}$ in the following way:

Definition 5.3.6 Let p' denote the program `class` B `is` A_1`,...,`A_q $r{:}R$ `m`$_1$`=`$exp_1 \ldots$`m`$_n$`=`exp_n`[[` `m`$_1$`:`$V_1 \ldots$`m`$_n : V_n$`]] in` p; then

- $\mathcal{M}[\![p']\!]_{\Gamma\ I\ A}(m) = \begin{cases} \pi_j(M) & \text{for } m = m_j \\ \mathcal{M}[\![p]\!]_{\Gamma\ I\ A}(m) & \text{else} \end{cases}$

  Where $\Gamma' = \Gamma[m_i \leftarrow \Gamma(m_i) \oplus \{B \to T_i\}]_{i=1..n}$ and M has the following definition:

$$M \equiv \mu(m_1^{\mathcal{T}[\![p']\!](m_1)}, \ldots, m_n^{\mathcal{T}[\![p']\!](m_n)}).(M_1, \ldots, M_n)$$

  where

  1. If V_j is a raw type then

$$M_j \equiv ((\mathcal{M}[\![p]\!]_{\Gamma'\ I\ A}(m_j)) \&^{\mathcal{T}[\![p]\!](m_j) \oplus \{B \rightsquigarrow V_j\}} \lambda self^B.\Im[\![exp_j]\!]_{\Gamma'[self \leftarrow B]\ I[B \leftarrow r]\ B})$$

  2. If $j \in [1..n]$ and $V_j \equiv \#\{D_i \to T_i\}_{i=1..h}$, then exp_j must be of the following form:

     `& fn(`$x_1{:}\,D_1$`) =>` exp_{j_1}
     $\vdots$
     `& fn(`$x_h{:}\,D_h$`) =>` exp_{j_h}

     then M_j is defined in the following way:

     $(\cdots((\mathcal{M}[\![p]\!]_{\Gamma'\ I\ A}(m_j)$
     $\&^{\mathcal{T}[\![p]\!](m_j) \oplus \{B \times D_{\sigma(1)} \to T_{\sigma(1)}\}} \lambda(self^B, x_{\sigma(1)}^{D_{\sigma(1)}}).\Im[\![exp_{j_{\sigma(1)}}]\!]_{\Gamma'[self \leftarrow B]\ I[B \leftarrow r]\ B})$
     $\vdots$
     $\&^{(\mathcal{T}[\![p]\!](m_j) \oplus \ldots \oplus \{B \times D_{\sigma(h-1)} \to T_{\sigma(h-1)}\}) \oplus \{B \times D_{\sigma(h)} \to T_{\sigma(h)}\}}$
     $\lambda(self^B, x_{\sigma(h)}^{D_{\sigma(h)}}).\Im[\![exp_{j_{\sigma(h)}}]\!]_{\Gamma'[self \leftarrow B]\ I[B \leftarrow r]\ B})$

     where σ has the usual property.

- $\mathcal{M}[\![.]\!]$ is the function which returns ε in all the other cases.

□

Finally we have to make the same modifications in the interpretation of `extend`.

5.3.4 Correctness of the type-checking

We next prove that every well typed program of the toy-language is translated in a well typed term of λ_object . The semantics of the toy-language is given in terms of the translation we have just defined; also the notion of type error for the toy language comes from this translation: a program is type safe when its translation, if it stops, stops on a value[7]. Thus by the results of section 5.2.2 the translation of a well typed program is type safe, which means that the type checker for λ_object is correct.

Indeed we prove something stronger than the well typing of a term obtained by translating a well typed program: we prove that the translated program possesses the same type as its translation; note indeed that the types of the toy language are the same as those of λ_object.

Since the definition of $\Im[\![.]\!]$ is mutually recursive with $\mathcal{M}[\![.]\!]$ then the theorem must be proved mutually recursively with a theorem on $\mathcal{M}[\![.]\!]$. Thus the main theorem will be split in two propositions. But first we some auxiliary notation:

Notation 5.3.7 *We denote by C_p the set of type constraints declared in p, that is $C_p = \emptyset$ if p is an expression and $C_{(class\,A\,is\,A_1...A_n\,...\,in\,p')} = (A \leq A_1) \cup ... \cup (A \leq A_n) \cup C_{p'}$. We denote by S_p the stores of the internal states defined in p: again $S_p = \emptyset$ if p is an expression and $S_{(class\,A\,is...r:R...in\,p')} = [A \leftarrow R] \cdot S_{p'}$ (here $\cdot$ denotes simple juxtaposition)*

Theorem 5.3.8 *For every type constraint C, type environment Γ; for every $I \in InitState$ and $S : ClassNames \to$* **RecordTypes** *such that $I(A){:}\,S(A)$ (for every A atomic); if*

$$C; S; \Gamma \vdash p{:}\,T$$

then

1. *for all $m \in Vars$ $\qquad C \cup C_p; S \cdot S_p \vdash \mathcal{M}[\![p]\!]_{\Gamma\, I\, \Gamma(self)}(m){:}\,\mathcal{T}[\![p]\!](m)$*
2. *$C; S \vdash \Im[\![p]\!]_{\Gamma\, I\, \Gamma(self)}{:}\,T$*

Proof. See appendix D □

5.4 λ_object and λ&

In this last section of the chapter we show the exact correspondence between λ_object and λ& by presenting how the former can be encoded in one of the variants of λ& presented in chapter 4. We are not able to translate the whole λ_object; we have to restrain our attention to those programs that do not contain **super**. This was quite predictable since the introduction of **super** had required the modification of the rule $\beta_{\&}$.

[7]We tried to give an informal justification of this. More formally we should define type errors in the toy language and show that a type error in λ_object implies a type error in the toy language

The target language of this encoding will be $\lambda\&^{+}$+coerce. First of all we recall the encoding of surjective pairings given in section 2.5.1: we distinguish two isolated types P_1 and P_2 together with two constants $\pi_1 : P_1$ and $\pi_2 : P_2$ that we use to define the following encoding:

$$
\begin{array}{lcll}
(T_1 \times T_2) & \equiv & \{P_1 \rightarrow T_1, P_2 \rightarrow T_2\} & \\
 & & & \\
\pi_i(M) & \equiv & M \bullet \pi_i & \\
<M_1\,, M_2> & \equiv & (\varepsilon \& \lambda x^{P_1}.M_1 \& \lambda x^{P_2}.M_2) & \quad (\text{for } x^{P_i} \notin FV(M_i))
\end{array}
$$

We recall that the the rules of subtyping, typing and reduction are the *special cases* of the rules of λ& (and thus of $\lambda\&^{+}$+coerce) for the encoding.

5.4.1 The encoding of the types

We start by codifying the types of λ_object. Recall that in λ_object every atomic type is associated to a type used for its representation. This association is always relative to a program in which it is described. Thus given a well-typed program P we define

1. The set of atomic types defined in the program P:

$$
\mathcal{A}_P = \begin{cases} A \cup \mathcal{A}_{P'} & \text{if } P \equiv \textbf{let } A \textbf{ hide } T \textbf{ in } P' \\ \mathcal{A}_{P'} & \text{if } P \equiv \textbf{let } A \leq A_1, \ldots, A_n \textbf{ in } P' \\ \emptyset & \text{otherwise} \end{cases}
$$

2. The set of type constraints generated in the program P:

$$
\mathcal{C}_P = \begin{cases} (A \leq A_1) \cup \ldots \cup (A \leq A_n) \cup \mathcal{C}_{P'} & \text{if } P \equiv \textbf{let } A \leq A_1, \ldots, A_n \textbf{ in } P' \\ \mathcal{C}_{P'} & \text{if } P \equiv \textbf{let } A \textbf{ hide } T \textbf{ in } P' \\ \emptyset & \text{otherwise} \end{cases}
$$

3. The function that for every atomic type A in $\mathcal{A}_P$ returns the representation type associated in P.

$$
\mathcal{S}_P = \begin{cases} \mathcal{S}_{P'}[A \leftarrow T] & \text{if } P \equiv \textbf{let } A \textbf{ hide } T \textbf{ in } P' \\ \mathcal{S}_{P'} & \text{if } P \equiv \textbf{let } A \leq A_1, \ldots, A_n \textbf{ in } P' \\ \emptyset & \text{otherwise} \end{cases}
$$

Then the translation of the types of λ_object relative to a program P is defined in the following way[8]

Definition 5.4.1 For every well-typed program P, we translate a type $T \in_{\mathcal{C}_P,\mathcal{S}_P}$ **Types** into the set of λ&-pretypes generated from the po-set of atomic types $(\mathcal{A}_P, \leq)$ where $\leq$ is the transitive and reflexive closure of $\mathcal{C}_P$. The translation is defined by induction on the structure of T:

[8]As a matter of facts there cannot be in λ_object only user defined atomic types; there must be at least one predefined atomic type `*` together with a constant `?`:`*` to start the definitions (see the implementation of λ_object in appendix A). This does not change the essence of what follows. Just imagine that also $\lambda\&^{+}$contains `*` and `?` and that they are translated by the identity.

$$
\begin{array}{lcl}
[\![A]\!] & = & A \times [\![\mathcal{S}_P(A)]\!] \\
[\![A_1 \times A_2]\!] & = & [\![A_1]\!] \times [\![A_2]\!] \\
[\![S \to T]\!] & = & [\![S]\!] \to [\![T]\!] \\
[\![\{S_i \to T_i\}_{i\in I}]\!] & = & \{[\![S_i]\!] \to [\![T_i]\!]\}_{i\in I}
\end{array}
$$

□

The definition above is well defined. To prove it associate to every $T \in_{\mathcal{C}_P,\mathcal{S}_P}$ **Types** the weight $w(T)$ defined as follows

> $w(A) = n$ if A has been the n-th atomic type defined in the program P.
> $w(S \to T) = w(S \times T) = \max\{w(S), w(T)\}$
> $w(\{S_i \to T_i\}_{i\in I}) = \max_{i\in I}\{w(S_i), w(T_i)\}$

Then it is easy to verify that, thanks to rules for typing and type good formation of λ_object, the translation of a type is always given in terms of the translations of types with a minor weight or with the same weight but a less deep syntax tree (remember that the translation is given w.r.t. a *well-typed* program P, and thus the definitions **let** ... **hide** ... cannot be circular)

The weight above is also used to prove the following proposition

Proposition 5.4.2 $\quad \mathcal{C}_P \vdash S \leq T \quad \Leftrightarrow \quad [\![S]\!] \leq [\![T]\!]$

Proof. Let $d(T)$ denote the depth of the syntax tree of T and associate to every subtyping judgement $S \leq T$ the pair $(w(S) + w(T), d(S) + d(T))$. Then the result follows from a straightforward induction on the lexicographical order of the pairs. The only non trivial case is when $S \leq T$ is $A_1 \leq A_2$:
($\Leftarrow$) If $A_1 \leq A_2$ in $\lambda\&^+$ then this must have been obtained by transitivity and reflexivity from $\mathcal{C}_P$. Thus $\mathcal{C}_P \vdash A_1 \leq A_2$
($\Rightarrow$) Viceversa if $\mathcal{C}_P \vdash A_1 \leq A_2$ then

$$
\begin{array}{rcl}
[\![A_1]\!] \leq [\![A_2]\!] & \Leftrightarrow & A_1 \times [\![\mathcal{S}_P(A_1)]\!] \leq A_2 \times [\![\mathcal{S}_P(A_2)]\!] \\
& \Leftrightarrow & A_1 \leq A_2 \ \wedge\ [\![\mathcal{S}_P(A_1)]\!] \leq [\![\mathcal{S}_P(A_2)]\!]
\end{array}
$$

The first factor follows from $\mathcal{C}_P \vdash A_1 \leq A_2$ and definition 5.4.1. The second follows from the induction hypothesis since the left component of the associated pair strictly decreases. □

This proposition has the following important corollary

Corollary 5.4.3 $U_j = min_{i\in I}\{U_i | U \leq U_i\} \Leftrightarrow [\![U_j]\!] = min_{i\in I}\{[\![U_i]\!] | [\![U]\!] \leq [\![U_i]\!]\}$

We can now define precisely the target calculus of the translation that we call $\mathbf{TARGET}_P$

Definition 5.4.4 The target calculus $\mathbf{TARGET}_P$ of the translation relative to a well-typed program P has as raw terms the set of the $\lambda\&^+$+coerce terms constructed from a denumerable set of variables the constants to encode pairings and a constant c^A of each $A \in \mathcal{A}_P$. Its set of types is formed by $\mathcal{A}_P$ plus the pretypes that are in the image of the translation of definition 5.4.1 plus the types to encode pairings and to type fixpoint combinators. The subtyping relation is the one generated from $(\mathcal{A}_P, \leq)$ on the pretypes. The typing rules are those of $\lambda\&^+$+coerce. □

Thus $\mathbf{TARGET}_P$ is $\lambda\&^+$+coerce but without some types. In particular $A \times [\![T]\!]$ belongs to the types of $\mathbf{TARGET}_P$ if and only if **let** A **hide** T appears in P.

This is precisely stated by the following theorem

Theorem 5.4.5 *The translation of a well-formed type of λ_object satisfy the condition of type formation of $\lambda\&^+$(+coerce)*

Proof. The result follows nearly immediately from definition 5.4.4 and from proposition 5.4.2. Just note that the types added for fixpoint combinators do not interfere with the condition **(c+)** of page 124 □

Note that the statement of the theorem would not hold if we had not restricted the types of $\mathbf{TARGET}_P$. This because $A_1 = A_2 \sqcap A_3$ does not imply $\mathcal{S}_P(A_1) = \mathcal{S}_P(A_2) \sqcap \mathcal{S}_P(A_3)$.

5.4.2 The encoding of the terms

We can now give the translation for the terms

Definition 5.4.6 We give the translation relative to a well-typed program P, of a term of λ_object that does not contains **super**.

$$\begin{array}{lcll}
[\![x^T]\!] & = & x^{[\![T]\!]} & \\
[\![in^A(M)]\!] & = & \mathbf{coerce}^{[\![A]\!]}(\,(c^A\,,\,[\![M]\!])\,) & c^A \text{ is the constant of type } A \\
[\![out^A(M)]\!] & = & \pi_2([\![M]\!]) & \\
[\![\mathbf{coerce}^A(M)]\!] & = & \mathbf{coerce}^{[\![A]\!]}([\![M]\!]) & \\
[\![\lambda x^T.M]\!] & = & \lambda x^{[\![T]\!]}.[\![M]\!] & \\
[\![(M \&^T N)]\!] & = & ([\![M]\!] \&^{[\![T]\!]} [\![N]\!]) & \\
[\![M \circ N]\!] & = & [\![M]\!] \circ [\![N]\!] & \\
[\![<M\,,N>]\!] & = & <[\![M]\!]\,,[\![N]\!]> & \\
[\![\pi_i(M)]\!] & = & \pi'_i([\![M]\!]) & i = 1,2 \\
[\![\mu x^T.M]\!] & = & \Theta_{[\![T]\!]}(\lambda x^{[\![T]\!]}.[\![M]\!]) & \\
[\![\mathbf{let}\ \ldots\ \mathbf{in}\ P']\!] & = & [\![P']\!] &
\end{array}$$

where Θ is defined as in section 3.2 □

Strictly speaking we should have constructed $\mathcal{S}_P$ along the translation in the following way: $[\![\mathbf{let}\ A\ \mathbf{hide}\ T\ \mathbf{in}\ P']\!]_S = [\![P']\!]_{S[A\leftarrow T]}$; however, in the rest of this section, the declarations play a secondary role thus we prefer to deal with them more informally; consequently in the following we omit all the type constraint systems, understanding that they are all relative to the type system of a given program. Note also that we have distinguished two different pairings: one denoted by (,) with projections π_i, the other $<,>$ with projections π'_i. The former is used to codify objects, the latter to encode the pairings of λ_object. We differentiated them so that they cannot interfere one with the other.

Theorem 5.4.7 *If $M{:}T$ then there exists T' such that $[\![M]\!]{:}\,[\![T']\!] \leq [\![T]\!]$*

Proof. The proof consists in a straightforward induction on the structure of the program and uses proposition 5.4.2. Just note that $[\![M]\!]{:}\,[\![T]\!]$ does not hold because of the definition of $[\![out^A(M)]\!]$ □

To conclude the section we have to prove the correctness of our translation, i.e. that if a program of λ_object reduces to a value then its translation reduces to the translation of the value. To prove this theorem we need two technical lemmas. Let us denote by $N\!\downarrow$ the normal form of N.

Lemma 5.4.8 *Let N be a tagged value. If $N\colon D$ then $[\![N]\!]$ has a normal form and $[\![N]\!]\!\downarrow\colon [\![D]\!]$*

Proof. A trivial induction on the structure of tagged terms. Note how the coercion in the translation of in^A blocks the type. □

Lemma 5.4.9 (substitution) $[\![M[x^T\colon= N]]\!] = [\![M]\!][x^{[\![T]\!]}\colon= [\![N]\!]]$

Proof. A straightforward induction on M. Just note that Θ is a closed term. □

Theorem 5.4.10 *If $M \Rightarrow N$ then* $[\![M]\!] \rhd^{+}_{\beta\cup\beta_{\&}\cup(coerce)} [\![N]\!]$

Proof. By induction on the definition of ⇒. It suffices to prove the theorem for the axioms of λ_object. The result then follows by a straightforward use of the induction hypothesis. We have six cases (the axioms for the declarations are trivially solved and we do not consider the axioms for **super**).

1. $\pi_i(<G_1\,, G_2>) \;\Rightarrow\; G_i$ straightforward

2. $out^{A_1}(in^{A_2}(M)) \;\Rightarrow\; M$

$$\begin{array}{lll} [\![out^{A_1}(in^{A_2}(M))]\!] & = & \pi_2([\![in^{A_2}(M)]\!]) \\ & = & \pi_2(\mathbf{coerce}^{[\![A_2]\!]}(\,(c^{A_2}\,,\,[\![M]\!])\,)) \\ & \equiv & (\mathbf{coerce}^{[\![A_2]\!]}(\,(c^{A_2}\,,\,[\![M]\!])\,))\bullet\pi_2 \\ & \rhd_{(coerce)} & \pi_2(\,(c^{A_2}\,,\,[\![M]\!])\,) \\ & \rhd^{+}_{\beta_{\&}\cup\beta} & [\![M]\!] \end{array}$$

3. $out^{A_1}(\mathbf{coerce}^{A_2}(M)) \;\Rightarrow\; out^{A_1}(M)$

$$\begin{array}{lll} [\![out^{A_1}(\mathbf{coerce}^{A_2}(M))]\!] & = & \pi_2([\![\mathbf{coerce}^{A_2}(M)]\!]) \\ & = & \pi_2(\mathbf{coerce}^{[\![A_2]\!]}([\![M]\!])) \\ & \rhd_{(coerce)} & \pi_2([\![M]\!]) \\ & = & [\![out^{A_1}(M)]\!] \end{array}$$

4. $\mu x^T.M \;\Rightarrow\; M[x^T := \mu x^T.M]$

$$\begin{array}{llll} [\![\mu x^T.M]\!] & = & \Theta_{[\![T]\!]}(\lambda x^{[\![T]\!]}.[\![M]\!]) & \\ & \rhd^{*} & (\lambda x^{[\![T]\!]}.[\![M]\!])(\Theta_{[\![T]\!]}(\lambda x^{[\![T]\!]}.[\![M]\!])) & \\ & \rhd_{\beta} & [\![M]\!][x^{[\![T]\!]}\colon= [\![\mu x^T.M]\!]] & \\ & = & [\![M[x^T := \mu x^T.M]]\!] & \text{by lemma 5.4.9} \end{array}$$

5. $(\lambda x^T.M) \cdot N \Rightarrow M[x^T := N]$

$$\begin{array}{rcll} [\![(\lambda x^T.M) \cdot N]\!] & = & (\lambda x^{[\![T]\!]}.[\![M]\!])[\![N]\!] & \\ & \rhd_\beta & [\![M]\!][x^{[\![T]\!]} := [\![N]\!]] & \\ & = & [\![M[x^T := N]]\!] & \text{by lemma 5.4.9} \end{array}$$

6. $(M_1 \&^T M_2) \bullet G^D \Rightarrow M_i \circ G^D$ immediate from corollary 5.4.3 and lemma 5.4.8

$\square$

Chapter 6

Semantics

6.1 Introduction

The role of λ-calculus as core functional language is due to its nature as "pure" theory of functions: just application, MN, and functional abstraction, $\lambda x.M$, define it. In spite of the "minimality" of these notions, full computational expressiveness is reached, in the type-free case. In the typed case, expressiveness is replaced by the safety of type-checking. Yet, the powerful feature of implicit and explicit polymorphism may be added. With polymorphism, one may have type variables, which apparently behave like term variables: they are meant to vary over the intended domain of types, they can be the argument of an application and one may λ-abstract w.r.t. them. These functions depending on type variables, though, have a very limited behavior. A clear understanding of this is provided by a simple remark in [Gir72], where second order λ-calculus was first proposed: no term taking types as inputs can "discriminate" between different types. More precisely, if one extends System F by a term M such that, given different input types U and V, returns 0 when applied to input type V and 1 to U, then normalization is lost. Second order terms, then, are "essentially" constant, or "parametric". Indeed, the notion of parametricity has been the object of a deep investigation, since [Rey84] (see also [ACC93] and [LMS93] for recent investigations).

In chapter 2 with the definition of $\lambda\&$, we have used the functional expressiveness of the typed λ-calculus and extended it by overloading and subtyping, in order to account for some features of object-oriented programming. In that chapter we laid the syntactic basis for a study of a form of dependency of the computation "on the type of the inputs" (in the second part of this thesis we we will focus on a dependency of the computation "on the types in input"). In this chapter, we investigate an *elementary* approach to its mathematical meaning. A more general (categorical) understanding of what we mean by "dependency on the type of the inputs" should be a matter of further investigation, possibly on the grounds of the concrete construction below. Indeed, our model provides an understanding of a slightly modified version of the system in chapter 2, as we focus on "early binding" (see the discussion below) and on the normalizing systems defined in chapter 3.

This chapter, which is a joint work with Giorgio Ghelli and Giuseppe Longo, is organized as follows: in section 6.2 we develop some general syntactic tools, instrumental to our semantic approach which can be applied to any of the systems presented in this part of the thesis.

Section 6.3, introduces the variant with "early binding". Section 6.4 presents the model.

6.2 The completion of overloaded types

This section presents some general, syntactic properties of (overloaded) types, which may be viewed as some sort of "preprocessing" on the syntactic structures and which provide by this an interface towards our semantic constructions.

We have already stressed that subtyping in our system is transitive but is not antisymmetric (it is only a *preorder* relation).

However, since we want to interpret "$\leq$" by an order relation among semantic types, in this section we look for a mechanism to get rid of irrelevant differences between equivalent types.

Definition 6.2.1 Given types U and V, set $U \sim V$ if $U \leq V$ and $V \leq U$. □

Remark If $\{U \to V\} \sim \{U \to V, U' \to V'\}$ then $U' \leq U$ and $V' \sim V$. Indeed, one must have $U \to V \leq U' \to V'$, so that $U' \leq U$ and $V \leq V'$, while $V' \leq V$ follows from $U' \leq U$ by covariance. This gives the intuitive meaning of the equivalence: a type $U' \to V'$ can be freely added or removed from an overloaded type if there is another type $U \to V$ which "subsumes" it, i.e. which is able to produce the same output type on a wider input type ($V \sim V'$ but $U \geq U'$).

We now extend the usual definitions of g.l.b., l.u.b., etc., to a preorder relation. For any partial preorder $\leq$ defined on a set Y and for any $X \subseteq Y$ define

$$\begin{aligned}
\min X &=_{def} \{U \in X | \forall V \in X. U \leq V\} \\
\max X &=_{def} \{U \in X | \forall V \in X. V \leq U\} \\
\inf X &=_{def} \max\{U \in Y | \forall V \in X. U \leq V\} \\
\sup X &=_{def} \min\{U \in Y | \forall V \in X. V \leq U\}
\end{aligned}$$

Note that the four functions above denote a subset of Y, which in the first two cases, if not empty, is an element of $X/\sim$, and in the last two cases an element of $Y/\sim$.

Our next step is the definition of the "completion" of overloaded types; intuitively, the completion of an overloaded type is formed by adding all the "subsumed types" (in the sense of the previous remark), so that two equivalent overloaded types should be transformed, by completion, in *essentially* the same completed type. For this purpose and for the purpose of their semantics, we now adopt a different notation for overloaded types. Write $\Downarrow H$, if the collection H of types has a lower bound.

Definition 6.2.2 [g.o.t.] A *general overloaded type* (g.o.t.) is a pair (K, out) where K is a set of types and out is a function from K to Type such that:

1. if $H \subseteq K$ and $\Downarrow H$ then there exists $V \in K$ such that $V \in \inf H$.
2. *out* is monotone w.r.t. the subtype preorder.

Sometimes we will use $\{U \to out(U)\}_{U \in K}$ to denote the g.o.t. (K, out). □

Notice that V at the point 1 of the definition is not required to be unique, and also that K is not required to be finite. Thus a g.o.t. is *not* a type. But any overloaded type can be seen as a g.o.t. (K, out), with a finite K.

The preorder on g.o.t.'s is the one defined by applying to g.o.t.'s the rules given for $\lambda\&$ in section 2.2.1.

We are now ready to define the notion of completion. We complete a g.o.t. (K, out) by enlarging its domain to its downward closure and by extending the "out" map to the enlarged domain. The extended map $\widehat{out}$ is defined over a type U', essentially, by setting

$$\widehat{out}(U') = out(\min\{U \in K | U' \leq U\}).$$

But recall that min denotes a set of types; thus we have to choose one of them. To this aim, we suppose that a choice function *choose* is defined which chooses a type out of a non-empty set of equivalent types. Then, the extended map can be defined as:

$$\widehat{out}(U') = out(choose(\min\{U \in K | U' \leq U\})).$$

For brevity, we will denote the functional composition of *choose* and min as *a_min*:

$$a_min(X) =_{def} choose(\min(X))$$

Remark Even if $\leq$ is a preorder on **Types**, in the rule [{ }ELIM] there is not ambiguity in the selection of the minimum. Indeed, by the definition of good formation of (overloaded) types we required the property

$$\textbf{(c)} \qquad (U_i {\Downarrow} U_j \Rightarrow \exists! h \in I \qquad U_h \in \inf\{U_i, U_j\})$$

Thus the rule picks up the *unique* U_j with the required property. For the same reason, when the g.o.t. which is the argument of completion (see below) is actually a type, the argument of the *choose* function is just a singleton.

Definition 6.2.3 [completion] Let $\{U \to out(U)\}_{U \in K}$ be a g.o.t.. Its *completion* $\{U \to \widehat{out}(U)\}_{U \in \widehat{K}}$ is the g.o.t. given by: $\widehat{K} = \{U' | \exists U \in K \;\; U' \leq U\}$ and $\widehat{out}(U') = out(a_min\{U \in K | U' \leq U\})$. □

Fact 6.2.4 *The completion of a g.o.t.* $\{U \to out(U)\}_{U \in K}$ *is a well-defined g.o.t..*

Proof. Recall first that Type is a partial lattice: this gives 1 in 6.2.2. As for 2 ($\widehat{out}$ monotonicity), let $U' \leq V'$ be two types such that out is defined on both of them. Both $U'' = a_min\{U \in K | U \geq U'\}$ and $V'' = a_min\{U \in K | U \geq V'\}$ are well-defined by 1; moreover, $V'' \in K$, $V'' \geq V'$ and $V' \geq U'$ imply that $V'' \in \{U \in K | U \geq U'\}$, and then that $U'' \leq V''$, so that $\widehat{out}(U') = out(U'') \leq out(V'') = \widehat{out}(V')$. □

Fact 6.2.5 *In the completion of a g.o.t.* $\{U \to out(U)\}_{U \in K}$*, if* $U \in K$*, then* $\widehat{out}(U) \sim out(U)$*, since* $a_min\{V \in K | U \leq V\} \sim U$ *and out is monotone.*

Clearly, the completion is an idempotent operation (modulo equivalence). Note also that, even for a singleton $K = \{U\}$, $\widehat{K}$ may be infinite (e.g. U equal to the type of ε i.e. $\{\ \}$).

Fact 6.2.6 *By completion, one obtains an equivalent g.o.t., that is:*

$$\{U \to out(U)\}_{U\in K} \sim \{U \to \widehat{out}(U)\}_{U\in\widehat{K}}$$

Proof. "$\geq$": we have to prove that

$$\forall U\in K.\exists U'\in\widehat{K}.U \to out(U) \geq U' \to \widehat{out}(U').$$

Take $U' = U$. By fact 6.2.5, since $U \in K$, $\widehat{out}(U) \sim out(U)$, hence $U \to out(U) \sim U \to \widehat{out}(U)$.

"$\leq$": conversely, we have to prove that

$$\forall U'\in\widehat{K}.\exists U\in K.U \to out(U) \leq U' \to \widehat{out}(U').$$

For $U'\in\widehat{K}$, $\exists U\in K.U' \leq U$; thus, let $Z = a_min\{V\in K|V \geq U'\}$, one has $Z \to out(Z) \leq U' \to \widehat{out}(U')$, since by definition of completion $\widehat{out}(U') = out(Z)$. □

The idea is to interpret overloaded types by using their completions, in the model. However, as some preliminary facts may be stated at the syntactic level, we preferred to define syntactic completions and work out their properties. The theorem 6.2.7 below, is the most important one, since it guarantees the monotonicity of completion. Note that subtyping between overloaded types is contravariant w.r.t. the collections $\widehat{K}$ and $\widehat{H}$. The reader familiar with the semantics of records as indexed products (see [BL90]) may observe analogy with that contravariant understanding of records. This it is not surprising since we showed in section 2.5 that record types may be coded as particular overloaded types.

Theorem 6.2.7 *Let (K, out) and (H, out') be g.o.t.. Then*

$$\{U \to out'(U)\}_{U\in H} \leq \{U \to out(U)\}_{U\in K} \Leftrightarrow \widehat{K} \subseteq \widehat{H} \text{ and } \forall U\in\widehat{K}.\ U \to \widehat{out'}(U) \leq U \to \widehat{out}(U)$$

Proof. ($\Rightarrow$) As for $\widehat{K} \subseteq \widehat{H}$, just observe that, $\forall U\in\widehat{K}$, $\exists U'\in K, U \leq U'$; hence $\exists V\in H.U' \leq V$, by the assumption, and, thus, $U \in \widehat{H}$. Let now $U \in \widehat{K}$. There exists then $V\in K$ such that $U \leq V$: take $V = a_min\{W\in K|U \leq W\}$. As $V\in K$, by the assumption one has:

$$\exists U'\in H.V \leq U' \text{ and } out'(U') \leq out(V) \tag{6.1}$$

Now,

$$\widehat{out}(U) = out(a_min\{W\in K|U \leq W\}) = out(V) \tag{6.2}$$

Thus:

$$\begin{array}{rcll} \widehat{out'}(U) & \leq & \widehat{out'}(U') & \text{since } U \leq V \leq U' \text{ and } U, U'\in\widehat{H} \\ & \sim & out'(U') & \text{by fact 6.2.5, since } U' \in H \\ & \leq & out(V) & \text{by (6.1)} \\ & = & \widehat{out}(U) & \text{by (6.2)} \end{array}$$

In conclusion, $\forall U\in\widehat{K}.U \to \widehat{out'}(U) \leq U \to \widehat{out}(U)$.

($\Leftarrow$) We have to prove that $\forall U\in K.\exists V\in H.V \geq U$ and $out'(V) \leq out(U)$. Let $U\in K$. By hypothesis, $U\in\widehat{H}$. Hence, $V = a_min\{Z\in H|U \leq Z\}$ is well-defined. Thus:

$$\begin{array}{rcll} out'(V) & = & \widehat{out'}(U) & \text{by definition} \\ & \leq & \widehat{out}(U) & \text{hypothesis} \\ & \sim & out(U) & \text{by fact 6.2.5, since } U\in K \end{array}$$

□

Corollary 6.2.8 *Let (K, out) and (H, out') be g.o.t.; then:*

$$\{U \to out'(U)\}_{U\in H} \sim \{U \to out(U)\}_{U\in K} \Leftrightarrow \widehat{K} = \widehat{H} \text{ and } \forall U \in \widehat{K}. \widehat{out'}(U) \sim \widehat{out}(U)$$

In conclusion, completions are not exactly canonical representatives of equivalence classes, but at least they push the differences between two overloaded types one level inside the types. In this way in the interpretation of types we will be able to get rid of the differences between equivalent types by iterating completion at all the levels inside the type structure. The fact that a type is equivalent to its completion makes it clear that an overloaded type, seen modulo $\sim$, does not describe the structure of the corresponding functions (e.g. how many different branches they have) but just which are the contexts where they can be inserted. Hence we understand overloaded types as "type-checkers", used to check the "dimension" of programs, similarly as in Physics where, by a "dimensional analysis", one checks that in an equation, say, a force faces a force etc.. This will be the crucial semantic difference between arrow types and overloaded types, since arrow types will keep their usual, more restrictive, meaning as "collection of functions or morphisms identified by the input and output types" (see section 6.4.2 for further discussions).

6.3 Early Binding

Object-oriented languages are characterized by an interplay of many features. We have selected three of them—overloading, late binding and subtyping—that, in our opinion, suffice to model the relevant features of a class-based object-oriented language. Not that these features are exclusive to this approach: overloading existed long before object-oriented languages (FORTRAN already used it) while subtyping, even if it was first suggest by object-oriented paradigms, has been included in other different paradigms (e.g. EQLOG [GM85], LIFE [AKP91] or Quest [CL91a]). But their combination is peculiar to object-oriented programming. And exactly the interplay of all these features makes the object-oriented approach so useful in the large-scale software production.

At semantic level, our system presents four main technical challenges. The first is the true dependence of overloaded functions from types. The second is the fact that subtyping is not an order relation. The third is that subtyping even if it respects the structure of types it does not respect their "size" (we will say more about it below). The fourth is the distinction between run-time types and compile-time types. In this chapter we just concentrate on the first two aspects, which already requires some technical efforts, while we will avoid the third problem by considering only the normalizing systems of chapter 3 and we will avoid the fourth one by taking into consideration only a subsystem where the type of the arguments of overloaded functions is "frozen", i.e. is the same at compile time and at run time. This is just a first step in the direction of defining a semantics for the full system.

The resulting system is someway intermediate between late binding and early binding overloading. It features early binding, since for any application of an overloaded function the type which will be used to perform branch selection is already known at compile time, as happens for example with arithmetic operators in imperative languages. It has still a form of late binding since, as overloaded functions are first class values which can be the result of expression evaluation, it is not possible to get rid of branch selection at run time. For

example if in the body of a function which has a formal parameter x of type $\{U_i \to T_i\}$, this parameter is applied to an argument of type U, then we know that the branch selection will be based on U; though this branch selection cannot be statically performed since the function associated with x is unknown at compile time.

If overloaded functions were not first class (i.e. if no variable were allowed to possess an overloaded type) this restricted calculus would correspond to the "classical" (i.e. with early binding) implementation of overloading in imperative languages: the standard example is the operator `+` which is defined both on *reals* and *integers*, though a different code is used according to the type of the argument[1]. What happens in these languages is that a *preprocessor* scans *at compile time* the text of a program looking for all occurrences of `+` and it substitutes them by a call to the appropriate code, depending on whether they are applied to reals or integers[2]. As far as we know, all languages that use in an explicit way overloading (and not implicitly as it is done in object-oriented programming via the method definitions) base the selection of the code on the type possessed by the arguments at compile time.

We obtain this "half-way-early-binding" restriction of our system simply by adding explicit coercions and imposing that every argument of an overloaded function is coerced. We recall that a coercion $\mathbf{coerce}^V$ is just a function which, informally, does nothing, but which cannot be reduced, so that the type of all the residuals of a term $\mathbf{coerce}^V(M)$ is always V (see section 4.2). Thus to model overloading with early binding we require that, for each overloaded application, a coercion freezes the type of the argument up to branch selection. We change the system in the following way: Pretypes, Types and rules are as before. Terms are now:

$$M ::= x^V \mid \lambda x^V.M \mid M \cdot M \mid \mathbf{coerce}^V(M) \mid \varepsilon \mid M \& M \mid M \bullet \mathbf{coerce}^V(M)$$

We add to the rules of type-checking the one for coercions:

[COERCION] $$\frac{M : U \leq V}{\mathbf{coerce}^V(M) : V}$$

We define the reduction on the coercion as in section 4.2

(*coerce*) $$\mathbf{coerce}^V(M) \circ N \ \triangleright M \circ N$$

where $\circ$ denotes either $\cdot$ or $\bullet$. This is the minimal extension of the system. This rule is needed since otherwise coercions could prevent some β or $\beta_{\&}$ reductions. This rule does not interfere with our use of coercions, since it only allows us to reduce the left hand side, but not the right hand side, of an application.

[1] This example is sometimes misleading because of the fact that the codes for the two branches must give the same results when applied to integers (integer numbers being a subset of real numbers); this extra property is proper to *coherent overloading*, but in general the branches do not need to be related on the values they return

[2] The same is true for paradigms which have a cleverer use of overloading: for example, when the programmer can define his or her own overloaded operators. This is possible in Haskell; in this language the implementation of overloading is based on strong theoretical grounds as shown in [WB89]; in that paper it is also shown how the selection in overloading can be solved at compile time by the use of a preprocessor

Finally we consider only stratified systems, i.e. those system in which the typing or subtyping relations are defined so to satisfy the hypothesis of theorem 3.3.1. The necessity of restricting our attention to the strongly normalizing systems defined in section 3.3 comes from the fact that the subtyping relation for the full λ&-calculus does not respect the size of the types, since a type maybe a subtype of a strict occurrence of itself: for example, $\{\{\} \to T\} \leq \{\}$ and $\{\}$ is a strict occurrence of $\{\{\} \to T\}$. The consequence of this fact is that it is impossible to give a definition of the semantics by induction on the structure of types, since to give the semantics to an overloaded type we need to know the semantics of the subtypes of its input types and thus, in the case above, of the type itself (in particular definition 6.4.12 would not be well-defined for the full λ&). We are in presence of a new form of impredicativity. This does not happen in the strongly normalizing systems, where it is possible to define a well-founded order that respects both the subtyping relation and the size of the types.

We are now able to give the denotational semantics of any of these systems that we generally denote by λ&-*early*.

6.4 Semantics

6.4.1 PER as a model

In this section we give the basic structural ideas which will allow us to interpret the syntax of λ&-*early*. Namely, we state which geometric or algebraic structures may interpret arrow and overloaded types; terms will be their elements and will be interpreted in full details in section 6.4.3.

A general model theory of λ&-*early* may be worth pursuing as an interesting development on the grounds of the concrete model below. Indeed, by some general categorical tools, one may even avoid to start with a model of type-free lambda calculus, but this may require some technicalities from Category Theory (see [AL91]). Thus we use here a model $(\mathcal{D}, \cdot)$ of type-free lambda calculus and a fundamental type structure out of it. We survey first the basic ideas for the construction. Later we specialize the general construction by starting out with a specific type-free model which will yield a semantics for our typed calculus.

PER out of $(\mathcal{D}, \cdot)$

Let $(\mathcal{D}, \cdot)$ be an applicative structure, which yields a model of type-free lambda calculus (see [Bar84]).

Example 6.4.1 Let $\mathcal{P}\omega$ be the powerset of the natural numbers, ω. $\mathcal{P}\omega$ may be turned into an applicative structure $(\mathcal{P}\omega, \cdot)$, indeed a model of type-free λ-calculus, by setting:

$$a \cdot b = \{k | \exists e_h \subseteq b, \prec h, k \succ \in a\}$$

where a and b are elements of $\mathcal{P}\omega$, $\{e_i\}_{i\in\omega}$ is an enumeration of finite sets of numbers and $\prec _, _ \succ$ is a bijective coding of $\omega \times \omega$ into ω (see [Sco76] and also [Lon83] for a general set-theoretic construction). □

We first define the category of types as Partial Equivalence Relations out of $(\mathcal{D}, \cdot)$. When A is a symmetric and transitive relation on $\mathcal{D}$, we set, for $n, m \in \mathcal{D}$:

$$\begin{array}{rcl} \mathbf{nAm} & \stackrel{def}{\Leftrightarrow} & n \text{ is related to } m \text{ by } A \\ \mathbf{dom}(\mathbf{A}) & \stackrel{def}{=} & \{n | nAn\} \\ \lceil \mathbf{n} \rceil_{\mathbf{A}} & \stackrel{def}{=} & \{m | mAn\} \text{ (the equivalence class of } n \text{ with respect to } A) \\ \mathbf{Q}(\mathbf{A}) & \stackrel{def}{=} & \{\lceil n \rceil_A | n \in dom(A)\} \text{ (the quotient set of } A). \end{array}$$

Clearly, if A is a symmetric and transitive relation on $\mathcal{D}$ then A is an equivalence relation on $dom(A)$, as a subset of $\mathcal{D}$. (Note that, even if we will use n, m for arbitrary elements of $\mathcal{D}$, when $\mathcal{D}$ is $\mathcal{P}\omega$ each element n in $\mathcal{D}$ is actually a *set* of numbers).

Definition 6.4.2 The category **PER** (of Partial Equivalence Relations) is defined as:
- *objects*: $A \in \mathbf{PER}$ iff A is a symmetric and transitive relation on $\mathcal{D}$
- *morphisms*: $f \in \mathbf{PER}[A, B]$ iff $f : Q(A) \to Q(B)$ and $\exists n \in \mathcal{D}. \forall a \in dom(A). f(\lceil a \rceil_A) = \lceil n \cdot a \rceil_B$ □

Note that the morphisms in **PER** are computable, w.r.t. $(\mathcal{D}, \cdot)$, in the sense that any $n \in \mathcal{D}$ such that $\forall a \in dom(A). f(\lceil a \rceil_A) = \lceil n \cdot a \rceil_B$ computes or **realizes** $f : Q(A) \to Q(B)$ in the definition (notation: $n \Vdash_{A \to B} f$). Thus **PER** is a category where the identity map, in each type, is computed by (at least) the interpretation of the term $\lambda x.x$, i.e. the identity function on $\mathcal{D}$.

Theorem 6.4.3 **PER** *is a CCC.*

Proof.(Hint, the proof is in several papers since [Sco76]; in particular, in [AL91]). The exponent object $A \to B$ is defined by

$$\forall m, n.\ m(A \to B)n \quad \Leftrightarrow \quad \forall p, q(pAq \Rightarrow (m \cdot p)B(n \cdot q))$$

Products are defined by taking a coding of pairs of $\mathcal{D}$ into $\mathcal{D}$, as given for example by the fact that $\mathcal{D}$ is a model of type free lambda-calculus. □

To clarify the construction, let us look in more detail at the exponent objects in **PER**. Take say $A \to B$, that is, the representative of $\mathbf{PER}[A, B]$. Then by definition each map $f \in \mathbf{PER}[A, B]$ is uniquely associated with the equivalence class of its realizers, $\lceil n \rceil_{A \to B} \in A \to B$ in the sense above. It should be clear that the notion of realizer, or "type-free computation" computing the typed function, is made possible by the underlying type-free universe, $(\mathcal{D}, \cdot)$. As we will discuss later, this gives mathematical meaning to the intended type-free computations of a typed program after compilation. In this context, it is common to identify, by an abuse of language, each typed function with the equivalence class of its realizers. Of course, the semantic "$\to$" gives meaning to arrow types.

Definition 6.4.4 The semantics of arrow types is given by $[\![U \to V]\!] = [\![U]\!] \to [\![V]\!]$ □

Subtyping

Before going into the semantics of the other types, we briefly introduce the meaning of "subtypes", in view of the relevance this notion has in our language. The semantics of subtypes over **PER** is given in terms of "subrelations", (see [BL90]).

Definition 6.4.5 [subtypes] Let $A, B \in \mathbf{PER}$. Define: $A \leq B$ iff $\forall n, m.(nAm \Rightarrow nBm)$ □

The intuition for this approach to subtyping is better understood when looking at "arrow types".

Proposition 6.4.6 *Let $A, A', B, B' \in \mathbf{PER}$ be such that $A' \leq A$ and $B \leq B'$. Then $A \to B \leq A' \to B'$. In particular, for $n \in dom(A \to B)$, one has $\lceil n \rceil_{A\to B} \subseteq \lceil n \rceil_{A'\to B'}$*

Proof.

$$\begin{array}{rcll} n(A \to B)m & \Leftrightarrow & \forall p, q.(pAq \Rightarrow n \cdot pBm \cdot q) & \\ & \Rightarrow & \forall p, q.(pA'q \Rightarrow n \cdot pB'm \cdot q) & \text{as } pA'q \Rightarrow pAq \Rightarrow n \cdot pBm \cdot q \Rightarrow n \cdot pB'm \cdot q \\ & \Leftrightarrow & n(A' \to B')m & \end{array}$$

The rest is obvious. □

The proposition gives the antimonotonicity of $\to$ in its first argument, as formalized in the rules and required by subtyping. Moreover, and more related to the specific nature of this interpretation of $\to$, this gives a nice interplay between the extensional meaning of programs and the intensional nature of the underlying structure, namely between functions and the set of indexes that compute them. Indeed, typed programs are interpreted as extensional functions in their types, as we identify each morphism in **PER** with the equivalence class of its realizers. That is, in the notation of the proposition, let $\lceil n \rceil_{A\to B} \in A \to B$ represent $f \in \mathbf{PER}[A, B]$ in the exponent object $A \to B$. Note then that the intended meaning of subtyping is that one should be able to run any program in $A \to B$ on terms of type A' also, as A' is included in A. When $n \Vdash_{A\to B} f$, this is exactly what $\lceil n \rceil_{A\to B} \subseteq \lceil n \rceil_{A'\to B'}$ expresses: any computation which realizes f in the underlying type-free universe actually computes f viewed in $A' \to B'$ also. Of course, there may be more programs for f in $A' \to B'$, in particular if A' is strictly smaller than A. This elegant interplay between the extensional collapse, which is the key step in the hereditary construction of the types as partial equivalence relations, and the intensional nature of computations is a fundamental feature of these realizability models. Clearly "$\leq$" is a partial order which turns the objects of **PER** into an algebraic complete lattice. The crucial point here is that "$\leq$" defines a refinement relation which goes exactly in the sense we want in order to interpret subtypes. Namely if $A \leq B$ then the equivalence class of A are included in those of B or A is finer than B.

Note finally that, given $n \in \mathbf{dom}(A)$ and $A \leq B$, we may view the passage from $\lceil n \rceil_A$ to $\lceil n \rceil_B$ as an obvious coercion.

Definition 6.4.7 [semantic coercions] Let $A, B \in \mathbf{PER}$ with $A \leq B$. Define $c_{AB} \in \mathbf{PER}[A, B]$ by $\forall n \in \mathbf{dom}(A)$ $c_{AB}(\lceil n \rceil_A) = \lceil n \rceil_B$ □

Remark By the previous definition, for any $a \in \mathrm{Q}(A)$, $c_{AB}(a) \supseteq a$

Syntactic coercions are denoted by $\mathbf{coerce}^V$ where V is the type the argument is coerced to; the type-checker guarantees that this type is greater than the type of the argument of $\mathbf{coerce}^V$. Also in the semantics we need to know the type of the argument since the semantic coercions are "typed functions", from a p.e.r. to another: thus, we have denoted semantic coercions between p.e.r.'s A and B by c_{AB}; the double indexation and the different font distinguish a semantic coercion from the syntactic symbol. Also, for the sake of conciseness if U and V are syntactic types, we denote by c_{UV} the semantic coercion $c_{[\![U]\!][\![V]\!]}$

Note that semantic coercions do not do any work as type-free computations but, indeed, change the "type" of the argument, i.e. its equivalence class and the equivalence relation where it lives . Thus they are realized by the indexes of the type free identity map, among others, and they are meaningful maps in the typed structure.

Since terms will be interpreted as equivalence classes in (the meaning as p.e.r.'s of) their types, we need to explain what the application of an equivalence class to another equivalence class may mean, as, so far, we only understand the application "$\cdot$" between elements of the underlying type-free structure $(\mathcal{D}, \cdot)$.

Definition 6.4.8 [Application] Let A, A' and B be p.e.r.'s, with $A' \leq A$. Define then, for $n(A \to B)n$ and $mA'm$, $\lceil n \rceil_{A\to B} \cdot \lceil m \rceil_{A'} = \lceil n \cdot m \rceil_B$. □

Note that this is well defined, since $mA'm'$ implies mAm' and, thus, $n \cdot mBn' \cdot m'$, when $n(A \to B)n'$. This is clearly crucial for the interpretation of our "arrow elimination rule". We end this section on subtyping by two technical lemmas that will be heavily used in the next sections.

Lemma 6.4.9 (Monotonicity of application) *Let a, b, a', b' be equivalence classes such that the applications $a \cdot b$ and $a' \cdot b'$ are well defined (i.e. $a \in Q(A_1 \to A_2)$ and $b \in Q(B)$ with $B \leq A_1$, and similarly for a' and b'). If $a \subseteq a'$ and $b \subseteq b'$ then $a \cdot b \subseteq a' \cdot b'$*

Proof. $n \in a \cdot b \quad \Leftrightarrow \quad \exists p \in a, q \in b.\ n = p \cdot q \quad \Rightarrow \quad p \in a', q \in b' \quad \Rightarrow \quad n = p \cdot q \in a' \cdot b'$ □

Lemma 6.4.10 (Irrelevance of coercions) *Let A, A' and B be p.e.r.'s, with $A' \leq A$. Assume that $n(A \to B)n$ and $mA'm$. Then*
$\lceil n \rceil_{A\to B} \cdot c_{A'A}(\lceil m \rceil_{A'}) = \lceil n \rceil_{A\to B} \cdot \lceil m \rceil_A = \lceil n \cdot m \rceil_B = \lceil n \rceil_{A\to B} \cdot \lceil m \rceil_{A'} = c_{A\to B A'\to B}(\lceil n \rceil_{A\to B}) \cdot \lceil m \rceil_{A'}$.

Proof. Immediate □

6.4.2 Overloaded types as Products

The intuitive semantics of overloaded types is quite different from the meaning of arrow types. The essential difference is that types directly affect the computation: the output value of a $\beta_{\&}$ reduction explicitly depends on the type of the $(M_1 \& M_2)$ term, and on the type of the argument N.[3]

[3]The fact that terms depend on types should not be confused with the different situation of "dependent types" where types depend on terms, e.g. in the Calculus of Construction [CH88]

A second difference has to do with the ability to accept as parameters values of any type which is a subtype of the input types explicitly specified. This fact is managed implicitly in arrow types. For example, let $M{:}U \to V$, then M will be interpreted as a function from the meaning of U to the meaning of V, as sets (or objects in a category of sets) and $U \to V$ will be interpreted as the collection of such functions. Robust structural properties of the model we propose will allow a function in $U \to V$ to be applied to elements of a subtype of U, as if they were in U. This kind of interpretation is not possible with overloaded types, at least since the set of acceptable input types has not, generally, a maximum.

Thus two crucial properties need to be described explicitly in the semantics of overloaded terms. First, output values depend on types; second, as a type may have infinitely many subtypes and the choice of the branch depends on "$\leq$", overloaded semantic functions explicitly depend on infinitely many types. By this, we will consider overloaded functions as, essentially, functions which take two parameters, one type and one value, and give a result whose type depend on the first parameter. Hence overloaded functions of, say, a type $\{U \to V, U' \to V\}$ will be elements of the indexed product (see later)

$$\prod_{W \leq U,U'} (W \to V).$$

In other words, the interpretation of $U \to V$ will be given by the usual set of functions from the interpretation of U to the interpretation of V (in a suitable categorical environment), while the meaning of $\{U \to V, U' \to V\}$ will directly take care of the possibility of applying overloaded functions to all the subtypes of the argument types.

The fact that the index in the product ranges over all subtypes of U, U', not just over $\{U, U'\}$, solves a third problem of overloaded types: the fact that subtyping is just a preorder, while its semantics interpretation is an order relation. By exploiting the notion of completion defined in section 6.2, we will be able to interpret all equivalent types as the same object.

We are now ready to be formal.

In set theory, given a set A and a function $G : A \to \mathbf{Set}$ ($\mathbf{Set}$ is the category of sets and set-theoretical maps), one defines the indexed product:

$$\bigotimes_{a \in A} G(a) = \{f \mid f : A \to \cup_{a \in A} G(a) \text{ and } \forall a \in A.\ f(a) \in G(a)\}$$

If A happens to be a subset of an applicative structure $(\mathcal{D}, \cdot)$ and $G : A \to \mathbf{PER}$, then the resulting product may be viewed as a p.e.r. on $\mathcal{D}$, as follows.

Definition 6.4.11 Let $A \subseteq \mathcal{D}$ and $G : A \to \mathbf{PER}$. Define the p.e.r. $\prod_{a \in A} G(a)$ by

$$n(\prod_{a \in A} G(a))m \quad \Leftrightarrow \quad \forall a \in A.\ n \cdot a G(a) m \cdot a$$

□

Remark [Empty product] Notice that, by the definition above, for any G:

$$\prod_{a \in \emptyset} G(a) = \mathcal{D} \times \mathcal{D}$$

Clearly, $\prod_{a\in A} G(a)$ is a well defined p.e.r. and may be viewed as a collection of computable functions, relative to $\mathcal{D}$: any element in $\mathbf{dom}(\prod_{a\in A} G(a))$ computes a function in $\bigotimes_{a\in A} G(a)$, and when $n \prod_{a\in A} G(a) m$, then n and m compute the same function. That is, by the usual abuse of language, we may identify functions and equivalence classes and write:

$$f \in \prod_{a\in A} G(a) \;\text{ iff }\; f \in \bigotimes_{a\in A} G(a) \text{ and } \exists n \in \mathcal{D}.\forall a \in A. f(a) = \lceil n \cdot a \rceil_{G(a)}. \tag{6.3}$$

We then say that n **realizes** f.

Our aim is to give meaning to functions "computing with types". The idea is to consider the type symbols as a particular subset of $\mathcal{D}$ and use some strong topological properties of a particular model $(\mathcal{D}, \cdot)$, namely of $(\mathcal{P}\omega, \cdot)$ in 6.4.1, to interpret these peculiar functions. Thus, from now on, we specialize $(\mathcal{D}, \cdot)$ to $(\mathcal{P}\omega, \cdot)$. Recall that $\mathcal{P}\omega$ may be given a topological structure, the Scott topology, by taking as a basis the empty set plus the sets $\{a \in \mathcal{P}\omega \mid e_n \subseteq a\}$, where $\{e_n\}_{n\in\omega}$ is an enumeration of the finite subsets of $\mathcal{P}\omega$.

Assume then that each type symbol U is associated, in an injective fashion, with an element n in $\mathcal{D}$, the **semantic code** of U in $\mathcal{D}$. Call [**Type**]$\subseteq \mathcal{D}$ the collection of semantic codes of types. The choice of the set of codes is irrelevant, provided that

- it is in a bijection with **Type**;
- the induced topology on [**Type**] is the discrete topology.

These assumptions may be easily satisfied, in view of the cardinality and the topological structure of the model $\mathcal{D}$ we chose. For example, enumerate the set of type symbols and fix [**Type**] to be the collection of singletons $\{\{i\}|i \in \omega\}$ of $\mathcal{P}\omega$ (**Type** is countable as each type has a finite representation). We then write $\mathcal{T}_n$ for the type-symbol associated with code n[4] and, given $K \subseteq$ **Type**, we set $[K] = \{n|\mathcal{T}_n \in K\}$.

We can now interpret as a p.e.r. any product indexed over a subset $[K]$ of [**Type**]. Indeed, this will be the semantic tool required to understand the formalization of overloading we proposed: in $\lambda\&$, the value of terms or procedures may depend on types. This is the actual meaning of overloaded terms: they apply a procedure, out of a finite set of possible ones, according to the type of the argument. As terms will be functions in the intended types (or equivalence classes of their realizers), our choice functions will go from codes of types to (equivalence classes in) the semantic types.

Remark The reader may observe that there is an implicit higher order construction in this: terms may depend on types. However:

- in view of the countable (indeed finite) branching of overloaded terms and types, we do not need higher order models to interpret this dependency;
- note though that the intended meaning of an overloaded term is a function which depends on a possibly infinite set of input types, as it accepts terms in any subtype of the U_i types in the $\{U_i \to V_i\}$ types. Whence the use of g.o.t.'s and completions.

[4] Remember that, despite the letter n, n is a singleton, not just an integer.

- known higher order systems (System F, Calculus of Constructions...) would not express our "true" type dependency, where different types of the argument may lead to essentially different computations. This was mentioned in the introduction and it is understood in the **PER** model of these calculi by a deep fact: the product indexed over (uncountable) collections of types is isomorphic to an intersection (see [LM91]). A recent syntactic understanding of this phenomenon may be found in [LMS93].

Remark Overloaded functions are similar, in a sense, to records; in the first case the basic operation is selection of a function depending on a type, while in the second case it is selection of a field depending on a label. Consequently, subtyping is strictly related too: theorem 6.2.7 shows that, working with the completion of types, subtyping is the same in the two cases. However we cannot get rid of overloaded types in λ&-early by encoding them as product types, using the technique developed in [CL91a] for record types, since the completion of an overloaded type is an infinite structure, and also since we want to lay foundations which can be used to study the whole late binding version of λ&.

Now we are ready to define the semantics of overloaded types as products. In view of the fact that we want to interpret subtyping, which is a preorder, by an order relation in the model, we will use the completion to get rid of "irrelevant differences" between overloaded types.

Definition 6.4.12 The semantics of overloaded types is given by

$$[\![\{U \to out(U)\}_{U\in K}]\!] = \prod_{n\in[\widehat{K}]} [\![\mathcal{T}_n \to \widehat{out}(\mathcal{T}_n)]\!]$$

where $[\widehat{K}] = \left\{n \,\middle|\, \mathcal{T}_n \in \widehat{K}\right\}$ □

This definition is well founded since we consider only the stratified system (and thus we can use the function rank to define a weight for each syntactic type and prove that the semantics of an overloaded type is given in terms of the semantic of types of smaller weight), and it has a well defined meaning over **PER**, by definition 6.4.11, where $A = [\widehat{K}]$ and $G\colon [\widehat{K}] \to \mathbf{PER}$ is given by $G(n) = [\![\mathcal{T}_n \to \widehat{out}(\mathcal{T}_n)]\!]$. It clearly extends to g.o.t.'s, as we only need that K is countable, here. Now we are finally in the position to check that the preorder on types is interpreted as the partial order "$\leq$" on **PER**.

Theorem 6.4.13 *If $U \leq V$ is derivable, then $[\![U]\!] \leq [\![V]\!]$ in* **PER**.

Proof. The proof goes by induction on the structure of types, the only critical case concerns the overloaded types and will be an easy consequence of theorem 6.2.7.

atomic types

by definition

arrow types

by proposition 6.4.6

overloaded types

Let (K, out) and (H, out') be g.o.t.. Assume that $\{U \to out'(U)\}_{U \in H} \leq \{U \to out(U)\}_{U \in K}$. We need to show that $\prod_{i\in[\widehat{H}]}[\![\mathcal{T}_i \to \widehat{out'}(\mathcal{T}_i)]\!] \leq \prod_{i\in[\widehat{K}]}[\![\mathcal{T}_i \to \widehat{out}(\mathcal{T}_i)]\!]$ in **PER**.

By 6.2.7, $[\widehat{K}] \subseteq [\widehat{H}]$ and $\forall i \in [\widehat{K}].\mathcal{T}_i \to \widehat{out'}(\mathcal{T}_i) \leq \mathcal{T}_i \to \widehat{out}(\mathcal{T}_i)$. Hence:

$$\begin{array}{rcll} m\prod_{i\in[\widehat{H}]}[\![\mathcal{T}_i \to \widehat{out'}(\mathcal{T}_i)]\!]n & \Leftrightarrow & \forall i\in[\widehat{H}].m[\![\mathcal{T}_i \to \widehat{out'}(\mathcal{T}_i)]\!]n & \text{by definition} \\ & \Rightarrow & \forall i\in[\widehat{K}].m[\![\mathcal{T}_i \to \widehat{out}(\mathcal{T}_i)]\!]n & \text{by } 6.2.7 \\ & \Rightarrow & m\prod_{i\in[\widehat{K}]}[\![\mathcal{T}_i \to \widehat{out}(\mathcal{T}_i)]\!]n & \end{array}$$

□

6.4.3 The semantics of terms

We can now give meaning to terms of the $\lambda\&$-*early*, with the use of coercions introduced in the previous section. In the following we index a term by a type as a shorthand to indicate that the term possesses that type.

Recall that give a per A we denote by $Q(A)$ the set of its equivalence classes. An environment e for typed variables is a map $e\colon Var \to \bigcup_{A\in\mathbf{PER}} Q(A)$ such that $e(x^U) \in Q([\![U]\!])$. Thus each typed variable is interpreted as an equivalence class in its semantic type. This will be now extended to the interpretation of terms by an inductive definition, as usual. Since we make a large use of the quotient sets we prefer to introduce the following notation

Notation 6.4.14 *Let A be a per on $\mathcal{D}$ and $a \subseteq \mathcal{D}$. We write $a \Subset A$ if $a \in Q(A)$.*

Thus for an environment e we have that $e(x^U) \Subset [\![U]\!]$

In spite of the heavy notation, required by the blend of subtyping and overloading, the intuition in the next definition should be clear. The crucial point 6 gives meaning to an overloaded term by a function which lives in an indexed product (as it will be shown formally below): the product is indexed over (indexes for) types and the output of the function is the (meaning of the) term or computation that one has to apply. Of course, this is presented inductively. Some coercions are required as M_1 and M_2 may live in smaller types than the ones in $\&^{\{V_i \to W_i\}_{i\leq n}}$. Then, in point 7, this term is actually applied to the term argument of the overloaded term.

Definition 6.4.15 [semantics of terms] Let $e\colon Var \longrightarrow \bigcup_{A\in PER} Q(A)$ be an environment. Set then:

1. $[\![\varepsilon]\!]_e = \mathcal{D}$, the only equivalence class in the p.e.r. $\mathcal{D} \times \mathcal{D}$ (see remark 6.4.2)

2. $[\![x^U]\!]_e = e(x^U)$

3. $[\![\lambda x^U.M^V]\!]_e = \lceil n \rceil_{[\![U\to V]\!]}$ where n is a realizer of f such that $\forall u \Subset [\![U]\!]\ .\ f(u) = [\![M^V]\!]_{e[x:=u]}$

4. $[\![M^{U\to V}N^W]\!]_e = [\![M^{U\to V}]\!]_e[\![N^W]\!]_e$

5. $[\![\mathbf{coerce}^V(M^U)]\!]_e = c_{UV}([\![M^U]\!]_e)$ (the semantic coercion)

6. Let $(M_1\&^{\{V_i \to W_i\}_{i\leq n}} M_2): \{U \to out(U)\}_{U \in \{V_i\}_{i\leq n}}$ with $M_1: T_1 \leq \{V_i \to W_i\}_{i<n}$ and $M_2: T_2 \leq V_n \to W_n$.

   Set then $[\![M_1\&M_2]\!]_e = f$ such that, given $j \in [\widehat{\{V_i\}_{i\leq n}}]$ and $Z = \min{}^5\{U \in \{V_i\}_{i\leq n} | \mathcal{T}_j \leq U\}$, one has

$$f(j) = \begin{cases} c_{T_2(\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j))}([\![M_2]\!]_e) & \text{if } Z = V_n \\ (c_{T_1(\{V_i \to W_i\}_{i<n})}[\![M_1]\!]_e)(j) & \text{else} \end{cases}$$

7. $[\![M^{\{U \to out(U)\}_{U\in K}} \bullet \mathbf{coerce}^V(N^W)]\!]_e = [\![M^{\{U \to out(U)\}_{U\in K}}]\!]_e(i)[\![\mathbf{coerce}^V(N^W)]\!]_e$ where $\mathcal{T}_i = V$.

□

Remark Notice that this semantics does not interpret reduction as equality. Indeed:

$$\begin{array}{rcll} [\![(\lambda x^V.Q^U)P^W]\!]_e & = & [\![\lambda x^V.Q^U]\!]_e[\![P^W]\!]_e & \\ & = & \lceil n \rceil_{[\![V\to U]\!]}[\![P^W]\!]_e & \text{with } n \text{ as in point 3 of definition 6.4.15} \\ & = & \lceil n \rceil_{[\![V\to U]\!]}(c_{WV}[\![P^W]\!]_e) & \text{by 6.4.10} \\ & = & [\![Q^U]\!]_{e[x:=c_{WV}[\![P^W]\!]_e]} & \text{by point 3 of definition 6.4.15} \end{array}$$

In general, $[\![Q]\!]_{e[x:=c_{WV}[\![P^W]\!]_e]}$ is different from $[\![Q[x:=P^W]]\!]_e$. For example, if $Q = x$, the two expressions evaluate to $c_{WV}[\![P^W]\!]_e$ and to $[\![P^W]\!]_e$ respectively. This will be more generally understood in 6.4.18.

The soundness of this definition is split into two theorems and is proved right below. We recall first, in a lemma, that $\mathcal{P}\omega$ is an "injective" topological space.

Lemma 6.4.16 (injectivity) *Let Y be a topological space and $X \subseteq Y$, a subspace with the induced topology. Then any continuous $h : X \to \mathcal{P}\omega$ can be extended to a continuous $\hbar : Y \to \mathcal{P}\omega$. Indeed, $\hbar$ is given by $\hbar(y) = \sqcup\{\sqcap\{h(x)|x \in X \cap U\}| y \in U\}$. (The proof is easy; see [Sco76] for this and more properties of $\mathcal{P}\omega$).*

Theorem 6.4.17 (soundness w.r.t. type-checking) *If $N: U$ then, for any environment e, $[\![N]\!]_e$ is well defined and $[\![N]\!]_e \Subset [\![U]\!]$.*

Proof. The proof goes by induction on the structure of N:

1. If $N \equiv x^U$, then $e(x^U) \in [\![U]\!]$ by definition.

2. If $N \equiv \varepsilon$, since $[\![\varepsilon]\!] = \mathcal{D}$, then $[\![\varepsilon]\!] \Subset [\![\{\}]\!] \equiv \mathcal{D} \times \mathcal{D}$ (see 6.4.2).

3. If $N \equiv \lambda x^U.M^V$, consider $[\![\lambda x^U.M^V]\!]_e = f$ such that $\forall u \Subset [\![U]\!].f(u) = [\![M^V]\!]_{e[u:=x]}$. We need to show that f lives in the right type and it is realized, or, equivalently, that f, as a set of realizers, is in (the quotient set of) $[\![U \to V]\!]$. This is a consequence of the proof that **PER** is a CCC, as lambda abstraction is the currying operation (see for instance [AL91]).

[5] We should write $a_min\{\ldots\}$, but note that in this case the set $\min\{\ldots\}$ is a singleton.

4. If $N \equiv M^{U \to V} P^W$, where $W \leq U$. By induction $[\![M^{U\to V}]\!]_e \Subset [\![U \to V]\!]$ and $[\![P^W]\!]_e \Subset [\![W]\!] \leq [\![U]\!]$. Then the result follows by 6.4.8.

5. Assume that $N \equiv (M_1 \&^{\{V_i \to W_i\}_{i\leq n}} M_2)\colon \{U \to out(U)\}_{U \in \{V_i\}_{i\leq n}}$ with $M_1\colon T_1 \leq \{V_i \to W_i\}_{i<n}$ and $M_2\colon T_2 \leq V_n \to W_n$. Set $[\![M_1 \& M_2]\!]_e = f$ as in the definition 6.4.15. We prove this case by induction on n; in each case we have to prove that f lives in the set-theoretic indexed product $\bigotimes_{j \in [\widehat{\{V_i\}_{i\leq n}}]} [\![\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j)]\!]$ and that it is realized.

   $(n = 1)$. In this case $\{V_i\}_{i\leq n} = \{V_1\}$ By definition 6.4.15 we have that $\forall j \in [\widehat{\{V_1\}}]. f(j) = c_{T_2(\mathcal{T}_j \to W_1)}([\![M_2]\!]_e)$ as there is only one branch to choose. By induction $[\![M_2]\!]_e \Subset [\![T_2]\!]$ and, since $\mathcal{T}_j \leq V_1$, the coercion application makes sense. The result lives in the correct type since, for each $j \in [\widehat{\{V_1\}}]$, $f(j) = c_{T_2(\mathcal{T}_j \to W_1)}([\![M_2]\!]_e) \in [\![\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j)]\!]$ and, thus, the whole f lives in the correct set-theoretic indexed product. To conclude the proof we have to show that f is realizable.

   Intuitively, f can be realized by an index for a constant function, since for any input type f executes always the same code M_2; this code is coerced to different types $\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j)$, but, thanks to the interpretation of subtyping, any realizer for this code in the minimum type T_2 lives in the domain of any of its supertypes $\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j)$.

   Formally, consider first a map $f'\colon [\widehat{\{V_1\}}] \to \mathcal{P}\omega$ which chooses, for each $j \in [\widehat{\{V_1\}}]$, always the same element n of the equivalence class $[\![M_2]\!]_e \in [\![T_2]\!]$. Then f' is computed by any index of the chosen constant function. These indexes realize f: since $f'(j)$ lives in $[\![M_2^T]\!]_e$, then

$$\lceil f'(j) \rceil_{[\![\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j)]\!]} = c_{T, \mathcal{T}_j \to \widehat{out}(\mathcal{T}_j)} [\![M_2^T]\!]_e = f(j).$$

   Note that, though f is realized by an index of a constant function, f itself is *not* constant.

   $(n > 1)$. In this case, we have that $M_1\colon T_1 \leq \{V_i \to W_i\}_{i<n} \equiv \{U \to out(U)\}_{U \in \{V_i\}_{i<n}}$. By induction hypothesis, $[\![M_1]\!]_e \Subset [\![T_1]\!]$ and thus $c_{T_1(\{V_i \to W_i\}_{i<n})}([\![M_1]\!]_e) \Subset [\![\{V_i \to W_i\}_{i<n}]\!]$. In particular, let g be the map such that for each $j \in [\widehat{\{V_i\}_{i<n}}]. g(j) = c_{T_1(\{V_i \to W_i\}_{i<n})}([\![M_1]\!]_e)(j)$. Then, by the induction, g is in $\prod_{j \in [\widehat{\{V_i\}_{i<n}}]} [\![\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j)]\!]$. Consider now f defined as in 6.4.15 point 6 from $(M_1 \&^{\{V_i \to W_i\}_{i\leq n}} M_2)$, that is, $f(j) = c_{T_2(\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j))}([\![M_2]\!]_e)$ or $f(j) = g(j)$ according to whether $Z = \min\{U \in \{V_i\}_{i\leq n} | \mathcal{T}_j \leq U\}$ is equal to V_n or to some V_m for $m < n$. Note that f is well defined, as, for each $j \in [\widehat{\{V_i\}_{i\leq n}}]$ there exists (and it is unique) the least V_z such that $\mathcal{T}_j \leq V_z$, by the definition of well-formed overloaded types. By this, f is in (the quotient set of) $\bigotimes_{j \in [\widehat{\{V_i\}_{i\leq n}}]} [\![\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j)]\!]$.

   To prove that f is realizable, consider the map f' which chooses for each $j \in [\widehat{\{V_i\}_{i\leq n}}]$ an element $f'(j)$ of the equivalence class $f(j)$. Clearly, f' is (well defined and) continuous, since $[\widehat{\{V_i\}_{i\leq n}}] \subseteq \mathcal{P}\omega$ is endowed with the discrete topology. Then its continuous extension $\underline{f'}\colon \mathcal{P}\omega \to \mathcal{P}\omega$, given as in the lemma, is computed

by some $n \in \mathcal{P}\omega$, i.e. $\underline{f'}(p) = n \cdot p$, see [Sco76]. In conclusion,

$$\exists n \in \mathcal{P}\omega.\ \forall j \in [\widehat{\{V_i\}_{i\leq n}}].\ f(j) = \lceil f'(j) \rceil_{[\![\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j)]\!]} = \lceil n \cdot j \rceil_{[\![\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j)]\!]}$$

and thus $f \in \prod_{j \in [\widehat{\{V_i\}_{i\leq n}}]} [\![\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j)]\!]$. (Equivalently, one could extend by continuity g, defined as above, to f, by the same "injectivity" argument; this argument is needed, anyway, as an extension by a constant value, does not need to be continuous, in general. We preferred to define a new realizer and use the inductive hypothesis just to check the semantic types).

6. If $N \equiv M^{\{U \to out(U)\}_{U \in K}} \bullet \mathbf{coerce}^V(P^W) : out(Z)$ where $Z = \min\{U \in K | V \leq U\}$, then

$$[\![N]\!] = [\![M^{\{U \to out(U)\}_{U \in K}}]\!]_e(j) c_{WV}([\![P^W]\!]) \text{ for } \mathcal{T}_j = V.$$

By the previous point, $[\![M^{\{U \to out(U)\}_{U \in K}}]\!]_e(j) \Subset [\![\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j)]\!] \equiv [\![V \to out(Z)]\!]$, hence

$$[\![N]\!]_e \Subset [\![out(Z)]\!]$$

□

We observed that reductions are not interpreted as equalities in the model. Indeed they yield set theoretic inclusions.

Lemma 6.4.18 (substitution) *$[\![Q[x := P]]\!]_e \subseteq [\![Q]\!]_{e[x:=p]}$ where $p = c_{T'T}[\![P]\!]_e$, $x : T$, $P : T' \leq T$*

Proof. The proof goes by induction on the structure of the terms:

1. $Q \equiv y \neq x$ or $Q \equiv \varepsilon$: trivial

2. $Q \equiv x$

   $[\![x[x := P]]\!]_e = [\![P]\!]_e \Subset [\![T']\!]$ is contained in $[\![x]\!]_{e[x:=p]} = p = c_{T'T}[\![P]\!]_e \Subset [\![T]\!]$, by the definition of semantic coercions.

3. $Q \equiv \lambda y^U.M$

   As usual we identify a function with the set of its realizers, in the intended type. Thus

   $$[\![\lambda y^U.M[x := P]]\!]_e = f \text{ such that } \forall u \Subset [\![U]\!].f(u) = [\![M[x := P]]\!]_{e[y:=u]}$$

   while

   $$[\![\lambda y^U.M]\!]_{e[x:=p]} = f' \text{ such that } \forall u \Subset [\![U]\!].f'(u) = [\![M]\!]_{e[x:=p;y:=u]} \supseteq [\![M[x := P]]\!]_{e[y:=u]}$$

   (the inclusion holds by induction hypothesis), thus each realizer for f is also a realizer for f', that is

   $$[\![\lambda y^U.M[x := P]]\!]_e \subseteq [\![\lambda y^U.M]\!]_{e[x:=p]}$$

4. $Q \equiv M \cdot N$

   the proof goes by the usual induction. Just recall 6.4.10 and the monotonicity of application (Lemma 6.4.9).

5. $Q \equiv M \bullet \mathbf{coerce}^V(N)$

$$
\begin{array}{lll}
\multicolumn{2}{l}{[\![M \bullet \mathbf{coerce}^V(N)]\!]_{e[x:=p]} =} & \\
= & [\![M]\!]_{e[x:=p]}(j)([\![\mathbf{coerce}^V(N)]\!]_{e[x:=p]}) & \text{for } \mathcal{T}_j = V \\
\supseteq & [\![M[x:=P]]\!]_e(j)([\![\mathbf{coerce}^V(N)[x:=P]]\!]_e) & \text{by ind. and the monotonicity of appl. in } P\omega \\
= & [\![(M[x:=P]) \bullet (\mathbf{coerce}^V(N)[x:=P])]\!]_e & \\
= & [\![(M \bullet \mathbf{coerce}^V(N))[x:=P]]\!]_e &
\end{array}
$$

6. $Q \equiv M_1 \& M_2 \colon \{V_i \to W_i\}_{i \leq n}$. Let $Z = \min\{U \in \{V_i\}_{i \leq n} | \mathcal{T}_j \leq U\}$ then

$$
[\![M_1 \& M_2]\!]_{e[x:=p]} = f \text{ such that } f(j) = \begin{cases} c_{T_2(\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j))}([\![M_2]\!]_{e[x:=p]}) & \text{if } Z = V_n \\ (c_{T_1(\{V_i \to W_i\}_{i<n})}[\![M_1]\!]_{e[x:=p]})(j) & \text{else} \end{cases}
$$

$$
[\![(M_1 \& M_2)[x:=P]]\!]_e = f' \text{ s.t. } f'(j) = \begin{cases} c_{T_2(\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j))}([\![M_2[x:=P]]\!]_e) & \text{if } Z = V_n \\ (c_{T_1(\{V_i \to W_i\}_{i<n})}[\![M_1[x:=P]]\!]_e)(j) & \text{else} \end{cases}
$$

   Note that in both cases we get the same Z, since it depends only on the index of the &. Then the result follows by the same reasoning as in (3.). More precisely, since in this case $[\![Q[x:=P]]\!]_e$ and $[\![Q]\!]_{e[x:=p]}$ have the same type, from $[\![Q[x:=P]]\!]_e \subseteq [\![Q]\!]_{e[x:=p]}$ we can deduce $[\![Q[x:=P]]\!]_e = [\![Q]\!]_{e[x:=p]}$.

□

The immediate consequence of the work done so far is the construction of a simple model of "reduction" and not, as customary in denotational semantics, of "conversion". This is precisely stated by the following theorem.

Theorem 6.4.19 (soundness wrt reductions) *If M is well typed and $M \rhd N$ then $[\![M]\!]_e \supseteq [\![N]\!]_e$*

Proof. The proof goes by induction on the definition of $\rhd$. The critical cases are:

1. $M \equiv (\lambda x^U.P)Q^W$ and $N \equiv P[x:=Q]$ with $(\lambda x^U.P) \colon U \to V$ and $Q \colon W \leq U$

$$
\begin{array}{rcll}
[\![(\lambda x^U.P)Q]\!]_e & = & [\![(\lambda x^U.P)]\!]_e[\![Q]\!]_e & \\
& = & [\![(\lambda x^U.P)]\!]_e c_{WU}([\![Q]\!]_e) & \text{by 6.4.10} \\
& = & [\![P]\!]_{e[x:=q]} & \text{where } q = c_{WU}([\![Q]\!]_e) \\
& \supseteq & [\![P[x:=Q]]\!]_e & \text{by the substitution lemma} \\
& = & [\![N]\!]_e &
\end{array}
$$

2. $M \equiv (M_1 \& M_2) \bullet P$ with $(M_1 \&^{\{V_i \to W_i\}_{i \leq n}} M_2): \{U \to out(U)\}_{U \in \{V_i\}_{i \leq n}}$, $M_1: T_1 \leq \{V_i \to W_i\}_{i<n}$, $M_2: T_2 \leq V_n \to W_n$ and $P \equiv \mathbf{coerce}^{\mathcal{T}_j}(P'): \mathcal{T}_j$ for some $P' \in$**Terms** and $\mathcal{T}_j \in$**Types**. Thus, $N \equiv M_h P$ for $h = 1$ or $h = 2$. More precisely, for $Z = \min\{U \in \{V_i\}_{i \leq n} | \mathcal{T}_j \leq U\}$,

$$N \equiv \begin{cases} M_2 \cdot P & \text{if } Z = V_n \\ M_1 \bullet P & \text{else} \end{cases}$$

Compute then

$$\begin{array}{rcl} [\![(M_1 \& M_2) \bullet P^{\mathcal{T}_j}]\!]_e & = & [\![(M_1 \& M_2)]\!]_e(j)[\![P]\!]_e \\ & = & \begin{cases} c_{T_2(\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j))}([\![M_2]\!]_e)[\![P]\!]_e & \text{if } Z = V_n \\ (c_{T_1(\{V_i \to W_i\}_{i<n})}[\![M_1]\!]_e)(j)[\![P]\!]_e & \text{else} \end{cases} \end{array}$$

In the first case,

$$\begin{array}{rcll} [\![M]\!]_e & = & c_{T_2(\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j))}([\![M_2]\!]_e)[\![P]\!]_e & \\ & \supseteq & ([\![M_2]\!]_e)[\![P]\!]_e & \text{by monotonicity and by } a \subseteq c_{UV}(a) \\ & = & [\![N]\!]_e & \end{array}$$

Otherwise

$$\begin{array}{rcll} [\![M]\!]_e & = & (c_{T_1(\{V_i \to W_i\}_{i<n})}[\![M_1]\!]_e)(j)[\![P]\!]_e & \\ & \supseteq & [\![M_1]\!]_e(j)[\![P]\!]_e & \text{by monotonicity and by } a \subseteq c_{UV}(a) \\ & = & [\![M_1 \bullet P]\!]_e & \\ & = & [\![N]\!]_e & \end{array}$$

□

Remark Clearly, theorem 6.4.19 specializes to the implicative fragment of our calculus, which is simply typed λ-calculus with subtyping. Thus, by a simple observation of the properties of **PER**, we spotted a mathematical model of the reduction predicate "$\triangleright$" between terms of λ-calculi, instead of conversion "=". The non-syntactic models so far constructed could only give mathematical meaning to the theory of "=" between λ-terms and β-reduction was interpreted as the "=".

It is important to notice, however, that the decrease of the size of the equivalence class which is the interpretation of a term is not directly related to the reduction process, but to the fact that types decrease during computation. In fact, if you consider the two terms M and $\mathbf{coerce}^V(M)$ and apply the same reduction steps to both of them, while the semantics of M can decrease, any time its type changes, the semantics of $\mathbf{coerce}^V(M)$ remain fixed, even if the same reduction steps are executed.

6.5 Summary of the semantics

As already mentioned in the introduction of this chapter, there is a general understanding that polymorphism, as intended in λ-calculus, is not compatible with "procedures depending

on input types". As pointed out in [Gir72], one cannot extend second order λ-calculus with a term giving different output terms according to different input types. Indeed, in [LMS93], it is shown that terms depending on types use types as "generic", i.e. the value on just one type determines the value everywhere. This is why, in order to express an explicit type dependency, it was not sufficient to extend simply typed λ-calculus by type variables, and we proposed an entirely new feature, based on "finite branching of terms", in order to formalize the dependency we wanted. Moreover, the use of late binding and subtyping added expressiveness to the system. Indeed, as we said in the introduction of this chapter, the expressive power of the syntax poses four problems: pre-order, type-dependent computation, late binding, impredicativity. We have handled the first two and circumvented the others. More in detail:

1. The use of a pre-order between types is handled, at a syntactic level in section 6.2, by the notion of completion. But then our finite branching immediately becomes an infinite one: this is indeed what is actually meant in the syntax, by the rules, as we allow terms to work also on inputs inhabiting types *smaller* than the intended one. Thus, the intended function depending on the type of the input, may depend on an infinity of input types, implicitly. This must be made explicit in the semantics.

2. Type dependent computations are handled by considering types as "coded" in the semantics and using their indexes also as meaning in the model. Note that this mathematical meaning of types corresponds to the practice of type-dependent computations. In programming, when and if computing depends on types, this is possible as types, after all, are just "code" (in OOP it corresponds to consider classes as *tags* as we saw in chapter 5); thus they are handled like any countable (and enumerated) data type. This is impossible in sound mathematical models which respect the logical "second order" convention. Indeed, in this case, types must be (arbitrary) subsets of the (infinite) sets interpreting terms. Observe also that the implicit polymorphism of our approach shows up in the semantics by the interpretation of overloaded functions as elements of an *infinite* indexed product.

3. We were not able to deal with late binding: suppose that you want to interpret the overloaded application $M \bullet N$ where N is not in normal form; the semantics will be of the form $((\llbracket M \rrbracket)\llbracket T \rrbracket)\llbracket N \rrbracket$, where T is the syntactic type on which the selection of the branch is performed. With early binding we know that T is the type of N, but what should we use with late binding? T must be the run-time type of N but we cannot know it yet. The solution probably consists in giving the semantics of a term as the pair (computation, runtime type) of the term. Thus the computation part of $\llbracket M \bullet N \rrbracket$ would be something of the form

   $$(\Lambda X.(\mathrm{fst}\llbracket M \rrbracket)\llbracket X \rrbracket)\,(\mathrm{snd}\llbracket N \rrbracket)\,(\mathrm{fst}\llbracket N \rrbracket)$$

   As we see we are passing to a formalism where type dependency is explicit and we think that a better understanding, already at syntactic level, of this explicit type dependency would greatly help us in the study of the mathematical meaning of $\lambda\&$. This indeed was (chronologically) the first motivation that led us to study the second order formalisms we present in the next part of the thesis.

4. We were not able to deal with the new form of impredicativity introduced by the definition of subtyping for the overloaded types. We already showed the problem in the introduction of this chapter: the meaning of $\{\{\} \to T\}$ is the product indexed on the subtypes of $\{\}$ and thus on $\{\{\} \to T\}$ itself. The problem is similar to the one of System F: the semantics of $\forall X.T$ is the indexed product on the semantics of all types and thus of $\forall X.T$ itself. In PER, for example, the problem is solved (but it took nearly two decades to clarify its categorical meaning) by indexing not on the interpretations of the types, but on all per's:

$$[\![\forall X.T]\!]_E = \prod_{C \in \mathrm{PER}} [\![T]\!]_{E[X:=C]}$$

Thus the definition above is well given since the per's exists independently from the types they interpret (see [LM91]). The same happens with $F_{\leq}$ (section 7.1.3): the meaning of $\forall X \leq S.T$ is the indexed product on all the *sub-per's* of the meaning of S (see [CL91a]):

$$[\![\forall X \leq S.T]\!]_E = \prod_{C \subseteq [\![S]\!]_E} [\![T]\!]_{E[X:=C]}$$

Here we are not able to mimic this understanding of the intended circularity because we are forced to index our product on singletons as "codes" for types. Thus we are not able to define in the model a sound order on singleton that respects the subtyping relation of the corresponding types. In other terms in the models of $F_{\leq}$ one is able to define an order $\subseteq$ on PERs, such that if $S \leq T$ then $[\![S]\!] \subseteq [\![T]\!]$. But here we could not find *in the model* an order relation $\preccurlyeq$ on singletons (or more general on sets) such that if $\mathcal{T}_n \leq \mathcal{T}_m$ then $\{n\} \preccurlyeq \{m\}$.

In his PhD. Thesis [Tsu92] Hideki Tsuiki introduces a calculus similar to λ& but which models just the *coherent overloading*. This kind of overloading has the restriction that the definition of branches with related input types must be related. For example, define a overloaded function with two branches $(M_1 \,\&\, M_2)$ with $M_1{:}\, Int \to T$ and $M_2{:}\, Real \to T$, since $Int \leq Real$ "coherent overloading" requires that for all $N{:}\, Int$ $M_1N = M_2N$ (this is for example what happens with the successor function). If this condition is not satisfied then the computation ends with an error (note that this error cannot be statically detected since $M_1N = M_2N$ is undecidable, see more on it in section 11.2). Also in his work Tsuiki meets the problem of impredicativity we explained above (see also section 5.2.4 of [Tsu92]), but he can give a mathematical meaning to it thanks the strong relation of the various branches. For that he exploits some tools also used in constructing a domain theoretic model of the higher order λ-calculus. Thus, as also he says, his "merge" function is closer to a parametric polymorphic function than to an overloaded one, and the impredicativity due to subtyping is still an open problem for general type dependent computations.

We believe that the same could be done with λ&; indeed at the very beginning of this semantic study Pino Rosolini suggested us a model to interpret λ& with restriction to coherent overloading; though we preferred to follow the solution of early binding in view of the undecidable nature of the restriction.

Part II

Second order

Chapter 7

Introduction to part II

This second part of the thesis is completely dedicated to polymorphic systems.

Polymorphism in programming has been introduced for a variety of reasons and greatly contributed to a modular, flexible style of programming. Its relevance has been stressed also by the increase of reliability of polymorphic programs, in view of its connections to static type checking. Here we stress the further motivations which derived from the use of the "object as record" analogy and that are originated by the problem of "loss of information", explained below.

Also in our case the main motivation to the introduction of polymorphism is originated by this problem. However there are also other reasons that induced us to study the integration of bounded parametric and "ad hoc" polymorphism: one of this motivation is the semantic understanding of the mechanism of late binding, and has been illustrated in the conclusion of last chapter; other proof theoretic motivations will be illustrated in chapter 9.

In this thesis we limit our study to second order type systems in which the polymorphism is *explicit*, i.e. where terms can be applied to types. The study of the *implicit* counterpart is under way.

We initially proceed in our study of the second order by following two parallel directions:

1. We try to ameliorate the existing type systems that use bounded quantification, by defining a variant enjoying many properties (foremost the decidability of the subtyping relation) that actual type systems do not. This is the topic of chapter 8

2. We define a calculus for explicit "ad hoc" polymorphism. Namely a calculus in which overloaded functions are applied to types and the selection is based on the type *at* the argument (rather than on the type *of* the argument). This is the topic of chapter 9.

After having merged the results of the two studies in a single calculus, we show how the second order overloading can be used to model object-oriented programming (chapter 10).

7.1 The loss of information in the record-based models: a short history

The problem of "loss of information", has been defined by Luca Cardelli in [Car88], where he also forecasted the solution by polymorphic types. Let show it by an example.

Consider the identity function $I_{\langle\!\langle \ell : Int \rangle\!\rangle}$ of type $\langle\!\langle \ell : Int \rangle\!\rangle \to \langle\!\langle \ell : Int \rangle\!\rangle$. In the syntax of typed lambda terms this corresponds to:

$$I_{\langle\!\langle \ell : Int \rangle\!\rangle} \equiv \lambda x^{\langle\!\langle \ell : Int \rangle\!\rangle}.x$$

Take now $M \equiv \langle \ell = 1; m = true \rangle$ which has type $\langle\!\langle \ell : Int; m : Bool \rangle\!\rangle$. By subsumption $M : \langle\!\langle \ell : Int \rangle\!\rangle$, thus $I_{\langle\!\langle \ell : Int \rangle\!\rangle}$ can be applied to it (equivalently use $[\to\text{ELIM}_{(\leq)}]$). This gives $(I_{\langle\!\langle \ell : Int \rangle\!\rangle}(M)) : \langle\!\langle \ell : Int \rangle\!\rangle$ and therefore $(I_{\langle\!\langle \ell : Int \rangle\!\rangle}(M)).m$ returns a type error. In other words, we have *lost the information* bound to the label "m" of M simply by applying it to the identity function.

The point is that the type of the result of a function is fixed before the function is applied; thus it cannot depend on the type of the input, as one would want. The solution is given by the use of *type variables* in its two main forms: type assignment (implicit polymorphism) and higher order type checking (explicit polymorphism).

7.1.1 Implicit Polymorphism

The idea is that terms possesses type schemata that may be consistently instantiated, in the usual sense of Logic [Mil78, Hin69]. For example, the identity $\lambda x.x$ has *type schema* $\forall \alpha.\alpha \to \alpha$, i.e. it has all the types obtained as instances of $\alpha \to \alpha$. The main features of this approach are widely discussed in the literature. The basic idea is that programs are type-free entities, which are assigned a type at compile time. In our example, there is no loss of information, as one may instantiate $\alpha \to \alpha$ by the intended type of the input obtaining the same type as output.

Observe though that in our case there are some complications. *Generic type variables* may have to satisfy some further constraints as for

$$M \equiv \lambda x.((\lambda y.x)((x.\ell) + 3))$$

In this example, M β-reduces to $\lambda x.x$. However, the use of field selection in $(x.\ell)$, impose that M may only accept parameters possessing *at least* a field "ℓ" (of type *Int* since it is added to 3). Thus M is assigned the type schema $\forall \alpha.\alpha \to \alpha$ with $\alpha \leq \langle\!\langle \ell : Int \rangle\!\rangle$ (which can be noted by $\forall \alpha \leq \langle\!\langle \ell : Int \rangle\!\rangle.\alpha \to \alpha$).

After the introduction of polymorphism the need was felt to add the definitions of operations on records which were more expressive then the ones in [CW85, Car88]. The problem was that, at that stage, only the functions that *read* a record could profit from polymorphism; functions modifying a field of a record did not take advantage from the use of type variables.

Along this line, powerful calculi of records (as the one given in [CM91]) or very expressing encodings (like the ones in [CL91a] or in [Car92]) have been defined. However, these calculi did not permit one to infer the principal type scheme of a type-free term. In the context of a type-assignment algorithm other calculi were developed by adding functional constants to the functional core of ML and using *Kinds* in the algorithm [Rém89, Wan91].

7.1.2 Explicit Polymorphism

The other idea is based on a higher order understanding of the approach above, as polymorphism is *explicit* rather than *implicit*. In other words, type variables, instead of being implicitly quantified (by an external, metalinguistic universal quantifier), are explicitly quantified by a linguistic, second order quantifier, as in second order λ-calculus of Girard and Reynolds [Gir72, Rey74]; moreover bounds can be imposed on the quantified type variables obtaining in this way the so-called *Bounded Quantification*.

The approach of bounded quantification, which originated in [CW85] by the definition of the language Fun, explicitly blends polymorphism and subtyping by allowing bounds (inclusion) over quantification. That is, the syntax is given by adding to the second order lambda-calculus constraints (bounds) on the lambda-abstracted type variables. In short, terms and types are defined as follows:

$$\begin{array}{lcl} a & ::= & x \mid (\lambda x^T.a) \mid a(a) \\ & \mid & \mathsf{top} \mid \Lambda X{\leq}T.a \mid a(T) \end{array}$$

$$T \quad ::= \quad X \mid \mathsf{Top} \mid T \to T \mid \forall(X{\leq}T)T$$

The second order λ-calculus is then the particular case where the type which bounds the type-variables is always equal to Top.

In this context, the function $I_{\langle\langle\ell:Int\rangle\rangle}$ in the previous section is written as[1] $\Lambda X{\leq}\langle\langle\ell\colon Int\rangle\rangle.\lambda x^X.x$ and it has type $\forall(X{\leq}\langle\langle\ell\colon Int\rangle\rangle)X \to X$. Then we may apply $I_{\langle\langle\ell:Int\rangle\rangle}$ to M by first applying it to the actual type of M:

$$I_{\langle\langle\ell:Int\rangle\rangle}(\langle\langle\ell\colon Int; m\colon Bool\rangle\rangle)(M)$$

The result has type $\langle\langle\ell\colon Int; m\colon Bool\rangle\rangle$ and there is no loss of information. The different expressiveness of Bounded Quantification w.r.t. implicit polymorphism may be understood by the following example: in ML+records (see [Rém89]) there is no way to assign a type to the function

$$\lambda f.f(\langle\ell = 3; \ell' = true\rangle) + f(\langle\ell = 5; \ell' = 6\rangle)$$

This is due to non genericity of λ-bound variables. In the apprach with Bounded Quantification the term above may be seen as the *erasure* of the (well-typed) term

$$\begin{array}{l}\lambda f^{\forall(X\leq\langle\langle\ell:Int\rangle\rangle)X\to Int}. \\ \qquad f(\langle\langle\ell\colon Int; \ell'\colon Bool\rangle\rangle)(\langle\ell = 3; \ell' = true\rangle) + f(\langle\langle\ell\colon Int; \ell'\colon Int\rangle\rangle)(\langle\ell = 3; \ell' = 6\rangle)\end{array}$$

This expressiveness is paid by the loss of an algorithm of type assignment. It may be fair to say, however, that, in large scale programming—where object-oriented languages play their best role—type inference is not a primary objective. Indeed, if types are not directly specified by the programmer they soon get unmanageable. Thus, while a small amount of type inference is desirable in an object-oriented language, a large use of type inference could result harmful. assignment doesn't seem to fit to large-scale programming (types soon get unmanageable if not) where object-oriented languages play their best role.

[1]Records can be encoded

A final remark: note that we now use lowercase letters $(a, b \dots)$ to range over terms. This is the usual convention in the second order explicit polymorphism. Thus, in the rest of this thesis, we abandon uppercase letters M, N, ..., to adopt the standard convention.

7.1.3 $F_{\leq}$

Fun was further developed in many works (a non exhaustive list includes [CCH+89, Ghe90, CHC90, CL91a, CMMS91, BTCGS91, Bru92, KS92, PT93]) and, in particular that of Curien and Ghelli [CG92] where they give the standard definition of bounded quantification called $F_{\leq}$ ("F-sub"). This calculus is the starting point of the work in this second part of the thesis. Thus let us describe it more in detail.

The terms and the types of $F_{\leq}$ are those defined for Fun in the previous section. The subtyping and typing rules are given below:

(refl) $$C \vdash T \leq T$$

(trans) $$\frac{C \vdash T_1 \leq T_2 \qquad C \vdash T_2 \leq T_3}{C \vdash T_1 \leq T_3}$$

(taut) $$C \vdash X \leq C(X)$$

(Top) $$C \vdash T \leq \mathsf{Top}$$

($\to$) $$\frac{C \vdash T_1 \leq S_1 \qquad C \vdash S_2 \leq T_2}{C \vdash S_1 \to S_2 \leq T_1 \to T_2}$$

($\forall$) $$\frac{C \vdash T_1 \leq S_1 \qquad C \cup \{X \leq T_1\} \vdash S_2 \leq T_2}{C \vdash \forall(X \leq S_1)S_2 \leq \forall(X \leq T_1)T_2}$$

[Vars] $$C \vdash x^T : T$$

[$\to$Intro] $$\frac{C \vdash a : T'}{C \vdash (\lambda x^T.a) : T \to T'}$$

[$\to$Elim] $$\frac{C \vdash a : S \to T \qquad C \vdash b : S}{C \vdash a(b) : T}$$

[Top] $$C \vdash \mathsf{top} : \mathsf{Top}$$

[$\forall$Intro] $$\frac{C \cup \{X \leq T\} \vdash a : T'}{C \vdash \Lambda X \leq T.a : \forall(X \leq T)T'}$$

[∀ELIM] $$\frac{C \vdash a{:}\forall(X{\leq}S)T \qquad C \vdash S'{\leq}S}{C \vdash a(S'){:}T[X := S']}$$

[SUBSUMPTION] $$\frac{C \vdash a{:}T' \qquad C \vdash T' \leq T}{C \vdash a{:}T}$$

The typing rules are quite standard; note the use of subsumption. Among the subtyping rules note the rule (∀); the meaning of this rule and the consequences of using it will be broadly discussed in the next chapter, and constitute the starting point of the definition of $F_{\leq}^{\top}$.

Chapter 8

A roadmap to decidable bounded quantification

It is an important and popular fact that things are not always what they seem. For instance, on the planet Earth, man had always assumed that he was more intelligent than dolphins because he had achieved so much —the wheel, New York, wars and so on— while all the dolphins had ever done was muck about in the water having a good time. But conversely, the dolphins had always believed that they were far more intelligent than man—for precisely the same reasons.

Douglas Adams
The hitchhiker's guide to the galaxy

Extensions of System F with bounded quantification form the basis of much recent research on the foundations of programming languages. But the standard formulation of bounded quantification, $F_{\leq}$, is difficult to work with and lacks some important syntactic properties, such as decidability. More tractable variants have been studied, but those proposed to date either exclude significant classes of useful programs or lack a compelling semantic intuition.

We propose in this chapter a simple variant of $F_{\leq}$ that ameliorates these difficulties. It has a simple semantic interpretation, enjoys a number of important properties that fail in $F_{\leq}$, and includes all of the programming examples for which $F_{\leq}$ has been used in practice.

This chapter is a joint work with Benjamin Pierce, [CP94].

8.1 Introduction

In the previous chapter we gave the subtyping rules of $F_{\leq}$, defined by Curien and Ghelli in [CG92]. The definition of its subtype relation includes the following rule for comparing polymorphic types:

$$(\forall\text{-orig}) \qquad \frac{C \vdash T_1 \leq S_1 \qquad C \cup \{X \leq T_1\} \vdash S_2 \leq T_2}{C \vdash \forall(X \leq S_1)S_2 \leq \forall(X \leq T_1)T_2} \qquad X \notin dom(C)$$

(we added the suffix -orig to distinguish it from the other rules we describe in the following) Intuitively, a type T of the form $\forall(X \leq T_1)T_2$ describes a collection of polymorphic values (functions from types to values) each mapping subtypes of T_1 to instances of T_2. If T_1 is a subtype of S_1, then the domain of T is smaller than that of $\forall(X \leq S_1)T_2$, so this type has a weaker constraint and therefore it describes a larger collection of polymorphic values. Moreover, if given the type $\forall(X \leq S_1)S_2$ for each type U that is an acceptable argument to the functions in both collections (i.e., one that satisfies the more stringent requirement $U \leq T_1$), the U-instance of S_2 is a subtype of the U-instance of T_2, then T is a "pointwise weaker" constraint and again describes a larger collection of polymorphic values. In other terms the two functions are pointwise compared only on their common domain.

Though it is semantically quite natural, this rule is responsible for the loss of numerous desirable syntactic properties. To begin with, the subtype relation of $F_{\leq}$ is undecidable [Pie93, Ghe93b], which implies the undecidability of the typechecking problem even for explicitly typed terms. Moreover, $F_{\leq}$ lacks greatest lower bounds for compatible (i.e. lower-bounded) sets of types [Ghe90], which blocks certain useful forms of argument by induction on the subtype relation. The most natural extension of $F_{\leq}$ with bounded existential types [GP93] even fails to possess a minimal type for every typeable term! In all these cases, it is the quantifier subtyping rule that appears to be the principal culprit. Thus, it is reasonable to look for variants of this rule with better properties.

The crux of the problem is that the upper bound of the bound variable X in S_2 changes from S_1 in the rule's conclusion to T_1 in the right-hand premise. This "re-bounding" of variables is syntactically rather bizarre; in particular, it invalidates a whole class of arguments based on structural induction on types, where the case for a type variable normally requires applying the induction hypothesis to its upper bound.

The weaker "equal-bounds subtyping rule" from Cardelli and Wegner's original Fun calculus [CW85]

$$(\forall\text{-Fun}) \qquad \frac{C \cup \{X \leq U\} \vdash S_2 \leq T_2}{C \vdash \forall(X \leq U)S_2 \leq \forall(X \leq U)T_2} \qquad X \notin dom(C)$$

avoids this confusion and yields a system in which the subtyping problem can easily be shown to be decidable, at the cost of introducing an ugly syntactic restriction —that the bounds of the two types must be *identical*— with no natural semantic motivation.

Other variants of (∀-orig) have been described in the literature. Katiyar and Shankar [KS92] propose a restriction in which the bounds on polymorphic types may not contain Top. In this way they obtain a decidable subtype relation, but at significant expense in expressiveness: indeed in system with bounded quantification record types are encoded by tuples ending by Top (see [CL91a, Car92]); therefore in this variation it is not possible to quantify over record types, and thus it is not very interesting for object-oriented programming.

Another obvious variation uses the bound S_1 of the smaller type in place of T_1 in the right-hand premise:

$$(\forall\text{-local}) \qquad \frac{C \vdash T_1 \leq S_1 \qquad C \cup \{X \leq S_1\} \vdash S_2 \leq T_2}{C \vdash \forall(X \leq S_1)S_2 \leq \forall(X \leq T_1)T_2} \qquad X \notin dom(C)$$

But Giorgio Ghelli has pointed out [Ghe93c] that this variant is algorithmically impractical, although the problem of its decidability remains open.

Our aim here is to propose a simple and appealing alternative and study the properties of the resulting system. We use the quantifier subtyping rule

$$(\forall\text{-new}) \qquad \frac{C \vdash T_1 \leq S_1 \qquad C \cup \{X \leq \mathsf{Top}\} \vdash S_2 \leq T_2}{C \vdash \forall(X \leq S_1)S_2 \leq \forall(X \leq T_1)T_2} \qquad X \notin dom(C)$$

in which the right-hand premise requires that the bodies be (covariantly) related under *no* assumption about the bound variable. This essentially amounts to considering the subtyping relation relative to an unchanging context, since the type variables added to the context always have trivial bounds; the only type variables with interesting bounds will be those already present in the environment at the point where a subtyping check is required. (These are introduced, as usual, by the quantifier introduction rule).

Clearly, the proposed rule is strictly weaker than ($\forall$-orig). The system with ($\forall$-new) in place of ($\forall$-orig) (we call it $F_{\leq}^{\top}$ and give its full definition in in Section 8.2) cannot be used to prove Fsub-inequalities like

$$\vdash \forall(X \leq T)X \leq \forall(X \leq T)T$$

However, as we show in Section 8.3, this difference in power does not matter in any of the situations where bounded quantification has been used. Moreover, the rule ($\forall$-new) is arguably more natural than the other variants we have mentioned, since it embodies a notion of pointwise subtyping already familiar from the treatment of other type constructors [Mit90b, BM92, Bru93, PT93]: it simply says that $\forall F \leq \forall G$ (with $F, G : \mathbf{Type} \rightarrow \mathbf{Type}$) iff $dom(G) \subseteq dom(F)$ and F is pointwise smaller than G.

Most importantly, $F_{\leq}^{\top}$ enjoys many of the syntactic properties missing from $F_{\leq}$. First and foremost, its subtyping problem is decidable, as we show in Section 8.4; moreover, the system has least upper bounds for all pairs of types and, for all lower-bounded pairs, greatest lower bounds.

The easier syntactic formulation leads to a significant simplification of the proofs for this system. The deep motivation of this simplification resides in the decidability of subtyping; we have then a measure that decreases with the (backward) application of the subtyping rules. This allows us to perform proofs by induction on the depth of the derivations. For example, Curien and Ghelli prove the admissibility of the rule of transitivity [CG92] using a rewiting technique similar to Gentzen's cut-elimination theorem (we will use this technique for transitivity elimination in chapter 9 section 9.2.2), whereas here a simple induction on the proofs suffices. In the same way, we obtain a simpler proof of *coherence* of the semantic framework for bounded quantification proposed by Breazu-Tannen, Coquand, Gunter, and Scedrov [BTCGS91], which we sketch in Section 8.5; indeed, the existence of meets and joins

in our subtype relation permits us to prove the coherence for a somewhat stronger system including variant types.

One of the principal benefits of the simpler formulation of bounded quantification is the ease with which its basic properties may be extended to richer systems. We illustrate this observation in Section 8.6, which shows that a simple kind of recursive types form a conservative extension of $F_{\leq}^{\top}$ (another property that fails in $F_{\leq}$ [Ghe93b]).

Section 8.8 offers some concluding remarks and directions for further research.

8.2 Syntax

We begin by defining the syntax of $F_{\leq}^{\top}$. We work modulo α-conversion for type variables, with the convention that bound variables are silently α-converted as necessary so that type constraint systems and types appearing in instances of the rules are well formed. (Equivalently, we follow deBruijn [dB72] in regarding the connection between occurrences of variables and their binders as part of the syntax of the calculus, considering "poorly scoped" terms as not even parseable). It is easy to check that, for example, the rules below preserve the well-formedness of judgments, i.e. the judgments in the premises are well-formed if and only if the one in the conclusion is.

Type constraint systems are different from those of definition 5.1.1, since now they are used to store the bounds of the free type variables (not the type hierarchy on the atomic types). Thus

Definition 8.2.1 $\emptyset$ is a tcs with $dom(\emptyset){=}\emptyset$. If C is a tcs, $X \notin dom(C)$, and for every $Y \in FV(T)$, $Y \in dom(C)$, then $C \cup \{X{\leq}T\}$ is a tcs and $dom(C \cup \{X{\leq}T\}) = dom(C) \cup \{X\}$. $\Box$

where $FV(T)$ denotes the set of free type variables of the type T. We write $C(X)$ for the upper bound assigned to X by C. (In every situation where we use this notation, it will be clear that C includes a binding for X). The types of $F_{\leq}^{\top}$ are defined by the following abstract grammar:

$$T ::= X \mid \mathsf{Top} \mid T \to T \mid \forall(X{\leq}T)T$$

The syntax of $F_{\leq}$ terms has been described in section 7.1.2 and it is summarized with the typing rules in Appendix E. Since the change in our definition of the subtype relation does not affect the treatment of terms, we will focus only on types and subtyping.

We use $\equiv$ to denote syntactic identity.

There are two kinds of judgments: the subtyping relation $(C \vdash T{\leq}T')$ and the typing relation $(C \vdash a{:}T)$. A type T is *well-formed* in the tcs C if $FV(T) \subseteq dom(C)$. A judgment $C \vdash \Delta$ is said to be *well-formed* iff, every type T appearing in Δ is well-formed in C. In the rest of the chapter we consider only well-formed judgments.

The $F_{\leq}^{\top}$ subtype relation is the least three-place relation closed under the following rules:

(refl) $$C \vdash T{\leq}T$$

(trans) $$\frac{C \vdash T_1 \leq T_2 \qquad C \vdash T_2 \leq T_3}{C \vdash T_1 \leq T_3}$$

(taut) $$C \vdash X \leq C(X)$$

(Top) $$C \vdash T \leq \mathsf{Top}$$

($\rightarrow$) $$\frac{C \vdash T_1 {\leq} S_1 \qquad C \vdash S_2 {\leq} T_2}{C \vdash S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2}$$

($\forall$) $$\frac{C \vdash T_1 {\leq} S_1 \qquad C \cup \{X {\leq} \mathsf{Top}\} \vdash S_2 {\leq} T_2}{C \vdash \forall(X {\leq} S_1) S_2 \leq \forall(X {\leq} T_1) T_2} \qquad X \notin dom(C)$$

(When no ambiguity can arise, we denote the rule ($\forall$-new) simply by ($\forall$)).

8.3 Expressiveness

There are some values of S and T for which $C \vdash S \leq T$ is provable in $F_{\leq}$ but not in $F_{\leq}^{\top}$. But the examples are designed specifically to illustrate pathologies in $F_{\leq}$. Are there any *useful* examples that lie between $F_{\leq}$ and $F_{\leq}^{\top}$? We believe not.

For example, one of the areas where bounded quantification has been most intensively applied is in the study of static type systems for object-oriented languages. One recent study by Pierce and Turner [PT93] presents a statically typed model of the core of Smalltalk using an extension of $F_{\leq}$ with the higher-order polymorphism of Girard's System F^{ω} [Gir72]. All of the terms in this model have exactly the same types in $F_{\leq}^{\top}$ as in $F_{\leq}$.

Luca Cardelli [personal communication, 1993] confirms that all the programs that have been written for his implementation of $F_{\leq}$ [Car93] also typecheck in $F_{\leq}^{\top}$; the only difference in the behaviour of the two systems has been detected with Ghelli's examples of nontermination of the standard procedure [Ghe93a]: $F_{\leq}$ loops for ever, while $F_{\leq}^{\top}$ stops with the right answer.

Another fairly complex application of bounded quantification is Cardelli's translation of a calculus of *extensible record* operations into pure $F_{\leq}$ [Car92]. Again, since each of the *row variables* in the high-level calculus is translated into a row of type variables bounded by Top, $F_{\leq}^{\top}$ should work just as well.

The difference in expressive power between $F_{\leq}$ and $F_{\leq}^{\top}$ can be understood by observing the behavior of the "standard" algorithms for checking the subtype relation. (For $F_{\leq}$, the standard algorithm [CG92] is a semi-decision procedure; the algorithm for $F_{\leq}^{\top}$ presented in Section 8.4.1 below is a decision procedure). The two algorithms are identical at all points, except for the cases for comparing two quantified types, which are just the rules ($\forall$-orig) and ($\forall$-new), respectively. In both cases, the left-hand premise is the same; the only difference lies in the recursive call corresponding to the right-hand premise, which extends the environment with a nontrivial bound for the new variable in the case of $F_{\leq}$ and, for $F_{\leq}^{\top}$, with a trivial bound. If the $F_{\leq}$ algorithm later encounters this variable on the left-hand side of the $\leq$, it has

two choices: either it can check that the expression on the right of the $\leq$ is the same variable and immediately succeed, or it can *promote* the variable on the left to its upper bound from the tcs and try to show that this bound is less than the right hand side. The $F_{\leq}^{\top}$ algorithm has only one choice: if the expression on the right of the $\leq$ is not the same variable (or Top), it must fail.

$F_{\leq}^{\top}$ types strictly fewer terms than $F_{\leq}$. On the other hand, $F_{\leq}^{\top}$ and Cardelli and Wegner's original Fun are not strictly comparable. The assertion

$$\vdash \forall(X{\leq}S)X \leq \forall(X{\leq}S)S$$

is true in Fun and false in $F_{\leq}^{\top}$, while

$$\vdash \forall(X{\leq}\mathsf{Top})X \leq \forall(X{\leq}S)X$$

is true in $F_{\leq}^{\top}$ and false in Fun. But in practical situations, this difference does not appear very important. It is principally the more natural semantic intuition of [∀-new] that leads us to prefer it over [∀-Fun]. There is one intriguing difference, however, which may turn out to have practical consequences. Consider the standard enconding of existential quantifier:

$$\exists(X{\leq}S)T = \forall Y.(\forall(X{\leq}S)\; T \rightarrow Y) \rightarrow Y$$

This encoding in $F_{\leq}^{\top}$ gives rise to the following derived subtyping rule for existential types:

$$(\exists\text{-new}) \qquad \frac{C \vdash S_1{\leq}T_1 \qquad C \cup \{X{\leq}\mathsf{Top}\} \vdash S_2{\leq}T_2}{C \vdash \exists(X{\leq}S_1)S_2{\leq}\exists(X{\leq}T_1)T_2}$$

In Fun, on the other hand, this encoding leads to a derived rule in which S_1 and T_1 are required to be the same.

$$(\exists\text{-Fun}) \qquad \frac{C \cup \{X{\leq}U\} \vdash S_2{\leq}T_2}{C \vdash \exists(X{\leq}U)S_2{\leq}\exists(X{\leq}U)T_2}$$

It is not too difficult to imagine an application where the former rule would be preferable: intuitively, it corresponds to the observation that one package implementing an abstract data type may be more refined than another either when the first provides more (or more refined) operations than the second, or when the first publically reveals more of the structure of its hidden witness type. This may also have an impact on the use of $F_{\leq}$'s polymorphism as the basis for a language of modules, where the types exported by a module are modeled as existentially quantified type variables: different bounds on the variables correspond to different *views* of the same module.

8.4 Basic Properties

In this section we explore the fundamental properties of $F_{\leq}^{\top}$. We show the decidability of the subtyping relation, the existence of a least upper bound (lub) for every finite set of types and a greatest lower bound (glb) for every finite and downward-bounded set of types.

8.4.1 Subtyping algorithm

It is easy to adapt Curien and Ghelli's algorithm [CG92] to $F_{\leq}^{\top}$. Only the rule (Alg∀) is different: here it is identical to (∀-new), whereas in the algorithm for $F_{\leq}$ it coincides with (∀-orig). We use $\vdash_{\mathcal{A}}$ to denote the algorithmic system.

(AlgRefl) $$C \vdash_{\mathcal{A}} X \leq X$$

(AlgTrans) $$\frac{C \vdash_{\mathcal{A}} C(X) \leq T}{C \vdash_{\mathcal{A}} X \leq T}$$

(AlgTop) $$C \vdash_{\mathcal{A}} T \leq \mathsf{Top}$$

(Alg→) $$\frac{C \vdash_{\mathcal{A}} T_1 \leq S_1 \qquad C \vdash_{\mathcal{A}} S_2 \leq T_2}{C \vdash_{\mathcal{A}} S_1 \to S_2 \leq T_1 \to T_2}$$

(Alg∀) $$\frac{C \vdash_{\mathcal{A}} T_1 \leq S_1 \qquad C \cup \{X \leq \mathsf{Top}\} \vdash_{\mathcal{A}} S_2 \leq T_2}{C \vdash_{\mathcal{A}} \forall(X \leq S_1)S_2 \leq \forall(X \leq T_1)T_2} \qquad X \notin dom(C)$$

Note that the algorithm defined by these rules is deterministic, since the form of the input unequivocally determines the rule that must be used — not true in the original system because of (refl) and (trans) — and all the parameters of any recursive calls — not true in $\vdash$ because of (trans).

Soundness and completeness

We now show that every judgment provable in the algorithmic system, $\vdash_{\mathcal{A}}$, is also provable in the original system, $\vdash$ (soundness of the algorithm) and that every relation provable by $\vdash$ is provable also by $\vdash_{\mathcal{A}}$ (completeness of the algorithm). Finally, we show that the algorithm actually constitutes a decision procedure — that it halts on every input.

Lemma 8.4.1 *Let $C \vdash_{\mathcal{A}} S \leq T$. Then:*

1. *If $S = \mathsf{Top}$, then $T = \mathsf{Top}$.*
2. *If S is a variable X, then either $T = X$ or $T = \mathsf{Top}$, or else $C \vdash_{\mathcal{A}} C(X) \leq T$.*
3. *If $S = S_1 \to S_2$, then either $T = \mathsf{Top}$ or else $T = T_1 \to T_2$ with $C \vdash_{\mathcal{A}} T_1 \leq S_1$ and $C \vdash_{\mathcal{A}} S_2 \leq T_2$.*
4. *If $S = \forall(X \leq S_1)S_2$, then either $T = \mathsf{Top}$ or else $T = \forall(X \leq T_1)T_2$ with $C \vdash_{\mathcal{A}} T_1 \leq S_1$ and $C \cup \{X \leq \mathsf{Top}\} \vdash_{\mathcal{A}} S_2 \leq T_2$.*

Proof. By inspection. □

To prove the completeness of the algorithm we need two simple lemmas, which show that the relation defined by $\vdash_{\mathcal{A}}$ is reflexive (lemma 8.4.2) and transitive (lemma 8.4.3); in other words the two rules that belong to the definition of $\vdash$ and not $\vdash_{\mathcal{A}}$ are derived rules in $\vdash_{\mathcal{A}}$.

Lemma 8.4.2 *For each tcs C and type T (well-formed in C), the judgment $C \vdash_{\mathcal{A}} T \leq T$ is provable.*

Proof. By induction on the structure of T.

$T \equiv X$ immediate from (AlgRefl).

$T \equiv \mathsf{Top}$ immediate from (AlgTop).

$T \equiv S_1 \to S_2$ then by the induction hypothesis $C \vdash_{\mathcal{A}} S_1 \leq S_1$ and $C \vdash_{\mathcal{A}} S_2 \leq S_2$. The result follows by (Alg$\to$).

$T \equiv \forall(X \leq S_1)S_2$ then by the induction hypothesis $C \vdash_{\mathcal{A}} S_1 \leq S_1$ and $C \cup \{X \leq \mathsf{Top}\} \vdash_{\mathcal{A}} S_2 \leq S_2$. The result follows by (Alg$\forall$).

□

Lemma 8.4.3 *For each tcs C and well-formed types T_1, T_2, and T_3, if $C \vdash_{\mathcal{A}} T_1 \leq T_2$ and $C \vdash_{\mathcal{A}} T_2 \leq T_3$, then $C \vdash_{\mathcal{A}} T_1 \leq T_3$.*

Proof. If $T_3 \equiv \mathsf{Top}$, then the result follows from (AlgTop). Thus we can restrict our attention to those cases in which $T_3 \not\equiv \mathsf{Top}$; note that by lemma 8.4.1 T_2 must also be different from Top.

Let d_i denote the depth of the proof of $C \vdash_{\mathcal{A}} T_i \leq T_{i+1}$ for $i = 1, 2$. We proceed by induction on $(d_1 + d_2)$, with a case analysis on the last rule of the proof of $C \vdash_{\mathcal{A}} T_1 \leq T_2$

(AlgRefl) $C \vdash_{\mathcal{A}} T_1 \equiv T_2 \leq T_3$ is satisfied by hypothesis. (Note that this case forms the base of the induction where $(d_1 + d_2) = 2$, i.e. when $T_1 \equiv T_2 \equiv T_3 \equiv X$).

(AlgTrans) $T_1 \equiv X$. By applying the induction hypothesis to the second premise, we obtain that $C \vdash_{\mathcal{A}} C(X) \leq T_3$; the result follows by (AlgTrans).

(AlgTop) Cannot occur, since we assumed $T_2 \not\equiv \mathsf{Top}$.

(Alg$\to$) $T_1 \equiv S_1 \to S_1'$ and $T_2 \equiv S_2 \to S_2'$, and, since $T_3 \not\equiv \mathsf{Top}$, by lemma 8.4.1 $T_3 \equiv S_3 \to S_3'$. Furthermore because of the (Alg$\to$) rule $C \vdash_{\mathcal{A}} S_3 \leq S_2$ and $C \vdash_{\mathcal{A}} S_2 \leq S_1$; then by the induction hypothesis, $C \vdash_{\mathcal{A}} S_3 \leq S_1$ and similarly for S_1', S_2', S_3' and $C \vdash_{\mathcal{A}} S_1' \leq S_3'$; by (Alg$\to$) we obtain the result.

(Alg$\forall$) Similarly, we have by assumption that $T_1 \equiv \forall(X \leq S_1)S_1'$ and $T_2 \equiv \forall(X \leq S_2)S_2'$, and, since $T_3 \not\equiv \mathsf{Top}$ by lemma 8.4.1 $T_3 \equiv \forall(X \leq S_3)S_3'$. By the induction hypothesis, $C \vdash_{\mathcal{A}} S_3 \leq S_1$ and $C \cup \{X \leq \mathsf{Top}\} \vdash_{\mathcal{A}} S_1' \leq S_3'$; the result follows by (Alg$\forall$).

Note that this case causes significant difficulty in the proofs of Curien and Ghelli [CG92] (see also section 9.2.2) and Breazu-Tannen et al. [BTCGS91].

□

Soundness is also proved by induction on the derivations.

Theorem 8.4.4 (Soundness and completeness) *For each tcs C and well-formed types S_1 and S_2,*

$$C \vdash_{\mathcal{A}} S_1 \leq S_2 \iff C \vdash S_1 \leq S_2$$

Proof. We start by proving soundness ($\Rightarrow$) by induction on the depth of the proof of $C \vdash_{\mathcal{A}} S_1 \leq S_2$. At each stage of the induction, we proceed by cases on the final rule used in the proof. The cases for (AlgRefl) and (AlgTop) are immediate; (Alg$\rightarrow$) and (Alg$\forall$) follow by straightforward use of the induction hypothesis. For (AlgTrans), we have $S_1 \equiv X$. By the induction hypothesis, $C \vdash C(X) \leq S_2$; by (taut) $C \vdash X \leq C(X)$ thus by (trans) we obtain the result.

Completeness ($\Leftarrow$) follows from lemmas 8.4.2 and 8.4.3. □

This establishes that the two systems $\vdash$ and $\vdash_{\mathcal{A}}$ are equivalent and we can eliminate the $\mathcal{A}$ indexing the turnstyle symbol. In particular this implies that the lemma 8.4.1 is also valid for $\vdash$.

Note that in a proof of subtyping we are now allowed to mix rules of the two systems since, by induction, for every subproof there exists one equivalent proof formed only by rules of a same system. Thus in the proofs of the theorems that follow we will freely use rules from both systems. In particular we will refer to algorithmic or full transitivity, according to our needs.

Termination

We first define a weight on a type T with respect to a tcs C (with T well-formed in C):

$$\begin{array}{rcl} \mathcal{T}(\mathsf{Top})_C & = & 1 \\ \mathcal{T}(X)_C & = & \mathcal{T}(C(X))_C + 1 \\ \mathcal{T}(S_1 \rightarrow S_2)_C & = & \mathcal{T}(S_1)_C + \mathcal{T}(S_2)_C \\ \mathcal{T}(\forall(X \leq S_1)S_2)_C & = & \mathcal{T}(S_1)_C + \mathcal{T}(S_2)_{C \cup \{X \leq S_1\}} \end{array}$$

In the final case, the variable X is added to the tcs with the bound S_1 instead of Top so that it will have the same bound on the left as on the right. This will be used in the proof of lemma 8.4.5 to define an ordering that depends on the bounds of a variable. Katiyar and Shankar [KS92] give a similar proof of termination for their variant of $F_{\leq}$.

Lemma 8.4.5 *For each type T well-formed in a tcs C, the weight $\mathcal{T}(T)_C$ is finite and positive.*

Proof. First, it is obvious that the weight $\mathcal{T}(T)_C$ is always positive. Now, we give a well-founded rank for $\mathcal{T}(T)_C$ (i.e. we define a weight for the definition of the weight) and we show that it decreases at each stage in the definition of $\mathcal{T}$. To define the rank of $\mathcal{T}(T)_C$ consider all the variables that appear in T and C (no matter whether they appear free or bounded, only in a quantifier or in a bound). Since T is well-formed in C, every variable is associated to a unique bound (either in C or in T); furthermore it is also possible to totally order these variables in a way that if X_i is used in the bound of X_j then X_i precedes X_j (since T is well-formed in C, loops are not possible). If there is more than one order satisfying this

condition then choose one arbitrarily. Define the *depth* of each variable as the number of variables that precede it in this order. Then the rank of $\mathcal{T}(T)_C$ is the lexicographical size of the pair (D, L), where D is the maximum depth of any of the variables that appear in T, and L is the textual length of T. This rank is well-founded (the least element is $(0, 1)$). It easy to see that for the subproblem on the right-hand side of $\mathcal{T}(S_1 \to S_2)_C$ and $\mathcal{T}(\forall(X{\leq}S_1)S_2)_C$, the component D either is the same or it decreases, while the L component always strictly decreases; for the case $\mathcal{T}(X)_C$, the component D strictly decreases. □

The weight for types is extended to a weight for type judgments in the obvious way:

$$\mathcal{J}(C \vdash S_1 \leq S_2) = \mathcal{T}(S_1)_C + \mathcal{T}(S_2)_C.$$

Now we can show the termination of the algorithm.

Lemma 8.4.6 *For all types S and T well-formed in C, $\mathcal{T}(T)_{C\cup\{X\leq\mathsf{Top}\}} \leq \mathcal{T}(T)_{C\cup\{X\leq S\}}$*

Proof. By the previous lemma we are now allowed to prove this lemma by induction on $\mathcal{T}(T)_{C\cup\{X\leq\mathsf{Top}\}} + \mathcal{T}(T)_{C\cup\{X\leq S\}}$. The proof is straightforward. □

Theorem 8.4.7 *At every step of the subtyping algorithm, the weight of each of the premises is strictly smaller than the weight of the conclusion.*

Proof. The verification is easy in most cases. The only non-trivial case is (Alg∀), which is proved by the following inequalities:

$$\begin{array}{rcll}
\mathcal{J}(C \cup \{X{\leq}\mathsf{Top}\} \vdash S_2{\leq}T_2) & = & \mathcal{T}(S_2)_{C\cup\{X\leq\mathsf{Top}\}} + \mathcal{T}(T_2)_{C\cup\{X\leq\mathsf{Top}\}} & \\
& \leq & \mathcal{T}(S_2)_{C\cup\{X\leq S_1\}} + \mathcal{T}(T_2)_{C\cup\{X\leq T_1\}} & \text{by lemma 8.4.6} \\
& < & \mathcal{T}(S_1)_C + \mathcal{T}(T_1)_C + \mathcal{T}(S_2)_{C\cup\{X\leq S_1\}} + \mathcal{T}(T_2)_{C\cup\{X\leq T_1\}} & \\
& = & \mathcal{T}(\forall(X{\leq}S_1)S_2)_C + \mathcal{T}(\forall(X{\leq}T_1)T_2)_C & \\
& = & \mathcal{J}(C \vdash \forall(X{\leq}S_1)S_2 \leq \forall(X{\leq}T_1)T_2) &
\end{array}$$

□

Corollary 8.4.8 *The algorithm terminates.*

8.4.2 Meets and joins

Decidability of subtyping is clearly a desirable property, but in practice the undecidability of $F_{\leq}$ has been much less problematic than the nonexistence of least upper bounds and greatest lower bounds in the subtype relation. Ghelli [Ghe90] observed that, in $F_{\leq}$, the types

$$S \equiv \forall(Z{\leq}X \to Y)\ X \to Y \qquad\qquad T \equiv \forall(Z{\leq}X' \to Y')\ X' \to Y'$$

in the tcs

$$C \equiv \{X{\leq}\mathsf{Top},\ Y{\leq}\mathsf{Top},\ X'{\leq}X,\ Y'{\leq}Y\}$$

have two lower bounds (namely $U \equiv \forall(Z{\leq}X' \to Y)\ X \to Y'$ and $V \equiv \forall(Z{\leq}X' \to Y)\ Z)$ but that there is no greatest lower bound of S and T, since U and V have no common supertype

that is also a subtype of S and T. From this, it is easy to show that $F_{\leq}$ also lacks lubs. For example, $S \rightarrow \mathsf{Top}$ and $T \rightarrow \mathsf{Top}$ have no lub.

Here, by contrast, a simple induction on the weight defined in the previous section shows that every pair of types in $F_{\leq}^{\top}$ *does* possess a least upper bound, and that a pair of types with any lower bound at all possesses a greatest lower bound.

Theorem 8.4.9 *Let S and T be well-formed types in a tcs C.*

(a) *There is some U such that $C \vdash S,T \leq U$ and such that $C \vdash U \leq U'$ whenever $C \vdash S,T \leq U'$.*
(b) *If there is any V such that $C \vdash V \leq S,T$, then there is some L such that $C \vdash L \leq S,T$ and such that $C \vdash L' \leq L$ whenever $C \vdash L' \leq S,T$.*

Proof. In this proof we consider derivations in $\vdash_{\mathcal{A}}$.

We prove both cases simultaneously by induction on $\mathcal{T}(S)_C + \mathcal{T}(T)_C$. First, note that the theorem is immediate when one of the two types is Top: the lub is Top and the glb is the other type; this also provides the base case of the induction, i.e. when $(\mathcal{T}(S)_C + \mathcal{T}(T)_C = 2)$. Also trivial is the case when $S \equiv T$. Thus, we may assume that T and S are two distinct types both different from Top, and consider the remaining cases:

1. Both S and T are variables. For part (a) we have two cases: either the two variables are related, and in that case the greater of them is the lub; or the two variables are unrelated and we can apply the induction hypothesis to $C(S)$ and $C(T)$, obtaining that there exists a type U which is their lub. It is easy then to show that by the rule (AlgTrans) their lub is also the lub of S and T: indeed every upper bound of $C(S)$ is an upper bound of S too. And every *proper* upper bound of S is an upper bound for $C(S)$ (the same for T). Thus the set of proper upper bounds of S coincides with the set of upper bounds for $C(S)$; the same holds for T, too. Since the two variables are unrelated the set of their common upper bounds coincide with the set of their common proper upper bounds which is equal to the set of the common upper bounds of $C(S)$ and $C(T)$.

   For part (b), simply note that by lemma 8.4.1 if S and T have a common lower bound Z then it is a variable. If a variable is provably smaller than another variable then the proof consists only of rules (AlgTrans) (this can be obtained by a trivial induction on the depth of the derivation); consider the proofs of $C \vdash Z \leq S,T$. Since they both start from Z and always use the same rule (AlgTrans) (which is deterministic) then the shorter proof must appear as a subproof of the longest one; thus using the full transitivity either $C \vdash S \leq T$ or $C \vdash T \leq S$ hold. Thus the lesser of them is the sought-after glb.

2. Both S and T are arrow types. Thus let $S \equiv S_1 \rightarrow S_2$ and $T \equiv T_1 \rightarrow T_2$.

   case (a) If S_1 and T_1 have a common lower bound then by induction hypothesis there exists $S_1 \sqcap T_1$ glb of S_1 and T_1 w.r.t. C. Then it is trivial to show that the lub of S,T is $S_1 \sqcap T_1 \rightarrow S_2 \sqcup T_2$ where $S_2 \sqcup T_2$ is the lub of S_2, T_2 w.r.t. C (which exists by induction hypothesis). If S_1, T_1 have no common lower bound then it is clear that the glb of T,S is Top.

case (b) Suppose that there exists V satisfying the hypothesis of the lemma. Then by lemma 8.4.1 $V \equiv V_1 \rightarrow V_2$ or V is a variable. If $V \equiv V_1 \rightarrow V_2$ then $C \vdash V_2 \leq S_2, T_2$; thus by induction hypothesis there exists $S_2 \sqcap T_2$ glb of S_2, T_2 w.r.t. C. As in the case (a) it is then easy to show that $S_1 \sqcup T_1 \rightarrow S_2 \sqcap T_2$ is the glb of T, S where $S_1 \sqcup T_1$ is the lub of S_2, T_2 w.r.t. C which exists by induction hypothesis.

If V is a variable then consider the following definition of $\mathcal{B}(T)_C$:
1. $\mathcal{B}(T)_C = T$ if T is not a type variable.
2. $\mathcal{B}(T)_C = \mathcal{B}(C(T))_C$ else.

Of course $C \vdash V \leq \mathcal{B}(V)_C$ and by induction on the depth of this proof it is possible to prove that $C \vdash \mathcal{B}(V)_C \leq S, T$. By lemma 8.4.1 $\mathcal{B}(V)_C \equiv V_1 \rightarrow V_2$. Thus it suffice to apply the induction hypothesis as we have done above to obtain the result.

3. Both S and T are universally quantified types. This case is solved as the case above. Though, since this is the case in which the induction hypothesis does not work with $F_{\leq}$, we show it more in detail; let us examine the critical passage for example for the part (b) of the statement when V is not a variable: let $S \equiv \forall (X \leq S_1) S_2$, $T \equiv \forall (X \leq T_1) T_2$ and $V \equiv \forall (X \leq V_1) V_2$. Then by the rule ($\forall$) we have that

$$C \vdash S_1, T_1 \leq V_1$$

$$C \cup \{X \leq \mathsf{Top}\} \vdash V_2 \leq S_2, T_2$$

By the definition of $\mathcal{T}$ we can apply the induction hypothesis on both the judgments obtaining that exists L_1, L_2 such that:

$$C \vdash S_1, T_1 \leq L_1$$

$$C \cup \{X \leq \mathsf{Top}\} \vdash L_2 \leq S_2, T_2$$

and that $C \vdash W_1 \leq L_1$,$C \cup \{X \leq \mathsf{Top}\} \vdash W_2 \leq L_2$ whenever

$$C \vdash S_1, T_1 \leq W_1$$

$$C \cup \{X \leq \mathsf{Top}\} \vdash W_2 \leq S_2, T_2$$

It is then clear that $\forall (X \leq L_1) L_2$ is the sought glb.

4. One of the two types, say S is a type variable and the other not. Then for part (a) you can apply the induction hypothesis to $C(S)$ and T. For part (b) you have that $C \vdash S \leq T$ must hold (same observation as in case 1).

□

8.5 Semantics

Up to now we have dealt only with types and subtyping. In this section we introduce terms and three new type constructors. We provide a sound semantics for the obtained language and a coherence result for the corresponding proof system. This section is largely based on [BTCGS91], where the reader can find a more detailed presentation of the technique we use here. In the cited paper a semantics for $F_{\leq}$ plus record and recursive types is given by translating the system into an extension of System F for which sound semantic interpretations are already defined. The soundness of the method is given by a coherence result. The same

proof fails if variant types are also added to the system (this is essentially due to the absence of meets and joins in $F_{\leq}$). Our system behaves better since the technique of [BTCGS91] works for our system even if we extend it by variant types, as we show in this section.

Therefore we enrich our set of types by record types ($\langle\!\langle \ell_1{:}T_1, \ldots, \ell_p{:}T_p \rangle\!\rangle$), variant types ($[\ell_1{:}T_1, ..., \ell_p{:}T_p]$) and recursive types ($\mu X.T$). We extend the subtyping relation by adding just the following rules.

(recd) $$\frac{C \vdash S_1 \leq T_1 \ldots C \vdash S_p \leq T_p}{C \vdash \langle\!\langle \ell_1{:}S_1, ..., \ell_p{:}S_p, ..., \ell_q{:}S_q \rangle\!\rangle \leq \langle\!\langle \ell_1{:}T_1, \ldots, \ell_p{:}T_p \rangle\!\rangle}$$

(vart) $$\frac{C \vdash S_1 \leq T_1 \ldots C \vdash S_p \leq T_p}{C \vdash [\ell_1{:}S_1, \ldots, \ell_p{:}S_p] \leq [\ell_1{:}T_1, \ldots, \ell_p{:}T_p, \ldots, \ell_q{:}T_q]}$$

Note that we do a very naive treatment of recursive types since only reflexivity can be used to subtype them and no bound is imposed on the variable of the recursion (for a wider treatment of the combination of recursion and subtyping see [Ama91]). For that reason all the results of the previous section can be easily extended to this system.

The raw terms of the language are described by the following productions:

$$\begin{array}{rcl} a & ::= & x^T \mid (\lambda x^T.a) \mid a(a) \\ & \mid & \Lambda X{\leq}T.a \mid a(T) \mid \mathsf{top} \\ & \mid & \langle \ell_1 = a_1, \ldots, \ell_n = a_n \rangle \mid a.\ell \\ & \mid & [\ell_1{:}T_1, \ldots, \ell_i = a, \ldots, \ell_n{:}T_n] \\ & \mid & \mathbf{case}\ a\ \mathbf{of}\ \ell_1 \Rightarrow a_1, \ldots, \ell_n \Rightarrow a_n \\ & \mid & \mathbf{intro}_{\mu X.T}(a) \mid \mathbf{elim}(a) \end{array}$$

The typing rules for this language are exactly the same as those of $F_{\leq}$ (see Appendix E). plus the rules for the new terms:

[$\langle\!\langle\rangle\!\rangle$INTRO] $$\frac{C \vdash a_1{:}T_1 \ldots C \vdash a_p{:}T_p}{C \vdash \langle \ell_1 = a_1, \ldots, \ell_p = a_p \rangle{:}\,\langle\!\langle \ell_1{:}T_1, \ldots, \ell_p{:}T_p \rangle\!\rangle}$$

[$\langle\!\langle\rangle\!\rangle$ELIM] $$\frac{C \vdash a{:}\,\langle\!\langle \ell_1{:}T_1, \ldots, \ell_p{:}T_p \rangle\!\rangle}{C \vdash a.\ell_i{:}T_i}$$

[[]INTRO] $$\frac{C \vdash a{:}T_i}{C \vdash [\ell_1{:}T_1, \ldots, \ell_i = a_i, \ldots \ell_p{:}T_p]{:}\,[\ell_1{:}T_1, \ldots, \ell_i{:}T_i, \ldots, \ell_p{:}T_p]}$$

[[]ELIM] $$\frac{C \vdash b{:}\,[\ell_1{:}T_1, \ldots, \ell_p{:}T_p] \qquad C \vdash a_1{:}T_1 \to T \ \ \ldots \ \ C \vdash a_p{:}T_p \to T}{C \vdash \mathbf{case}\ b\ \mathbf{of}\ \ell_1 \Rightarrow a_1, \ldots, \ell_p \Rightarrow a_p{:}T}$$

[μINTRO] $$\frac{C \vdash a{:}\,T[X := \mu X.T]}{C \vdash \mathbf{intro}_{\mu X.T}(a){:}\,\mu X.T}$$

[μELIM] $$\frac{C \vdash a{:}\,\mu X.T}{C \vdash \mathbf{elim}(a){:}\,[X := \mu X.T]}$$

To give a semantics to this language we consider a language **TARGET** for which many semantic interpretations already exist. We translate the derivations of judgments of our language into derivations of judgments of **TARGET**. Then we give the semantics via this translation, i.e. the semantics for a typing judgment in our source language is the semantics

of its translation. But this leads to a problem of coherence since in our source language for one typing judgment there may correspond different derivations and thus different translations. Thus we end this section by a theorem proving that the typing judgments obtained as translations of different derivations for a same judgment are provably equal in **TARGET** and thus they have the same semantic interpretation. In this way we obtain the coherence of our semantic interpretation.

8.5.1 The language TARGET

The target language, whose complete definition can be found in appendix B of [BTCGS91], is an extension of Girard's System F [Gir72] by recursive types, variants and records, and coercion spaces. Coercion spaces will be used to interpret subtyping judgments (for the reader acquaitened with the models of System F, coercion spaces are interpreted as strict maps in the model of dI-domains and as linear maps in the coherent spaces).

Types

$$
\begin{array}{rcll}
C & ::= & T \multimap T & \text{coercion spaces} \\[1em]
T & ::= & X & \\
& | & T \to T & \\
& | & C \to C & \\
& | & C \to T & \\
& | & \forall X.T & \\
& | & \langle\!\langle \ell_1 : T_1, \ldots, \ell_n : T_n \rangle\!\rangle & \\
& | & [\ell_1 : T_1, \ldots, \ell_n : T_n] & \\
& | & \mu X.T &
\end{array}
$$

The key idea of the translation is to consider the rule of subsumption as the use of an implicit coercion, and the translation makes this coercion explicit. The translation of a term $\Lambda X{\leq}T.a$ takes as parameters both a type X and a coercion function from X to T; this coercion will be explicitly used in the translation of a to replace the implicit use of subsumption in a. More precisely we give a translation $_^*$ that translates the type $\forall(X{\leq}S)T$ into $\forall X.(X \multimap S^*) \to T^*$ and translates a term $a{:}T$ into a term $a^*{:}T^*$.

Raw Terms

$$
\begin{array}{rcl}
a & ::= & x \mid (\lambda x^T.a) \mid a(a) \\
& | & \Lambda X.a \mid a(T) \\
& | & \langle \ell_1 = a_1, \ldots, \ell_n = a_n \rangle \mid a.\ell \\
& | & [\ell_1 : T_1, \ldots, \ell_i = a, \ldots, \ell_n : T_n] \\
& | & \textbf{case } a \textbf{ of } \ell_1 \Rightarrow a_1, \ldots, \ell_n \Rightarrow a_n \\
& | & \textbf{intro}_{\mu X.T}(a) \mid \textbf{elim}(a)
\end{array}
$$

Just note that the variables are no longer indexed by their types.

Coercion combinators

A subtyping judgment $S \leq T$ will be translated as a coercion function in $(S^* \multimapdotboth T^*)$. We follow the technique of [BTCGS91] and we introduce a set of constants $\iota_{S,T}$ that transform a coercion into a function (in order to apply it) and another family of constants (coercion combinators) that we use to translate subtyping judgments; we use 1 to denote the empty record type $\langle\!\langle\,\rangle\!\rangle$:

$$\iota_{S,T}: (S \multimapdotboth T) \to (S \to T)$$

$$\begin{array}{l}
\mathsf{top}[T]: T \multimapdotboth 1 \\
\mathsf{refl}[T]: T \multimapdotboth T \\
\mathsf{trans}[S,T,U]: (S \multimapdotboth T) \to (T \multimapdotboth U) \to (S \multimapdotboth U) \\
\mathsf{arrow}[S,T,U,V]: (S \multimapdotboth T) \to (U \multimapdotboth V) \to ((T \to U) \multimapdotboth (S \to V)) \\
\mathsf{forall}[S,T,U,V]: (S \multimapdotboth T) \to (\forall X((X \multimapdotboth 1) \to (U \multimapdotboth V)) \\
\qquad\qquad \to \forall X((X \multimapdotboth T) \to U) \multimapdotboth \forall X((X \multimapdotboth S) \to V) \\
\mathsf{recd}[S_1,\ldots,S_q,T_1,\ldots T_p]: (S_1 \multimapdotboth T_1) \to \ldots \to (S_p \multimapdotboth T_p) \\
\qquad\qquad \to (\langle\!\langle \ell_1: S_1,\ldots,\ell_p: S_p,\ldots,\ell_q: S_q \rangle\!\rangle \multimapdotboth \langle\!\langle \ell_1: T_1,\ldots,\ell_p: T_p \rangle\!\rangle) \\
\mathsf{vart}[S_1,\ldots,S_p,T_1,\ldots T_q]: (S_1 \multimapdotboth T_1) \to \ldots \to (S_p \multimapdotboth T_p) \\
\qquad\qquad \to ([\ell_1: S_1,\ldots,\ell_p: S_p] \multimapdotboth [\ell_1: T_1,\ldots,\ell_p: T_p,\ldots,\ell_q: T_q])
\end{array}$$

For example $\mathsf{arrow}[S,T,U,V]$ is the coercion combinator that takes two coercions from S to T and from U to V and returns *the* corresponding coercion from $T \to U$ to $S \to V$. The precise behavior of each combinator is univoquely determined by ι, i.e. by its transformation into a function (in the following we omit the types in the combinators when they follow clearly from the context. We use ";" to denote the composition):

$$\begin{array}{lcll}
\iota(\mathsf{top}) & = & \lambda x^T.\langle\,\rangle & \\
\iota(\mathsf{refl}) & = & \lambda x^T.x & \\
\iota(\mathsf{trans}(a)(b)) & = & \iota(a);\iota(b) & \text{where } a: R \multimapdotboth S, b: S \multimapdotboth T \\
\iota(\mathsf{arrow}(a)(b)) & = & \lambda z^{T\to U}.(\iota(a)); z; \iota(b) & \text{where } a: S \multimapdotboth T, b: U \multimapdotboth V \\
\iota(\mathsf{forall}(a)(b)) & = & \lambda z^{\forall X(X \multimapdotboth T)\to U}.\Lambda X.\lambda x^{X \multimapdotboth S}.\iota(b(X)(\mathsf{top}[X]))(z(X)(\mathsf{trans}(x)(a))) & \\
 & & & \text{where } a: S \multimapdotboth T, b: \forall X.(X \multimapdotboth 1) \to (U \multimapdotboth V) \\
\iota(\mathsf{recd}(a_1)\cdots(a_p)) & = & \lambda z^{\langle\!\langle \ell_1:S_1,\ldots,\ell_p:S_p,\ldots,\ell_q:S_q \rangle\!\rangle}.\langle\!\langle \ell_1 = \iota(a_1)(z.\ell_1),\ldots,\ell_p = \iota(a_p)(z.\ell_p) \rangle\!\rangle & \\
 & & & \text{where } a_1: S_1 \multimapdotboth T_1,\ldots,a_p: S_p \multimapdotboth T_p \\
\iota(\mathsf{vart}(a_1)\cdots(a_p)) & = & \lambda z^{[\ell_1:S_1,\ldots,\ell_p:S_p]}.\mathbf{case}\ z\ \mathbf{of}\ \ell_1 \Rightarrow \iota(a_1);\mathbf{inj}_{\ell_1},\ldots,\ell_p = \iota(a_p);\mathbf{inj}_{\ell_p} & \\
 & & & \text{where } a_1: S_1 \multimapdotboth T_1,\ldots,a_p: S_p \multimapdotboth T_p \text{ and} \\
 & & & \mathbf{inj}_{\ell_i} = \lambda x^{T_i}[\ell_1: T_1,\ldots,\ell_i = a,\ldots,\ell_q: T_q]
\end{array}$$

The translation of forall in [BTCGS91] was not linear since the parameter x in the body of the function appeared twice. For $F^{\top}_{\leq}$ instead the translation of forall is a linear function: x appears exactly once in the body of the function. This fits very well with the semantics of the target language in the models based on stable maps, where $\multimapdotboth$ denotes the linear maps.

Equational Theory

As we said the interpretation via ι of the coercions as functions must uniquely determine the corresponding coercion. This is obtained by the following equality rule in **TARGET**:

$$\langle\iota\text{-INJ}\rangle \qquad \frac{\iota(a) = \iota(b)}{a = b}$$

A presentation of the whole equational theory of **TARGET** is given by this rule plus the classical set of rules of F of records, variant and recursive types ($\langle\beta\rangle$, $\langle\beta_2\rangle$, $\langle\eta\rangle$, $\langle\eta_2\rangle$, $\langle\xi\rangle$, $\langle\xi_2\rangle$, $\langle\beta_{recd}\rangle$, $\langle\eta_{recd}\rangle$, $\langle\beta_\mu\rangle$, $\langle\eta_\mu\rangle$), plus a special rule for variant types to handle coercion terms:

$$\langle\text{CRN}\rangle \qquad \iota(P)(\textbf{case}\ a\ \textbf{of}\ \ell_1 \Rightarrow a_1, ..., \ell_n \Rightarrow a_n) = (\textbf{case}\ a\ \textbf{of}\ \ell_1 \Rightarrow a_1;\iota(P), ..., \ell_n \Rightarrow a_n;\iota(P))$$

The precise definition of all the rules can be found in the Appendix B of [BTCGS91]. See this same paper for the motivations of the introduction of the rule ⟨CRN⟩.

8.5.2 Translation

We are now ready to give the formal translation for the derivations of our source language.

We start with the translation of the types and the tcs's:

$$\begin{array}{lcl}
X^* & = & X \\
\mathsf{Top}^* & = & 1 \\
(S \to T)^* & = & S^* \to T^* \\
(\forall(X{\leq}S)T)^* & = & \forall X.((X \multimap S^*) \to T^*) \\
\langle\!\langle \ell_1{:}T_1, \ldots, \ell_n{:}T_n \rangle\!\rangle^* & = & \langle\!\langle \ell_1{:}T_1^*, \ldots, \ell_n{:}T_n^* \rangle\!\rangle \\
([\ell_1{:}T_1, \ldots, \ell_n{:}T_n])^* & = & [\ell_1{:}T_1^*, \ldots, \ell_n{:}T_n^*] \\
(\mu X.T)^* & = & \mu X.T^* \\
 & & \\
\emptyset^* & = & \emptyset \\
(C \cup \{X{\leq}T\})^* & = & C^* \cup X \cup \{x{:}\, X \multimap T^*\}
\end{array}$$

Note that in the target language we also declare in the context the free type variables $(\ldots \cup X \cup \ldots)$.

Then it follows the translation of the subtyping rules:

$$(\text{refl})^* \qquad C^* \vdash \mathsf{refl}{:}\, T^* \multimap T^*$$

$$(\text{trans})^* \qquad \frac{C^* \vdash a : T_1^* \multimap T_2^* \qquad C^* \vdash b : T_2^* \multimap T_3^*}{C^* \vdash \mathsf{trans}(a)(b) : T_1^* \multimap T_3^*}$$

$$(\text{taut})^* \qquad C^* \cup X \cup \{x{:}\, \{X\} \multimap C(X)^*\} \vdash x{:}\, X \multimap C(X)^*$$

$$(\text{Top}) \qquad C \vdash \mathsf{top}[T^*]{:}\, T^* \multimap 1$$

$$(\to)^* \qquad \frac{C^* \vdash a{:}\, T_1^* \multimap S_1^* \qquad C^* \vdash b{:}\, S_2^* \multimap T_2^*}{C^* \vdash \mathsf{arrow}(a)(b){:}\, (S_1^* \to S_2^*) \multimap (T_1^* \to T_2^*)}$$

$$(\forall)^* \quad \frac{C^* \vdash a{:}\, T_1^* \multimap S_1^* \qquad C^* \cup \{X\} \cup \{x{:}\, X \multimap 1\} \vdash b{:}\, S_2^* \multimap T_2^*}{C^* \vdash \mathsf{forall}(a)(\Lambda X.\lambda x^{X \multimap 1}.b){:}\, \forall X((X \multimap S_1^*) \to S_2^*) \multimap \forall X((X \multimap T_1^*) \to T_2^*)}$$

$$(\text{recd})^* \quad \frac{C^* \vdash a_1{:}\, S_1^* \multimap T_1^* \ldots C^* \vdash a_p{:}\, S_p^* \multimap T_p^*}{C^* \vdash \mathsf{recd}(a_1)\cdots(a_p){:}\, \langle\!\langle \ell_1{:}\, S_1^*, \ldots, \ell_p{:}\, S_p^*, \ldots, \ell_q{:}\, S_q^* \rangle\!\rangle \multimap \langle\!\langle \ell_1{:}\, T_1^*, \ldots, \ell_p{:}\, T_p^* \rangle\!\rangle}$$

$$(\text{vart})^* \quad \frac{C^* \vdash a_1{:}\, S_1^* \multimap T_1^* \ldots C^* \vdash p{:}\, S_p^* \multimap T_p^*}{C^* \vdash \mathsf{vart}(a_1)\cdots(a_p){:}\, [\ell_1{:}\, S_1^*, \ldots, \ell_p{:}\, S_p^*] \multimap [\ell_1{:}\, T_1^*, \ldots, \ell_p{:}\, T_p^*, \ldots, \ell_q{:}\, T_q^*]}$$

Most of the typing rules are translated in a trivial way. We write here just three of them to give an example. The complete translation can be found in the appendix F

$$[\forall\text{Intro}]^* \quad \frac{C^* \cup X \cup \{x{:}\, X \multimap T^*\} \vdash a{:}\, T'^*}{C^* \vdash \Lambda X \lambda x^{X \multimap T^*}.a{:}\, \forall X(X \multimap T^*) \to T'^*}$$

$$[\forall\text{Elim}]^* \quad \frac{C^* \vdash a{:}\, \forall X(X \multimap S^*) \to T^* \qquad C^* \vdash b{:}\, S'^* \multimap S^*}{C^* \vdash a(S'^*)(b){:}\, T^*[X := S'^*]}$$

$$[\text{Subsumption}]^* \quad \frac{C^* \vdash a{:}\, T'^* \qquad C^* \vdash b{:}\, T'^* \multimap T^*}{C^* \vdash \iota(b)(a){:}\, T^*}$$

Theorem 8.5.1 (Coherence [BTCGS91]) *If Δ_1, Δ_2 are two derivations of our source language yielding the same typing judgment then their translations yield provably equal terms in* **TARGET**

Proof. The proof is much the same as the one in [BTCGS91] although it simplifies it a little bit. It is not surprising that the main differences lie in the proof of the coherence of the translation of the subtyping derivation, since it is where the systems differ. Thus the reader can just follow the proof in section 5 of the cited paper, here we point out the differences (lemmas typeset in SmallCaps refer to those in [BTCGS91]):

- in Lemma 6 use the equality
  $\mathsf{forall}(a)(\Lambda X.\lambda x^{X \multimap 1}.b) \odot \mathsf{forall}(c)(\Lambda X.\lambda y^{X \multimap 1}.d) = \mathsf{forall}(c \odot a)(\Lambda X.\lambda y^{X \multimap 1}.(b[x := y] \odot d)$
  to prove the equality of the translations when the derivation ends by a $(\forall)$ rule.

- Erase Lemma 7.

- Use the same proof for Lemma 8 (of course you no longer need to use Lemma 7).

- Extend the proof of lemma 8.4.9 of the previous section to take into account the new type constructors (it essentially amounts to add two cases that can be solved by a straightforward use of the induction hypothesis induction: the weight for a record/variant being the sum of the weights of its components): this corresponds to Lemma 11 of [BTCGS91] (and it is where that proof failed for $F_\leq$).

- The proof of LEMMA 12 is nearly the same: just replace in the case [$\forall$ELIM] (in the paper [B-SPEC]) every occurrence of $a \leq s_i$ and $a \leq r$ in the tcs's by $a \leq \mathsf{Top}$ (in [BTCGS91] type variables are denoted by a ... sigh!) and use the (FORALL) version of the lemma; the sublemma requires a slight modification but on the other hand the proof is simpler since you no longer need to use LEMMAS 6 and 7 in the proof of this case.

- THEOREM 13 is unchanged

□

For a review of the models of **TARGET** see section 6 of [BTCGS91].

8.6 Conservativity of Recursive Types

In section 8.5 we have seen a very naive use of recursive types: the only rule to subtype recursive types was reflexivity. Here we study a deeper use of recursion. We add to the subtyping rules of section 8.2 the following rules

(Unfold-L) $$\frac{C \vdash S[X := \mu X.S] \leq T}{C \vdash \mu X.S \leq T}$$

(Unfold-R) $$\frac{C \vdash T \leq S[X := \mu X.S]}{C \vdash T \leq \mu X.S}$$

Let us denote the new system by $F_{\leq}^{\top\mu}$ and the corresponding judgments by $\vdash_\mu$. In [Ghe93b] it is shown that adding these two rules to $F_{\leq}$ leads to a non-conservative extension of $F_{\leq}$ w.r.t. the theory of subtyping, in the sense that there is a subtyping judgment in $F_{\leq}$ which is not provable in the original system but which is provable with the extended system. In that same paper a characterization is given for the pairs of types for which the conservativity does not hold.

Here we can show that $F_{\leq}^{\top\mu}$ is a conservative extension of $F_{\leq}^{\top}$; unfortunately we cannot use the nice characterization of [Ghe93b] for the non-conservative judgments since the set of non-provable judgments in our system is strictly larger than the one in $F_{\leq}$, and thus there might be a judgment that makes the conservativity fail in $F_{\leq}^{\top}$, but that is provable in $F_{\leq}$ (whence it would not be taken into account by the characterization above). Therefore we prove conservativity directly, by showing that if a judgment not containing recursive types is provable in $F_{\leq}^{\top\mu}$ then it is provable in $F_{\leq}^{\top}$. More exactly we prove that if $C \vdash_\mu S \leq T$ is provable and C, S, T do not contain recursive types then $C \vdash S \leq T$ is provable, too. This can be obtained very straightforwardly by replacing in $\vdash_\mu$ the full transitivity by a transitivity on the variables. Thus replace the rule (trans) by the rule

(AlgTrans) $$\frac{C \vdash C(X) \leq T}{C \vdash X \leq T}$$

We denote the corresponding proof system by $\vdash_{\mu\mathcal{A}}$.

Let us see more precisely the types we take into account; they are those of $F_{\leq}^{\top}$ plus the recursive types:

$$T ::= X \mid T \to T \mid \forall(X \leq S)T \mid \mu X.T$$

As in [Ghe93b] we forbid recursive types whose body is either Y, Top or $\mu X.T$; this will simplify the proofs without changing the expressivity of the system: indeed the first two bodies can be forbidden since $\mu X.Y$ ($Y \not\equiv X$), and $\mu X.\mathsf{Top}$ denote just Y and Top, $\mu X.X$ is meaningless (this is the only real restriction), while $\mu X.\mu Y.T$ can be forbidden since it has the same unfolding as $\mu X.T[Y := X]$. From a formal point of view we had to distinguish syntactically the variable for recursion from those for universal quantification, since the former has only to appear in a tcs C while the latter must also be associated with a bound in C, but it is not essential to the purpose of this section.

Lemma 8.6.1 *If there is a proof of $C \vdash_{\mu\mathcal{A}} S \leq T$ and no recursive type appears in the judgment then no unfold rule appears in the proof.*

Proof. A trivial induction on the depth of the proof of $C \vdash_{\mu\mathcal{A}} S \leq T$. □

In order to shorten the statement of the lemma that follows we introduce the notion "having the same shape".

Definition 8.6.2 Two types *have the same shape* if they are both type variables, or both Top, or both arrow types, or both parametric types quantified over the same type variable, or both recursive types quantified over the same type variable. □

Now we can to prove a lemma which roughly corresponds to the lemma 8.4.1 of section 8.4:

Lemma 8.6.3 *Let $C \vdash_{\mu\mathcal{A}} T_1 \leq T_2$. Then*

1. *If T_1 is not a variable then one of these cases holds:*
   - *T_2 is Top*
   - *T_2 has the same shape as T_1*
   - *$T_1 \equiv \mu X.T$ and T_2 has the same shape as T*
   - *$T_2 \equiv \mu X.T$ and T_1 has the same shape as T*
2. *If T_2 is not Top then one of these cases holds:*
   - *T_1 is a variable*
   - *T_1 has the same shape as T_2*
   - *$T_2 \equiv \mu X.T$ and T_1 has the same shape as T*
   - *$T_1 \equiv \mu X.T$ and T_2 has the same shape as T*

Proof. By induction on the depth of the proof of $C \vdash_{\mu\mathcal{A}} T_1 \leq T_2$. We perform a case analysis on the last rule of the proof: when the last rule is (refl), ($\rightarrow$) or ($\forall$) then result follows since T_1 and T_2 have the same shape. When the last rule is (Top) or (VarTrans) both the points of the lemma are satisfied, too. Suppose that the last rule is (unfold-L): then $T_1 \equiv \mu X.T$ and $C \vdash_{\mu\mathcal{A}} T[X := T_1] \leq T_2$. By construction T can be neither a variable nor a recursive type. Thus by induction hypothesis on the point 1. there remain three subcases:

a $T_2 \equiv \mathsf{Top}$ and the result is immediate.

b $T[X := T_1]$ has the same shape as T_2. But since T is not a variable this implies that also T has the same shape as T_2 which gives the result.

c T_2 is a recursive type, but then it has the same shape as T_1.

The case for (unfold-R) is similar: use the hypothesis that T cannot be Top. □

Finally we have the lemma on the transitivity elimination:

Lemma 8.6.4 *For each tcs C and types T_1, T_2, and T_3 well-formed in C, if $C \vdash_{\mu\mathcal{A}} T_1 \leq T_2$ and $C \vdash_{\mu\mathcal{A}} T_2 \leq T_3$ then $C \vdash_{\mu\mathcal{A}} T_1 \leq T_3$.*

Proof. The proof is very similar to the one of lemma 8.4.3. Again we prove the lemma by induction on the sum of the depths d_i of the proofs of $C \vdash_{\mu\mathcal{A}} T_i \leq T_{i+1}$ $(i = 1, 2)$, only for the cases in which T_2 and T_3 are different from Top. Moreover we can also suppose that the last rule of $C \vdash_{\mu\mathcal{A}} T_2 \leq T_3$ is different from (refl), since in that case the result trivially holds. Let us perform a case analysis on the last rule of the proof of $C \vdash_{\mu\mathcal{A}} T_1 \leq T_2$:

(refl) the result then gets $C \vdash_{\mu\mathcal{A}} T_1 \equiv T_2 \leq T_3$ satisfied by hypothesis. Note that this case solves the base of the induction for which $(d_1 + d_2) = 2$, i.e. when $T_1 \equiv T_2 \equiv T_3$.

(VarTrans) thus $T_1 \equiv X$. By induction hypothesis we obtain that $C \vdash_{\mu\mathcal{A}} C(X) \leq T_3$; by applying (VarTrans) we obtain the result.

(Top) not possible for $T_2 \not\equiv \mathsf{Top}$.

$(\rightarrow)$ then $T_1 \equiv T_1' \rightarrow T_1''$, $T_2 \equiv T_2' \rightarrow T_2''$. There are two subcases and both of them are proved by applying the induction hypothesis on $C \vdash_{\mu\mathcal{A}} T_3' \leq T_1'$ and $C \vdash_{\mu\mathcal{A}} T_1'' \leq T_3''$. Indeed, since $T_3 \not\equiv \mathsf{Top}$ then by lemma 8.6.3 there are two possible cases:
1. $T_3 \equiv T_3' \rightarrow T_3''$.
2. $T_3 \equiv \mu X.T' \rightarrow T''$. Then set $T_3' \equiv T'[X := T_3]$ and $T_3'' \equiv T''[X := T_3]$

in both cases the thesis is obtained by applying the induction hypothesis and $(\rightarrow)$ or (unfold-R). we obtain the result.

$(\forall)$ as the case before.

(unfold-L) This case is very easy since T_2 is on the right-hand side also in the premise of the rule. Thus the proof is a trivial application of the induction hypothesis.

(unfold-R) In this case we have that $T_2 \equiv \mu X.T$. Then there are two possible cases:

1. The last rule of $C \vdash_{\mu\mathcal{A}} T_2 \leq T_3$ is (unfold-L). But then we can apply the induction hypothesis on $C \vdash_{\mu\mathcal{A}} T_1 \leq T[X := T_2]$ and $C \vdash_{\mu\mathcal{A}} T[X := T_2] \leq T_3$.
2. The last rule of $C \vdash_{\mu\mathcal{A}} T_2 \leq T_3$ is (unfold-R) but then $T_3 \equiv \mu Y.T'$. Thus we can apply the induction hypothesis to $C \vdash_{\mu\mathcal{A}} T_1 \leq T_2$ and $C \vdash_{\mu\mathcal{A}} T_2 \leq T'[Y := T_3]$. By applying once more (unfold-R) we obtain the result.

□

Corollary 8.6.5 *If $C \vdash_{\mu} S \leq T$ is provable then $C \vdash_{\mu\mathcal{A}} S \leq T$ is also provable.*

Proof. By induction on the depth of the proof of $C \vdash_{\mu} S \leq T$. Apply lemma 8.6.4 when the last rule is (trans). □

8.7 The typing relation

As we have shown, the passage from the rule (∀-orig) to the rule (∀-new) brings many benefits to the subtyping system, which then enjoys many desirable properties. Unfortunately, that same passage has a nasty effect on the typing relation. Let us define the typing relation for $F_{\leq}^{\top}$ by using exactly the same rules of $F_{\leq}$, as they are summarized in section 7.1.3. The difference between the two systems would only be in the subtyping relations used by the subsumption rule. But, Giorgio Ghelli has remarked that, then, $F_{\leq}^{\top}$ does not enjoy the minimal typing property; in the sense that the set of all types proved by a given term may not have a least element [Giorgio Ghelli, personal communication, 1994]. This can be shown by the following example due to Ghelli. Consider the term $a \equiv \Lambda X \leq Y.\lambda x^X.x$. By subsumption, one can prove that it is typed both by $\forall(X{\leq}Y)X \to X$ and $\forall(X{\leq}Y)X \to Y$. Yet, these two types are both minimal w.r.t. the set of the types of a. The consequence of this fact is that the "standard" typing algorithm for $F_{\leq}^{\top}$ is correct but not complete. Take the standard typing algorithm for $F_{\leq}$, which is described in appendix E.3. Use the algorithm of section 8.4.1 to check the subtyping judgements appearing in the typing rules. Then, the resulting typing algorithm is correct but not complete, since, for example, it will deduce for a above the first type, but not the second one. The problem of finding a complete typing algorithm in this case remains an open problem, as well as the problem of decidability of the typing relation.

The culprit of the loss of this property is the interaction between the subsumption rule and the (∀-new) rule. Indeed, by subsumption, one is allowed to "raise" the type of the body of a Λ-abstraction, while the rule (∀-new) prevents the typing algorithm from doing the same. But note that the tests we performed on the existing interpreters for $F_{\leq}$ consisted of replacing in the subtyping algorithm the original rule (∀-orig) by (∀-new), but continuing to use the usual typing algorithm. Thus, no use of subsumption was done there. Therefore, one could continue like this, justified by a philosophy of cutting off useless judgments (which underlies the whole definition of $F_{\leq}^{\top}$), and consider as the typing relation the one defined by the typing algorithm for $F_{\leq}$ and the new subtyping relation (thus, take the rules of appendix E.3 as the typing rules for $F_{\leq}^{\top}$). However, another problem comes up, that is, the resulting system does not satisfy the subject reduction property. Consider the term $\Lambda X \leq Y.\lambda y^Y.(\lambda x^X.x)y$. The typing algorithm returns for this term, the type $\forall(X{\leq}Y)Y \to X$. This term reduces in one step to $\Lambda X{\leq}Y.\lambda y^Y.y$ which has type $\forall(X{\leq}Y)Y \to Y$, incomparable (by the (∀-new) rule) with the previous type (this example is due to Benjamin Pierce).

Now, take this same system (i.e. the $F_{\leq}^{\top}$-subtyping rules plus the typing rules in the appendix E.3), but do not allow reductions inside Λ-abstractions (or reductions involving free type variables); thus, remove not only some "useless" (typing and subtyping) judgments but also some "useless" reductions. Then, for this system, which possesses a decidable typing relation (obvious since every well-typed term has just one type: the one returned by the algorithm) and a subtyping relation with all the nice properties shown in this chapter, we conjecture the subject-reduction property. And note that this system is the one that we implicitly tested by using the typing algorithm of the $F_{\leq}$ implementations of Cardelli and of Pierce and Turner, with the modified subtyping relation.

8.8 Conclusions

We have presented in this chapter a new formulation of bounded quantification, enjoying many properties that the "canonical" formulation, $F_{\leq}$, lacks. This system also has a large tolerance for extensions, as we showed in the case of recursive types and for the semantics. This make us believe that it also retain conservativity if we use with recursive types the subtyping relation of Amadio and Cardelli [AC90] (in this case however we lose the nice property of having at premises of a rule essentially the same tcs's of the conclusion; thus transitivity elimination becomes again an hard problem that probably requires the same technique as in [CG92]).

It is actually under work the proof that the extension of this system by the intersection types enjoys the property of existence of glbs for downward bounded pairs of types, and lubs. And of course we are working for proving that the system without subsumption and reductions involving free type variables satisfies the subject-reduction property.

Finally, we believe that there exists a polynomial subtyping algorithm for $F_{\leq}^{\top}$, for it does not seem very difficult to adapt the polynomial algorithm recently discovered by Giorgio Ghelli for Fun, to the (sub)typing discipline of $F_{\leq}^{\top}$.

Chapter 9

Bounded quantification with overloading

System F is a language that permits us to write functions that take types as inputs; however these functions depend on their input in a very strict way: different input types affect just the type of the result, not its value. The practical counterpart of this observation is given by the fact that types are thrown away during the computation which is then performed on the *erasures* of the terms. $F_{\leq}$, that we have studied in the previous chapters, is a conservative extension of F, which allows us to specify bounds on the types that can be passed to a function; the type-checker uses this further information to type the body of the function. Though the functions of $F_{\leq}$ still have the same kind of dependence as in System F, since types again disappear during the computation. Here we want to extend $F_{\leq}$ by a type dependency also affecting the computation. We want to have functions that dispatch on different codes according to the type passed as argument. As a side effect, types will no longer be erasable at runtime.

This research fits into a larger framework: In language theory, polymorphism has two orthogonal classifications: "parametric vs. *ad hoc*" (see [Str67]) and "explicit vs. implicit". Parametric polymorphism, i.e. the capability of performing the same code on different types, has been widely studied, both in the explicit form (where types participate directly in the syntax; e.g. System F) and in the implicit one (where types participate via the terms they type; e.g. ML). "Ad hoc" polymorphism, i.e. the capability of performing a different code for each different type, has not received the same attention. In chapter 2, with the definition of the $\lambda\&$-calculus, we started a theoretical analysis of simply typed "ad hoc" polymorphism. In this chapter we tackle the second order *explicit* counterpart, by defining $F_{\leq}^{\&}$ a calculus with subtyping, which integrates parametric and "ad hoc" explicit polymorphism.

9.1 The loss of information in the overloading-based model

This extension has not a mere logical interest but it is strongly motivated by the modeling of object-oriented languages and the definition of a type discipline to strongly type them. A first motivation of this study was given in section 6.5, as an important step toward the definition of a mathematical meaning of the $\lambda\&$-calculus. But the main interest of this extension in the

framework of this thesis is that it solves a problem of loss of information analogous to the one we described in section 7.1 for the record-based models.

Let us try to be more specific. Suppose we have a message m' which modifies the internal state of a class C_1. Since we are in a functional approach the method in C_1 returns a *new* object of class C_1. Thus $m'\colon\{...,C_1 \to C_1,...\}$. Let C_2 be a subclass of C_1 from which it inherits the method at issue. If we pass the message m' to an object of C_2 then the branch defined in C_1 is selected. Since this branch has type $C_1 \to C_1$, the result of message passing has type C_1, rather than C_2 as would be natural. Note that we already met this problem at the very beginning of the thesis. In example 1.1.3, page 69, we defined in the class `2DPoint` a method for the message erase. Then in example 1.1.4 we defined a subclass `2DColPoint` which inherited `erase` from `2DPoint`. Thus in, λ&, `erase` would have been implemented by an overloaded function of just one branch, i.e. $erase : \{2DPoint \to 2DPoint\}$. Passing an object of class *2DColPoint* to *erase* would give an object of type *2DPoint*. And in fact the type system define in section 5.1.2, would have assigned to the term `[ new(2DColPoint) erase]` the type `2DPoint`.[1]

As you can see, this problem is very close to the one that Cardelli [Car88] pointed out for the record-based model. In our case the problem is less important than in Cardelli's calculus: indeed, in the case above we could imagine to add to m' a fake branch $C_2 \to C_2$ which would be used only during the phase of type-checking and then it would be discarded[2] (this has been done in [Cas92]). However this solution is interesting only in practical cases, where there is a finite number of classes; otherwise an infinite branching would be required. Although this solution works whenever the set of classes has a well-founded ordering (as it is always the case in practice) it becomes unmanageable when one starts to distinguish subtyping from subclassing (as done in [CHC90]) and it gives no suggestion on how to define polymorphic type inference: it is just a patchwork. In conclusion we need a new brand type system to account for this problem.

As we already said, the solution adopted for the record-based model was to pass to a second order formalism.

Here we adopt the same solution w.r.t. the λ&-calculus. The rough idea is to have a type system to type the previous m' in the following way:

$$m'\colon\{..., \forall X{\leq}C_1.X \to X, ...\}$$

For this reason in this chapter we define $F_{\leq}^{\&}$ where this type dependency is dealt with in an explicit way[3].

---

[1] This is the reason why it was not very important to be able to define in λ& a polymorphic operation to update record values (see section 2.5.3): the eventual polymorphism would have been hidden by this problem of loss of information. In this chapter, on the contrary it will be very important to have such an operation, and the encodings of $F_{\leq}$ defined in [Car92] will do the work.

[2] Note that such a solution cannot be used also for the record-based model. We cannot replace the function, say, $\lambda x\colon C_1.x$ by $\lambda x\colon C_2.x$ since the latter would no longer accept inputs of type C_1.

[3] The other solution is to deal with it in an implicit way by introducing type schemas *à la* ML, with bounds on the generic variables. Its study is presently under way on the base of the results of this chapter.

9.1.1 Type dependency

In a programming language a function which performs a dispatch on a type passed as argument would probably be written as:

```
Fun(X:*) => if X<T1 then exp1 else if X<T2 then exp2 ... else if X<Tn then exp_n
```

This function executes exp_1 if we pass a type less than or equal to T_1, exp_2 if it is less than or equal to T_2 and so on. If there is more than one candidate we select among them the branch with the least bound. In $F_{\leq}^{\&}$ this function is denoted by:

$$(\Lambda X{\leq}T_1.exp_1 \;\&\; \Lambda X{\leq}T_2.exp_2 \;\&\ldots\&\; \Lambda X{\leq}T_n.exp_n)$$

and its type is $\forall X\{T_1.S_1, T_2.S_2, \ldots, T_n.S_n\}$ (where $exp_i\colon S_i$). However this type is a rough approximation yet. Indeed, to obtain a coherent and expressive system, we need strong restrictions on the T_i's and the S_i's.

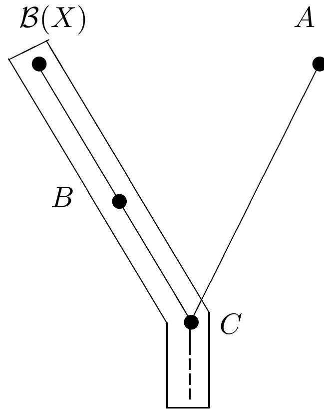
First of all note that the selected branch may change during the computation. For example take a function f of type $\forall X\{T_1.S_1, T_2.S_2\}$ with $T_2 \leq T_1$. Consider now the expression $(\Lambda Y{\leq}T_1.f[Y])$. Since $Y{\leq}T_1$ we guess that the selected branch will be the one associated to T_1 and thus the type of this expression will be $\forall(X{\leq}T_1)S_1$ (more exactly $\forall(Y{\leq}T_1)S_1[X\!:=Y]$). But if we pass to the function above the type T_2 then, being Y bound to T_2, the selected branch will be the second one and the result will have type S_2. System F and $F_{\leq}$ satisfy the subject reduction property, i.e. types are preserved under reductions. If we want reductions to preserve the type also in the new system we must require S_2 to be the same type as S_1. Though, this happens to be too strong a condition to model object-oriented languages (see the examples in section 10.1). Thus we adopt a less restrictive discipline, according to which types are allowed to decrease during computation. Thus in the example above it must be possible to deduce $X{\leq}T_2 \vdash S_2{\leq}S_1$. Summing up, the first restriction we impose on an overloaded type $\forall X\{T_i.S_i\}$ is that if $\vdash T_i{\leq}T_j$ then $X{\leq}T_i \vdash S_i{\leq}S_j$ (we call it the *covariance condition*, since it corresponds to the homonymous condition of $\lambda\&$). Note the use of sequents: the premise records the subtyping relation on the type variables; we already met this in the previous chapter, where the premise were called type constraint system: see the definition 8.2.1.

By definition 8.2.1, for a given tcs C and a type variable $X \in dom(C)$ there always exists a least non variable type T greater than X. We denote it by $\mathcal{B}(X)_C$ (the $\mathcal{B}$ stands for *bound*). More precisely we have the following definition.

Definition 9.1.1 Let C be a tcs and T a (raw) type such that $FV(T) \subseteq dom(C)$ then
1. $\mathcal{B}(T)_C = T$ if T is not a type variable.
2. $\mathcal{B}(T)_C = \mathcal{B}(C(T))_C$ otherwise. □

In the rest of the chapter we omit the subscript C in $\mathcal{B}(T)_C$ when it is clear from the context.

We limit our study to the case where the bounds of an overloaded function range over basic types (e.g. Bool, Int, Real ...). Indeed, the use of arrow types in the bounds poses many non-trivial problems, due to the contravariance of the left argument in the subtyping relation.



Therefore the second restriction we impose is that $\forall X\{T_i.S_i\}_{i\in I}$ is well-formed only if for every $i\in I$, $\mathcal{B}(T_i)$ is a basic type[2]. Thus every bound T_i must be an atomic type, i.e. either a basic type or a type variable. When the bound is a type variable, say X, the basic type $\mathcal{B}(X)$ plays an important role, since the set of its subtypes (denoted by $\mathcal{P}(\mathcal{B}(X))$) is the range of X. When we apply a type to an overloaded function, a selection rule picks the branch to execute. As we already said, this rule selects the branches with a bound provably larger than or equal to the type passed as argument, and among them it chooses the one with the least bound. Some conditions are required to assure that this minimum exists. In λ&-calculus this was assured by requiring that the bounds had to form a partial downward semi-lattice.[4] But there we had only closed types. Now with type variables this restriction no longer suffices: consider the example of the figure above; it is clear that X and A have no common lower bound. Nevertheless if we give to X the value B, it can come into conflict with A since they have a common lower bound C. Thus if a variable X appears in an overloaded type as a bound then conflicts must be checked taking into account every type in $\mathcal{P}(\mathcal{B}(X))$. To this purpose we require that every set of bounds satisfies the property of $\cap$-*closure*, defined as follows:

Definition 9.1.2 Let C be a type constraint system. Given a set of atomic types $\{A_i\}_{i\in I}$ we write $C \vdash \{A_i\}_{i\in I}\cap$-closed if and only if for all $i,j \in I$ if $\mathcal{B}(A_i)_C {\Downarrow} \mathcal{B}(A_j)_C$ then there exists $h \in I$ such that $C \vdash A_h = A_i \cap A_j$. $\Box$

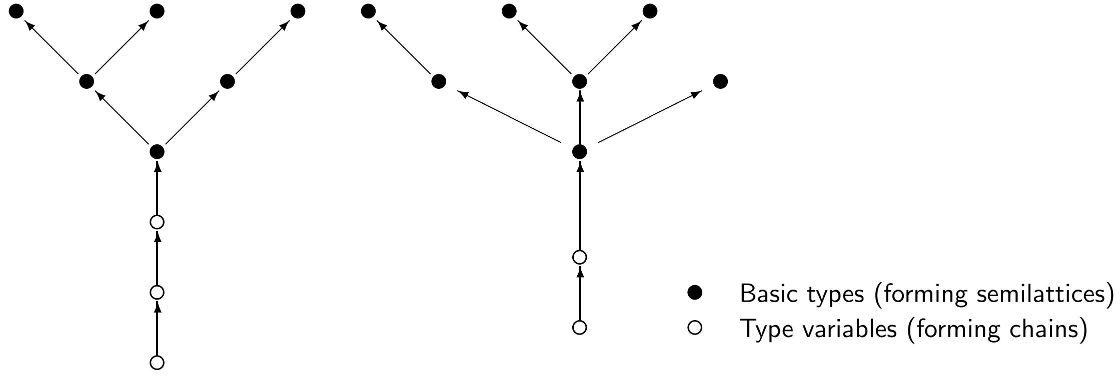
where $C \vdash A_h = A_i \cap A_j$ means that from C it is provable that A_h is the g.l.b. of A_i and A_j, and $B_1 \Downarrow B_2$ that B_1 and B_2 have a common lower bound.

Note that $\cap$-closure is quite a draconian restriction. Indeed $\cap-$closed sets of bounds have a very precise form (see proposition 9.2.5): they are partial downward semi-lattices, i.e. formed by disjoint unions of downward semi-lattices. These semi-lattices are divided in two parts: the upper part is a semi-lattice formed only by basic types; the lower part is formed by a chain of type variables starting from the least element of a semi-lattice of basic types. Any of these two parts may be missing. A pictorial representation of the situation is the following one:

---

[2] The major drawback of this restriction is that we cannot obtain the quantification of system F as a special case of the overloaded one and thus we will be obliged to add it explicitly: cf. section 10.2.

[4] A set S is a partial downward semi-lattice if and only if for all $a, b \in S$ if $a \Downarrow b$ then $a \cap b \in S$. Where, as usual, $a \Downarrow b$ means that a and b have a common lower bound (in S) and $a \cap b$ denotes their greatest common lower bound.



9.2 Type system

In this section we describe the type system. We first define the raw types. Among them we select the types, i.e. those raw types that do not contain overloaded types not satisfying the three rules we hinted in the introduction. In other terms $\forall X\{A_i.T_i\}_{i\in I}$ must:

1. have bounds ranging over basic types, i.e. for each $i\in I$ $\mathcal{B}(A_i)$ must be a basic type.
2. have a $\cap$-closed set of bounds.
3. satisfy covariance, i.e. if $A_i \leq A_j$ then $X\leq A_i \vdash T_i\leq T_j$

We assume that we have a predefined ordering on basic types which must form a partial lattice. This partial order is extended to higher types by a set of subtyping rules that are mutually recursive with those selecting the types.

Raw Types

$$A \;::=\; X \mid B \qquad \text{(atomic types [}B\text{ basic types])}$$

$$\begin{array}{lll} T & ::= & A \mid \mathsf{Top} \qquad\qquad \text{(raw } F_{\leq}^{\&} \text{ types)} \\ & \mid & T \to T \\ & \mid & \forall(X\leq T)T \\ & \mid & \forall X\{A_1.T_1,\ldots,A_n.T_n\} \qquad \text{(also denoted by } \forall X\{A_i.T_i\}_{i=1..n}) \end{array}$$

Judgments
We have three kinds of judgment: for type good-formation ($C \vdash T$ type), for the subtyping relation ($C \vdash T\leq T'$) and for the typing relation ($C \vdash a{:}T$). We call the first two kinds of judgments *type judgments*. Along the chapter we also use some informal judgements: for example "$C\vdash T = \min_{i\in I}\{T_i\}$" stands for "$T \in \{T_i\}_{i\in I}$ and for all $i\in I$ $C\vdash T \leq T_i$".

Types

(Basic$_{type}$) $\qquad C\vdash B$ type

(Top$_{type}$) $\qquad C\vdash \mathsf{Top}$ type

(Vars$_{type}$) $$\frac{C \vdash T \text{ type}}{C \cup \{X{\leq}T\} \vdash X \text{ type}} \qquad X \notin dom(C)$$

($\rightarrow_{type}$) $$\frac{C \vdash T \text{ type} \qquad C \vdash T' \text{ type}}{C \vdash T \rightarrow T' \text{ type}}$$

($\forall_{type}$) $$\frac{C \cup \{X{\leq}T\} \vdash T' \text{ type} \qquad C \vdash T \text{ type}}{C \vdash \forall(X{\leq}T)T' \text{ type}} \qquad X \notin dom(C)$$

($\{\}_{type}$) $$\frac{\begin{array}{c} C \vdash A_i \text{ type} \\ C \vdash \{A_i\}_{i=1..n} \ \cap\text{-closed} \\ C \cup \{X{\leq}A_i\} \vdash T_i \text{ type} \\ \text{if } C \vdash A_i \leq A_j \text{ then } C \cup \{X{\leq}A_i\} \vdash T_i{\leq}T_j \end{array}}{C \vdash \forall X\{A_1.T_1, \ldots, A_n.T_n\} \text{ type}} \qquad \begin{array}{c} X \notin dom(C) \\ \mathcal{B}(A_i)_C \text{ basic type } (i = 1..n) \\ \text{for } i,j \in [1..n] \end{array}$$

Subtyping

(refl) $$\frac{C \vdash T \text{ type}}{C \vdash T{\leq}T}$$

(trans) $$\frac{C \vdash T_1 \leq T_2 \qquad C \vdash T_2 \leq T_3}{C \vdash T_1 \leq T_3}$$

(taut) $$\frac{C \vdash T \text{ type}}{C \cup \{X{\leq}T\} \vdash X{\leq}T} \qquad X \notin dom(C)$$

(Top) $$\frac{C \vdash T \text{ type}}{C \vdash T{\leq}\mathsf{Top}}$$

($\rightarrow$) $$\frac{C \vdash T_1'{\leq}T_1 \qquad C \vdash T_2{\leq}T_2'}{C \vdash T_1 \rightarrow T_2{\leq}T_1' \rightarrow T_2'}$$

($\forall$) $$\frac{C \vdash T_1'{\leq}T_1 \qquad C \cup \{X{\leq}T_1'\} \vdash T_2{\leq}T_2' \qquad C \vdash \forall(X{\leq}T_1)T_2 \text{ type}}{C \vdash \forall(X{\leq}T_1)T_2{\leq}\forall(X{\leq}T_1')T_2'} \qquad X \notin dom(C)$$

($\{\}$) $$\frac{\begin{array}{c} C \vdash \forall X\{A_j.T_j\}_{j\in J} \text{ type} \quad C \vdash \forall X\{A_i'.T_i'\}_{i\in I} \text{ type} \\ \text{for all } i \in I \text{ exists } j \in J \text{ s.t.} C \vdash A_i'{\leq}A_j \quad C \cup \{X{\leq}A_i'\} \vdash T_j{\leq}T_i' \end{array}}{C \vdash \forall X\{A_j.T_j\}_{j\in J}{\leq}\forall X\{A_i'.T_i'\}_{i\in I}} \qquad X \notin dom(C)$$

9.2.1 Some useful results

Theorem 9.2.1 *If* $C \vdash T{\leq}T'$ *then* $C \vdash T$ type *and* $C \vdash T'$ type

Proof. By induction on the depth of the proof of $C \vdash T{\leq}T'$. □

Let us recast the terminology we introduced in the previous chapter: we say that two types *have the same shape* if they are both constant types or both type variables, or both Top, or both arrow types, or they are both overloaded or both parametric types quantified over the same variable.

The following result on the form of the judgements will be used frequently in the rest of the chapter

Proposition 9.2.2 *Let $C \vdash T_1 \leq T_2$. Then*

1. *If T_1 is not a variable then T_2 either is* Top *or it has the same shape as T_1*
2. *If T_2 is not* Top *then T_1 either is a variable or it has the same shape as T_2*

Proof. By induction on the depth of the proof of $C \vdash T_1 \leq T_2$, performing a case analysis on the last applied rule of the proof. □

Another useful fact that will be extensively used in the proofs of this chapter is the following one:

Proposition 9.2.3 *If $C \vdash T_1 \leq T_2$ then $C \vdash \mathcal{B}(T_1)_C \leq \mathcal{B}(T_2)_C$*

Proof. By induction on the number of steps to calculate $\mathcal{B}(T_1)_C$. If T_1 is not a type variable then, by proposition 9.2.2, T_2 cannot be a variable, therefore, by the definition of $\mathcal{B}()$, the result coincides with the hypothesis. If T_1 is a variable then to prove $C \vdash T_1 \leq T_2$ either we used only (refl) or we have used at least once (taut). In the first case the result is obvious; in the second case we obtain that $C \vdash C(X) \leq T_2$ and the result follows from the induction hypothesis. □

Lemma 9.2.4 *If $C \vdash X \leq Y$ then $\mathcal{B}(X)_C = \mathcal{B}(Y)_C$*

Proof. An easy induction on the number of steps of the definition of $\mathcal{B}(X)_C$. □

The following proposition describes the form of the ∩-closed set of types:

Proposition 9.2.5 *If $C \vdash \{A_i\}_{i \in I}$∩-closed then for any pair of elements A_i and A_j such that $\mathcal{B}(A_i)_C \Downarrow \mathcal{B}(A_j)_C$ one of this cases must hold:*

1. *$\mathcal{B}(A_i)_C$ and $\mathcal{B}(A_j)_C$ are unrelated (w.r.t. the subtyping relation), A_i and A_j are both basic types and their g.l.b. is in $\{A_i\}_{i \in I}$*
2. *$\mathcal{B}(A_i)_C \leq \mathcal{B}(A_j)_C$ and both A_i and A_j are basic types (or the reverse).*
3. *$\mathcal{B}(A_i)_C \leq \mathcal{B}(A_j)_C$, A_i is a variable and A_j is a basic type.*
4. *$\mathcal{B}(A_i)_C \leq \mathcal{B}(A_j)_C$, A_i and A_j are both variables and $C \vdash A_i \leq A_j$ (or the reverse).*

Proof. Let us examine all the possible cases:

1. A_i and A_j are both basic types. Then all the possible cases are covered by the first two points of the proposition.

2. A_i is a variable and A_j is a basic type[5] . Then we want to prove that $\mathcal{B}(A_i)_C \leq A_j$. Consider $A_h = A_i \cap A_j$. Since $C \vdash A_h \leq A_i$ the by proposition 9.2.2 A_h is a variable too. By lemma 9.2.4 $\mathcal{B}(A_h)_C = \mathcal{B}(A_i)_C$. Since $C \vdash A_h \leq A_j$ then by proposition 9.2.3 we obtain the result.
3. A_i and A_j are both variables. Consider $A_h = A_i \cap A_j$. By proposition 9.2.2 A_h is also a variable and by lemma 9.2.4 $\mathcal{B}(A_h)_C = \mathcal{B}(A_i)_C = \mathcal{B}(A_j)_C$. Thus both A_i and A_j appear in the chain from A_h to $\mathcal{B}(A_h)_C$. Therefore either $C \vdash A_i \leq A_j$ or $C \vdash A_j \leq A_i$ holds, according the order they appear in the chain.

□

9.2.2 Transitivity elimination

The rules of subtyping given above do not describe a deterministic algorithm: a subtyping judgment does not uniquely determine either the rule to prove it or the parameters that such a rule must have. In particular non-determinism is introduced by the rules (refl) and (trans):

Consider the judgment $C \vdash T \leq T$; if T is not a variable nor Top then the judgment can be proved by at least two different derivations, one consisting just of the rule (refl) the other obtained by first applying the structural rule for T (e.g. $(\rightarrow)$ if T is an arrow type) and successively the rule (refl) to the components of T. This kind of non-determinism can be easily solved by choosing either to use (refl) as soon as possible or to use it as late as possible (i.e. only on atomic types). We choose this second solution thus we substitute the rule (refl) above by the following one:

(refl) $$\frac{C \vdash A \text{ type}}{C \vdash A \leq A}$$

It is then very simple to prove that this new system is sound and complete w.r.t. the previous one: soundness is obvious and completeness is given by the following lemma:

Lemma 9.2.6 *For each C and T such that $C \vdash T$ type the judgment $C \vdash T \leq T$ is provable using reflexivity only on atomic types.*

Proof. A straightforward induction on the structure of T □

Also the rule (trans) produces a non-determinism similar to the one of (refl): we have always the choice to apply transitivity or to push it to the subcomponents. But, besides that, (trans) introduces a deeper form of non-determinism quite harder to eliminate in this case. Indeed, the (trans) rule does not respect the so-called "sub-formula property", according to which all the types appearing at the premises of a rule must appear in its consequence, too. When proving $T_1 \leq T_3$ by transitivity, a new level of non-determinism is introduced by the choice of the *intermediate type* T_2 such that $T_1 \leq T_2$ and $T_2 \leq T_3$.

The reader will have recognized in it a cut elimination problem. Indeed, transitivity elimination in subtyping systems corresponds to cut elimination in Gentzen's sequent calculus for the first order logic. Both of them lead to a coherence result of the corresponding proof system, by returning a canonical derivation for each provable judgment. The resemblance

[5] Without loss of generality, we can consider for this case and for the case 4 that $\mathcal{B}(A_i)_C \leq \mathcal{B}(A_j)_C$ holds. Thus in this proof and in those that follow we will skip the reverse case.

is even stronger since we can use the Gentzen's technique for cut elimination to prove also transitivity elimination[6]. Namely, we define a weakly normalizing rewriting system on the derivations of subtyping judgments. This system will push the transitivity rules towards the leaves of the derivation; whenever it has to choose between pushing transitivity up into a left or a right subderivation it (arbitrarily) chooses the one on the right. The derivations in normal form will have all the (trans) rules applied to a leaf of the derivation tree.

Since it is difficult to work directly with derivations, we use the Curry-Horward isomorphism [How80] to define a set a terms to uniquely codify subtyping derivations. We follow for their definition [CG92], where these terms are called *coercion expressions*.

The syntax of the coercion expressions is:

$$c \ ::= \ K_{B_1B_2} \mid Id_A \mid X_T \mid \mathsf{Top}_T \mid c \to c' \mid \forall(X{\leq}c)c' \mid c\,c' \mid \forall^{\phi}_{T}X\{c_1.c'_1,\ldots,c_n.c'_n\}$$

where ϕ denotes a total function between two sets of indexes $\phi : I \to J$.

We next show how to use coerce expressions to codify derivations. In the rules that follow we do not consider the judgements of type formation ($C \vdash T$ type) and we concentrate only on the subtyping judgements. Considering them would greatly complicate the exposition, without bringing any benefit: firstly the rules defining type formation describe a deterministic algorithm, and thus they do not pose any coherence problem; secondly, all the proofs in the rest of this section will work on a given type and on its syntactical sub-formulas; if we suppose by hypothesis that the type is well-formed then the proofs will be valid also when restricted to well-formed types (sub-formulae of well-formed types are well-formed types).

Thus the derivations we codify involve only subtyping judgements and work under the hypothesis that all the types appearing in them are well-formed. We also use (refl) defined only for atomic types.

(basic) $$C \vdash K_{B_1B_2}\colon B_1 \leq B_2 \qquad \text{for all basic types } B_1, B_2 \text{ s.t. } B_1{\leq}B_2$$

(refl) $$C \vdash Id_A\colon A{\leq}A$$

(trans) $$\frac{C \vdash c\colon T_1 \leq T_2 \qquad C \vdash c'\colon T_2 \leq T_3}{C \vdash c'\,c\colon T_1 \leq T_3}$$

(taut) $$C \cup \{X{\leq}T\} \vdash X_T\colon X{\leq}T$$

(Top) $$C \vdash \mathsf{Top}_T\colon T{\leq}\,\mathsf{Top}$$

($\to$) $$\frac{C \vdash c_1\colon T'_1{\leq}T_1 \qquad C \vdash c_2\colon T_2{\leq}T'_2}{C \vdash c_1 \to c_2\colon T_1 \to T_2{\leq}T'_1 \to T'_2}$$

[6] It would be more correct to say that we *must* use this technique: indeed the smooth technique used in the previous chapter for transitivity elimination (cf section 8.4.1) does not work here, even if the resulting system will be the same in both cases. See also section 9.5.1

$$(\forall)\qquad \frac{C\vdash c_1{:}\,T_1'{\leq}T_1 \qquad C\cup\{X{\leq}T_1'\}\vdash c_2{:}\,T_2{\leq}T_2'}{C\vdash \forall(X{\leq}c_1)c_2{:}\,\forall(X{\leq}T_1)T_2{\leq}\forall(X{\leq}T_1')T_2'}$$

$$(\{\,\})\qquad \frac{\forall i\in I \qquad C\vdash c_i{:}\,A_i'{\leq}A_{\phi(i)} \quad C\cup\{X{\leq}A_i'\}\vdash c_i'{:}\,T_{\phi(i)}{\leq}T_i'}{C\vdash \forall^{\phi}_{\forall X\{A_j.T_j\}}X.\{c_i.c_i'\}_{i\in I}{:}\,\forall X\{A_j.T_j\}_{j\in J}{\leq}\forall X\{A_i'.T_i'\}_{i\in I}} \qquad \phi{:}\,I\to J \text{ total}$$

Note that the term associated to transitivity is the composition of the terms associated to the sub-derivations.

The last rule shows the use of the function ϕ: during the subtyping of two overloaded types, ϕ associates each branch of the greater overloaded type with the branch in the smaller type to which it has been compared in the proof of subtyping. Note that this information would not suffice to uniquely determine the derivation codified by a given coercion expression; in case of overloaded types we need also to know the type on the left-hand side of the relation, which is recorded in the lower index of $\forall$.

Proposition 9.2.7 *There is a 1-1 correspondence between well-typed coerce expressions and subtyping derivations.*

Proof. A simple induction on the rules[7]. □

The rewriting system

We now define a rewriting system on the derivations of subtyping judgements. In view of the proposition 9.2.7, it is equivalent to define it directly on the coerce expressions. We borrow the rewriting system from [CG92], to which we add the rules ({}') and ({}") to deal with overloaded types. In the rules that follow we suppose that $C\vdash c{:}\,S{\leq}T$ and $C\vdash c_i{:}\,A_i\leq A'_{\phi(i)}$:

(Assoc)	$(c\,d)\,e$	$\leadsto$	$c\,(d\,e)$
$(\to')$	$(c\to d)\,(c'\to d')$	$\leadsto$	$(c'\,c)\to(d\,d')$
$(\to'')$	$(c\to d)\,((c'\to d')\,e)$	$\leadsto$	$((c'\,c)\to(d\,d'))\,e$
$(\forall')$	$(\forall(X{\leq}c)d)\,(\forall(X{\leq}c')d')$	$\leadsto$	$\forall(X{\leq}c'\,c)(d\,d'[X_T{:=}\,c\,X_S])$
$(\forall'')$	$(\forall(X{\leq}c)d)\,((\forall(X{\leq}c')d')\,e)$	$\leadsto$	$(\forall(X{\leq}c'\,c)(d\,d'[X_T{:=}\,c\,X_S]))\,e$
$(\{\}')$	$(\forall^{\phi}_{T}X\{c_i.d_i\}_{i\in I})\,(\forall^{\psi}_{T'}X\{c_j'.d_j'\}_{j\in J})$	$\leadsto$	$\forall^{\psi\circ\phi}_{T'}X\{c'_{\phi(i)}\,c_i.d_i\,(d'_{\phi(i)}[X_{A'_{\phi(i)}}:=X_{A_i}]\}_{i\in I}$
$(\{\}'')$	$(\forall^{\phi}_{T}X\{c_i.d_i\}_{i\in I})\,((\forall^{\psi}_{T'}X\{c_j'.d_j'\}_{j\in J})\,e)$	$\leadsto$	$(\forall^{\psi\circ\phi}_{T'}X\{c'_{\phi(i)}\,c_i.d_i\,(d'_{\phi(i)}[X_{A'_{\phi(i)}}:=X_{A_i}]\}_{i\in I})\,e$

A simple analysis of the rules shows that the normal forms of this rewriting system are subterms of $(c\to d)\,e_1\,\ldots\,e_n$ or of $(\forall(X\leq c)d)\,e_1\,\ldots\,e_n$ or of $(\forall^{\phi}_{T}X\{c_i.d_i\})\,e_1\,\ldots\,e_n$ where c,c_i,d,d_i are in normal form and $e_1,\ldots,e_n$ are either X_t or Top_T or $K_{BB'}$ (composition is left associative). These normal forms correspond to derivations in which every left premise of a (trans) rule is a leaf. Thus the rewriting system pushes the transitivity up to the leaves. It remains to prove two facts:

[7] Actually this theorem is true modulo weakenings of the tcs

1. The rewriting system is sound, i.e. it rewrites a valid derivation for a certain judgment into another valid derivation for the same judgment. By the Curry-Howard isomorphism it is equivalent to prove the subject reduction theorem for the calculus of the coercion expressions; namely we have to show that a well-typed coerce expression rewrites only to well-typed coerce expressions of the same type.
2. The rewriting system is weakly normalizing. In this case there exists a reduction strategy which transforms every derivation into another that proves the same judgment and is in normal form (i.e. with the (trans) rules at the right places).

Soundness of the rewriting system

The proof of the soundness of the rewriting system is very similar to the corresponding one in [CG92]. We first have to prove the following lemmas:

Lemma 9.2.8 (weakening) *If* $C \vdash c\colon \Delta$ *is provable and* $C \cup \{X \leq T\}$ *is a tcs then also* $C \cup \{X{\leq}T\} \vdash c\colon \Delta$ *is provable.*

Proof. By a simple induction on the proof of $C \vdash c\colon \Delta$ □

Lemma 9.2.9 (substitution) *If* $C \cup \{X{\leq}T\} \vdash c\colon U \leq V$ *and* $C \cup \{X{\leq}S\} \vdash d\colon X \leq T$ *are provable then* $C \cup \{X{\leq}S\} \vdash c[X_T\colon= d]\colon U \leq V$ *is provable too.*

Proof. By induction on the structure of c. We only detail the proof when c is a variable; all the other cases are either trivial ($K_{BB'}$, Top_T and Id_A) or they are solved by a straightforward use of the induction hypothesis ($\rightarrow$, $\forall$, $\forall^{\phi}_{T}$).
There are two subcases:
1. $c{\equiv}X_T$. The result gets $C \cup \{X{\leq}S\} \vdash d\colon X \leq T$ which is satisfied by hypothesis.
2. $c{\equiv}Y_V$. The hypothesis gets $C \cup \{X{\leq}T\} \vdash Y_V\colon Y \leq V$. Therefore $C \vdash Y_V\colon Y \leq V$. By a weakening (lemma 9.2.8) we obtain the result $C \cup \{X{\leq}S\} \vdash Y_V\colon Y \leq V$. □

Now we are able to prove the soundness of the rewriting system

Theorem 9.2.10 *If* $c \overset{*}{\leadsto} d$ *and* $C \vdash c\colon \Delta$ *then* $C \vdash d\colon \Delta$

Proof. It suffices to prove the theorem for one step of rewriting: the result is then obtain by induction on the number of steps. First of all, observe that if in a derivation we replace a sub-derivation by another one proving the same judgment then the new derivation is correct: only the final judgment matters. This simple observation handles all the cases where $c \leadsto d$ is not an instance of one of the previous rewriting rules, but it is obtained by rewriting a strict occurrence of c (alternatively this can be proved by induction on the structure of c and a case analysis on all the possible rewriting rules). Thus it remains to prove that the rewriting rules preserve correctness. We give the proof only for ({}') since it is the most representative and all the other cases are obtained by simplifying or slightly modifying this case; however, apart from the proof for ({}") which is very similar to the one we give here (but typographically even more awkward), all the remaining cases are included in the proof of section 5.2 in [CG92]. For typographical reasons we omit from the proof some obvious indexing, and inessential tcs's. By hypothesis we have:

$$
\dfrac{
\dfrac{\forall j{\in}J \quad \overset{\Pi^1_j}{\vdash c'_j\colon A^2_j{\leq}A^1_{\psi(j)}} \quad \overset{\Pi^2_j}{X{\leq}A^2_j \vdash d'_j\colon T^1_{\psi(j)}{\leq}T^2_j}}{\vdash \forall^{\psi}_{T'}X\{c'_j.d'_j\}\colon \forall X\{A^1_h.T^1_h\}{\leq}\forall X\{A^2_j.T^2_j\}}
\qquad
\dfrac{\forall i{\in}I \quad \overset{\Pi^3_i}{\vdash c_i\colon A^3_i{\leq}A^2_{\phi(i)}} \quad \overset{\Pi^4_i}{X{\leq}A^3_i \vdash d_i\colon T^2_{\phi(i)}{\leq}T^3_i}}{\vdash \forall^{\phi}_{T}X\{c_i.d_i\}\colon \forall X\{A^2_j.T^2_j\}{\leq}\forall X\{A^3_i.T^3_i\}}
}{
\vdash (\forall^{\phi}_{T}X\{c_i.d_i\}_{i\in I})\,(\forall^{\psi}_{T'}X\{c'_j.d'_j\}_{j\in J})\colon \forall X\{A^1_h.T^1_h\}_{h\in H} \leq \forall X\{A^3_i.T^3_i\}_{i\in I}
}
$$

where $T \equiv \forall X\{A^2_j.T^2_j\}_{j\in J}$ and $T' \equiv \forall X\{A^1_h.T^1_h\}_{h\in H}$. The rule ({}') transforms it in the following way:

$$
\dfrac{\forall i \in I \qquad \overset{\Sigma_i}{\vdash c'_{\phi(i)}\;c_i\colon A^3_i \leq A^1_{\psi(\phi(i))}} \qquad \overset{\Sigma'_i}{X{\leq}A^3_i \vdash d_i\;(d'_{\phi(i)}[X_{A^2_{\phi(i)}} := c_i\;X_{A^3_i}])\colon T^1_{\psi(\phi(i))}{\leq}T^3_i}}{\vdash \forall^{\psi\circ\phi}_{T'}X\{c'_{\phi(i)}\;c_i.d_i\;(d'_{\phi(i)}[X_{A^2_{\phi(i)}} := c_i\;X_{A^3_i}])\}_{i\in I} : \forall X\{A^1_h.T^1_h\}_{h\in H} \leq \forall X\{A^3_i.T^3_i\}_{i\in I}}
$$

where for all $i \in I$ the proof Σ_i is:

$$
\dfrac{\overset{\Pi^3_i}{\vdash c_i\colon A^3_i{\leq}A^2_{\phi(i)}} \qquad \overset{\Pi^1_{\phi(i)}}{\vdash c'_{\phi(i)}\colon A^2_{\phi(i)}{\leq}A^1_{\psi(\phi(i))}}}{\vdash c'_{\phi(i)}\;c_i\colon A^3_i \leq A^1_{\psi(\phi(i))}}
$$

and the proof Σ'_i is:

$$
\dfrac{
\dfrac{\overset{\Pi^2_{\phi(i)}}{X{\leq}A^2_{\phi(i)} \vdash d'_{\phi(i)}\colon T^1_{\psi(\phi(i))}{\leq}T^2_{\phi(i)}} \qquad \dfrac{X{\leq}A^3_i \vdash X_{A^3_i}\colon X{\leq}A^3_i \qquad \overset{(\Pi^3_i)^*}{X{\leq}A^3_i \vdash c_i\colon A^3_i{\leq}A^2_{\phi(i)}}}{X{\leq}A^3_i \vdash c_i\;X_{A^3_i}\colon X{\leq}A^2_{\phi(i)}}}{X{\leq}A^3_i \vdash d'_{\phi(i)}[X_{A^2_{\phi(i)}} := c_i\;X_{A^3_i}]\colon T^1_{\psi(\phi(i))}{\leq}T^2_{\phi(i)}}\;(\text{dotted})
\qquad \overset{\Pi^4_i}{X{\leq}A^3_i \vdash d_i\colon T^2_{\phi(i)}{\leq}T^3_i}
}{
X{\leq}A^3_i \vdash d_i\;(d'_{\phi(i)}[X_{A^2_{\phi(i)}} := c_i\;X_{A^3_i}])\colon T^1_{\psi(\phi(i))}{\leq}T^3_i
}
$$

In the last derivation the dotted line is proved by lemma 9.2.9 and $(\Pi^3_i)^*$ is obtained from Π^3_i by a weakening. In conclusion the rule ({}') preserves the correctness of the derivations. □

Weak normalization

The task of proving that the rewriting system is weakly normalizing is very simple since most of the work has already been done in [CG92]: define

$$\begin{aligned} size(A) = size(\mathsf{Top}) &\stackrel{def}{=} 1 \\ size(S \to T) = size(\forall(X{\leq}S)T) &\stackrel{def}{=} size(S) + size(T) \\ size(\forall X\{A_i.T_i\}_{i\in I}) &\stackrel{def}{=} \sum_{i\in I}(size(A_i) + size(T_i)) \end{aligned}$$

Let m and m' be two multisets of natural numbers; define

$$m < m' \stackrel{def}{\Longleftrightarrow} \forall n' \in m' \exists n \in m \;\; n < n'$$

Definition 9.2.11 ([CG92]) Define the *intermediate type* of a coerce composition $d\,e$, where $e{:}\,S{\leq}T$ and $d{:}\,T{\leq}U$, as the type T. Then the *complexity measure* of a coerce expression c is the multiset of the sizes of the intermediate types of all the redexes of c, modulo (Assoc). □

Theorem 9.2.12 *Every innermost strategy for $\leadsto$ strictly decreases the complexity measure and thus terminates.*

Proof. The proof is strictly the same as the one of section 5.3.3 in [CG92] modulo some slight modifications for the cases involving overloaded types. □

9.2.3 Subtyping algorithm and coherence of the system

Consider the following rewriting rules

(id$_l$)	$Id_T\, c$	$\leadsto$	c
(id$_r$)	$c\, Id_S$	$\leadsto$	c
(bas$'$)	$K_{BC}\, K_{AB}$	$\leadsto$	K_{AC}
(bas$''$)	$K_{BC}\,(K_{AB}\, c)$	$\leadsto$	$K_{AC}\, c$
(top)	$\mathsf{Top}_T\, c$	$\leadsto$	Top_S
(varTop)	X_{Top}	$\leadsto$	Top_X

These rules perform some cleaning of the derivations, basically by erasing useless coercions.

This set of rules clearly constitutes a strongly normalizing rewriting system (use as metrics for the coercion expressions the lexicographical order of the pairs formed by the number of compositions in the expression and by the number of variables occurring in it). Furthermore no rule increases the complexity measure given in the previous section for weak normalization, and they are all sound. Therefore we can safely add these rules to the previous rewriting system: all the results of the previous section still hold. In the rest of this section we will always consider the rewriting system formed by the old rules and those introduced above.

The shape of the normal forms

It is very important to analyze the shape of the normal forms of the composed rewriting system. We have the following theorem:

Proposition 9.2.13 *Every well-typed coerce expression in normal form has the form $c_0\, c_1 \dots c_n$ with $n \geq 0$, where c_0 can be any coerce expression different from a composition (of other coerce expressions) whose subformulae are in normal form, and $c_1 \dots c_n$ are variables.*

Proof. This proposition can be easily proved by induction on n. For $n = 0$ the result is obvious. The inductive case ($n > 0$) is proved by a case analysis on the shape of c_0, by using proposition 9.2.2 and the reduction rules. First of all note that because of the rewriting rules (top) and (id$_l$) c_0 can be neither Top_T nor Id_A:

$c_0 \equiv X_T$. Consider c_1. It cannot be a composition because of (Assoc). By proposition 9.2.2 it can be nothing but a variable: indeed we have that $c_1 \colon S \leq X$ thus S must be a type variable, say, Y and therefore $c_1 \equiv Y_X$. The result follows by induction hypothesis.

$c_0 \equiv K_{B_1 B_2}$. Consider c_1: it cannot be a composition because of (Assoc); it cannot be a basic type because of (bas$'$) if $n = 1$, because of (bas$''$) if $n > 1$; it cannot be Top_T or $c \to c'$ or $\forall(X{\leq}c)c'$ or $\forall^{\phi}_{T}X\{c_1.c'_1, \dots, c_n.c'_n\}$ because of proposition 9.2.2. Thus it can be but a variable. The result follows by induction hypothesis.

$c_0 \equiv c \to c'$. Consider c_1: it cannot be a composition because of (Assoc); it cannot be a $d \to d'$ because of ($\to'$) if $n = 1$, because of ($\to''$) if $n > 1$; it cannot be Top_T or $c \to c'$ or $\forall(X{\leq}c)c'$ or $\forall^{\phi}_{T}X\{c_1.c'_1, \dots, c_n.c'_n\}$ because of proposition 9.2.2. Thus it can be but a variable. The result follows by induction hypothesis.

All the other cases are solved as the last two cases. □

This theorem has two important consequences: the coherence of the proof system for the subtyping relation and the definition of a subtyping algorithm.

Coherence

Lemma 9.2.14 *For every provable subtyping judgment there exists only one coerce expression in normal form proving it.*

Proof. We follow the pattern of the proof of the corresponding proposition in [CG92]. Let c be a well-typed coercion expression in normal form. From propositions 9.2.13 and 9.2.2 it follows almost immediately that we have only these possible cases:

1. if $c\colon A \leq A$ then $c \equiv Id_A$
2. if $c\colon X \leq Y$ then c is a composition of variables, which is determined in an unique way by the tcs.
3. if $c\colon B_1 \leq B_2$ then $c \equiv K_{B_1 B_2}$
4. if $c\colon S \to S' {\leq} T \to T'$ then c is a $\to$ coercion.
5. if $c\colon \forall(X{\leq}S_1)S_2{\leq}\forall(X{\leq}T_1)T_2$ then c is a $\forall$ coercion.

6. if $c\colon\forall X\{A_j.T_j\}_{j\in J}\leq\forall X\{A'_i.T'_i\}_{i\in I}$ then c is a $\forall^{\phi}_{T}$ coercion.
7. if $c\colon X\leq B$ then c is a composition of variables, which is determined in an unique way by the tcs, composed with a coercion of case 3 if $\mathcal{B}(X)\neq B$
8. if $c\colon X\leq T\to T'$ then c is a composition of variables, which is determined in an unique way by the tcs, composed with a coercion of case 4 if $\mathcal{B}(X)\neq T\to T'$
9. if $c\colon X\leq\forall(X\leq T_1)T_2$ then c is a composition of variables, which is determined in an unique way by the tcs, composed with a coercion of case 5 if $\mathcal{B}(X)\neq\forall(X\leq T_1)T_2$
10. if $c\colon X\leq\forall X\{A'_i.T'_i\}_{i\in I}$ then c is a composition of variables, which is determined in an unique way by the tcs, composed with a coercion of case 6 if $\mathcal{B}(X)\neq\forall X\{A'_i.T'_i\}_{i\in I}$
11. if $c\colon T\leq\mathsf{Top}$ then c is Top_T

After this simple observation then the result can be proved by induction on the structure of c. □

Theorem 9.2.15 (coherence) *Let Π_1 and Π_2 be two proofs of the same judgment $C\vdash\Delta$. If c_1 and c_2 are the corresponding coerce expressions then c_1 and c_2 are equal modulo the rewriting system.*

Proof. By the weak normalization there exist two coercion expressions in normal form d_1 and d_2 such that $c_1\overset{*}{\leadsto}d_1$ and $c_2\overset{*}{\leadsto}d_2$. By the soundness of the rewriting system (theorem 9.2.10) it follows that $C\vdash d_1\colon\Delta$ and $C\vdash d_2\colon\Delta$. But then by lemma 9.2.14 we have that $d_1\equiv d_2$ (note that this constitutes also a proof that $\leadsto$ is Church-Rosser.) □

Subtyping algorithm

Consider once more the normal forms of proposition 9.2.13. These correspond to derivations in which every application of a (trans) rule has as left premise an application of the rule (taut). From this observation one directly derives the definition of the the following subtyping algorithm:

(AlgRefl) $$C\vdash_{\mathcal{A}} X\leq X$$

(AlgTrans) $$\frac{C\vdash_{\mathcal{A}} C(X)\leq T}{C\vdash_{\mathcal{A}} X\leq T}$$

(AlgTop) $$C\vdash_{\mathcal{A}} T\leq\mathsf{Top}$$

(Alg→) $$\frac{C\vdash_{\mathcal{A}} T'_1\leq T_1 \qquad C\vdash_{\mathcal{A}} T_2\leq T'_2}{C\vdash_{\mathcal{A}} T_1\to T_2\leq T'_1\to T'_2}$$

(Alg∀) $$\frac{C\vdash_{\mathcal{A}} T'_1\leq T_1 \qquad C\cup\{X\leq T'_1\}\vdash_{\mathcal{A}} T_2\leq T'_2}{C\vdash_{\mathcal{A}}\forall(X\leq T_1)T_2\leq\forall(X\leq T'_1)T'_2}\qquad X\notin dom(C)$$

(Alg{ }) $$\frac{\text{for all } i\in I \text{ exists } j\in J \text{ s.t.} C\vdash A'_i\leq A_j \quad C\cup\{X\leq A'_i\}\vdash T_j\leq T'_i}{C\vdash\forall X\{A_j.T_j\}_{j\in J}\leq\forall X\{A'_i.T'_i\}_{i\in I}}\qquad X\notin dom(C)$$

This set of rules denotes a deterministic algorithm since the form of the input —the judgment one has to prove— unequivocally determines the rule that must be used and all the parameters of any recursive calls

This algorithm is sound and complete w.r.t. our first system. This means that the sets of provable judgements of the two systems are the same. This is stated by the following theorem:

Theorem 9.2.16 $C \vdash_{\mathcal{A}} \Delta \iff C \vdash \Delta$

Proof. Soundness ($\Rightarrow$) is easily proved by induction on the depth of the derivation of $C \vdash_{\mathcal{A}} \Delta$. Completeness ($\Leftarrow$) stems directly from the work of this section: take any proof of $C \vdash \Delta$, apply to it the complete rewriting system with an innermost strategy; replace in the obtained normal form all the sequences of (taut) (trans) rules by an (AlgTrans) rule; add the index $\mathcal{A}$ to every turnstile and you have obtained a proof for $C \vdash_{\mathcal{A}} \Delta$. □

9.3 Terms

In this section we describe the terms of the language. We start by the definition of the *raw terms*, among which we distinguish the *terms*, i.e. those raw terms that possess a type. Roughly speaking, (raw) terms are divided in three classes: terms of the simply typed λ-calculus, terms for parametric polymorphism and terms for overloading. Overloaded functions are built in a list fashion, starting by an *empty* overloaded function ε and concatenating new branches by &. The &'s are indexed by a list of types which is used to type the term and to perform the selection of the branch.

Indexes

$$I ::= [A_1.T_1 \parallel \ldots \parallel A_n.T_n]$$

Raw Terms

$$\begin{array}{rcll} a & ::= & x^T \mid (\lambda x^T.a) \mid a(a) & \text{simply typed } \lambda\text{-calc} \\ & \mid & \mathsf{top} \mid \Lambda X{\leq}T.a \mid a(T) & F_{\leq} \\ & \mid & \varepsilon \mid (a\&^I a) \mid a[A] & \text{overloading} \end{array}$$

We required that the bounds of an overloaded function range over constant types. Therefore the argument of an overloaded function can be restricted to be an atomic type ($a[A]$) since a term of the form, say, $a[S \to T]$ would be surely rejected by the type checker.

Terms

We use two meta notations: $a[x := b]$, $a[X := S]$, $T[X := S]$ for substitutions and $\cup$ for set-theoretic union. Also we use $C \vdash a{:}\,S{\leq}T$ to denote that $C \vdash a{:}\,S$ and $C \vdash S{\leq}T$. Type substitutions are performed on indexes, too. Terms are selected by the rules below; since term variables are indexed by their type, the rules do not need assumptions of the form $(x{:}\,T)$:

[Vars] $$C \vdash x^T : T \qquad C \vdash T \text{ type}$$

[→INTRO] $$\frac{C \vdash a : T'}{C \vdash (\lambda x^T.a) : T \to T'} \qquad C \vdash T \text{ type}$$

[→ELIM] $$\frac{C \vdash a : T' \qquad C \vdash b : S' \leq S}{C \vdash a(b) : T} \qquad \mathcal{B}(T')_C = S \to T$$

[TOP] $$C \vdash \mathsf{top} : \mathsf{Top}$$

[∀INTRO] $$\frac{C \cup \{X \leq T\} \vdash a : T'}{C \vdash \Lambda X \leq T.a : \forall (X \leq T) T'} \qquad C \vdash T \text{ type}$$

[∀ELIM] $$\frac{C \vdash a : T' \qquad C \vdash S' \leq S}{C \vdash a(S') : T[X := S']} \qquad \mathcal{B}(T')_C = \forall (X \leq S) T$$

[ε] $$C \vdash \varepsilon : \forall X\{\}$$

[{}INTRO] $$\frac{C \vdash a : T_1 \leq \forall X\{A_i.T_i\}_{i \leq n} \quad C \vdash b : T_2 \leq \forall (X \leq A) T}{C \vdash (a \&^{[A_1.T_1 \| \ldots \| A_n.T_n \| A.T]} b) : \forall X(\{A_i.T_i\}_{i \leq n} \cup \{A.T\})}$$
$$C \vdash \forall X(\{A_i.T_i\}_{i \leq n} \cup \{A.T\}) \text{ type}$$

[{}ELIM] $$\frac{C \vdash a : T \qquad C \vdash A_j = min_{i \in I}\{A_i | C \vdash A \leq A_i\}}{C \vdash a[A] : T_j[X := A]} \qquad \mathcal{B}(T)_C = \forall X\{A_i.T_i\}_{i \in I}$$

Note the form of the premises in the rule [{}INTRO]; we cannot require that the components of an & must have the same type as the one specified in the index: since it is possible to reduce inside an & then the types of the components may decrease (see the subject reduction theorem 9.4.8) and cannot be fixed (the index does not change with the reduction).

A first non trivial result for this system is given by the following theorem.

Theorem 9.3.1 *If* $C \vdash a : T$ *then* $C \vdash T$ type

Proof. The proof is an easy induction on the depth of the proof of $C \vdash a : T$ by performing a case analysis on the last applied rule. The cases for [∀ELIM] and [{}ELIM] are solved by using the lemma 9.4.5. □

As the careful reader will have noticed, we do not use subsumption in the type checking; since the selection of a branch is done according to the type of the argument we want, to avoid ambiguities, to ensure that every well-typed term has a unique type. This is stated by the following theorem:

Theorem 9.3.2 *If* $C \vdash a : T_1$ *and* $C \vdash a : T_2$ *then* $T_1 \equiv T_2$

Proof. An easy induction on the sum of the depths of the derivation of $C \vdash a{:}T_1$ and $C \vdash a{:}T_2$, by performing a case analysis on the structure of a □

Thanks to the theorem 9.2.15 we can associate to every provable judgment a canonical derivation.

Theorem 9.3.3 *Let Π_1 and Π_2 be two derivations for the same judgment $C \vdash a{:}T$. Let $(\Pi_i)^*$ $(i = 1, 2)$ denote the derivation Π_i in which every (sub-)derivation of a subtyping judgment has been replaced by its canonical form. Then $\Pi_1 \equiv \Pi_2$.*

Proof. By induction on the structure of a (which uniquely determines the typing rule to apply). □

By combining the result of this two theorems we obtain that every well-typed term has a canonical derivation for its type.

Thus one would expect that it is possible to define a type-checking algorithm for the raw terms. This is the cases, indeed: if in the system above you replace every subtyping judgment $C \vdash S{\leq}T$ by $C \vdash_{\!\mathcal{A}} S{\leq}T$ you have a type-checking algorithm that can be easily proved sound and complete w.r.t. the original system.

9.4 Reduction

In this section we give the equational theory of the terms of $F^{\&}_{\leq}$. We present it under the form of reduction rules. We suppose to work modulo α-conversion for term variables; note that no clash is possible for type variables because of the definition of tcs.

Notions of reduction

(β) $C \vdash (\lambda x^T.a)(b) \ \triangleright a[x^T := b]$

$(\beta_\forall)$ $C \vdash (\Lambda X{\leq}T.a)(T') \ \triangleright a[X := T']$

$(\beta_{\{\}})$ If $A, A_1 \ldots, A_n$ are closed then

$$C \vdash (a\&^{[A_1.T_1\|\ldots\|A_n.T_n]}b)[A] \ \triangleright \begin{cases} b(A) & \text{if } A_n = min_{1\leq i\leq n}\{A_i | C \vdash A{\leq} A_i\} \\ a[A] & \text{else} \end{cases}$$

Note that the selection of the branch is made on the index. Therefore while overloaded types are equal modulo reordering of their components, in indexes the order is meaningful since to a different ordering may correspond a different selection.

Besides these rules there are the usual rules for the context; among these the only one that deserves a note is the rule for Λ, for it changes the tcs of the reduction:

$$\frac{C \cup \{X{\leq}T\} \vdash a \ \triangleright a'}{C \vdash (\Lambda X{\leq}T.a) \ \triangleright (\Lambda X{\leq}T.a')}$$

For what it concerns the rules note that in $\beta_{\{\}}$ we require that the types involved in the selection of a branch are closed. In this way we always select the most precise branch (i.e. the one with the smallest possible bound). This corresponds to the implementation of the *late binding*.

9.4.1 The encoding of records

In this section we show how to encode in $F_{\leq}^{\&}$ updatable records. Records will be used in section 10.1. We start with a variation of the records of Wand that we have encoded in section 2.5.3: the records we consider here are constructed starting from an empty record value, denoted by $\langle\,\rangle$, and by two elementary operations:

- *Consistent Overwriting* $\langle r \leftarrow \ell_i = a\rangle$; if ℓ_i is not present in r, then it adds a field of label ℓ_i and value a to the record r; otherwise replaces the value of the field with label ℓ_i by the value a, provided that a has the same type as the type of the value it replaces.

- *Extraction* $r.\ell_i$; extracts the value corresponding to the label ℓ_i, provided that a field having that label is present.

The difference with the original calculus of Wand is in the operation of overwriting where there is a restriction which requires the preservation of the type of a field being updated. This operation is called *consistent updating* in [Car92]. We encode these records as follows:

Definition 9.4.1 Let $L_1, L_2, \dots$ be an infinite list of basic types. Assume that they are isolated (i.e., for every type T, if $L_i \leq T$ or $T \leq L_i$, then $L_i = T$). Then set

$$\langle\!\langle \ell_1 : T_1, \dots, \ell_n : T_n \rangle\!\rangle \;=\; \forall X\{L_1.T_1, \dots, L_n.T_n\} \qquad X \notin \cup_{i=1..n} FV(T_i)$$

$$\langle\,\rangle \;\equiv\; \varepsilon$$

$$r.\ell_i \;\equiv\; r[L_i]$$

$$\langle r \leftarrow \ell_i = a\rangle \;\equiv\; (r \;\&^I \Lambda X{\leq}L_i.a) \qquad \text{where } I \equiv [L_1.T_1\|...\|L_n.T_n\|L_i.T_i] \text{ if } r : \forall X\{L_1.T_1, \dots, L_n.T_n\}$$

□

Both the conditions in *Overwriting* and *Extraction* are enforced statically by the encoding: for example if $a_i{:}T_i$ then the record

$$\langle\,\langle\,\langle\rangle \leftarrow \ell = a_1\rangle \leftarrow \ell = a_2\rangle \tag{9.1}$$

is encoded by

$$(\varepsilon \;\&^{[L.T_1]}\Lambda X{\leq}L.a_1 \;\&^{[L.T_1\|L.T_2]}\Lambda X{\leq}L.a_2)$$

of type $\forall X\{L.T_1, L.T_2\}$. But then by covariance one has $T_1 \lesseqgtr T_2$

Once more, the rules to subtype simple record types and to type record values are obtained as the special case of the encoding, i.e. when the input types are isolated. For example, it is easy to check that if L_i's are isolated then $\forall X\{L_i.T_i\}_{i\in I} \leq \forall X\{L_i.T'_i\}_{i\in J}$ if and only if $J \subseteq I$ and $\forall i{\in}J\; T_i \leq T'_i$.

To encode the original records of [Wan87] we have to proceed as in section 2.5.3 and introduce the following meta-notation

Notation 9.4.2 *Given an overloaded type* $T \equiv \forall X\{A_i.T_i\}_{i\in I}$ *we denote by* $T \setminus A_j$ *the type* $\forall X\{A_i.T_i\}_{i\in I\setminus\{j\}}$ *if* $A_j \in \{A_i\}_{i\in I}$*, and the type* T *itself, otherwise*

For example $\forall X\{A_1.T, A_2.T\}\setminus A_1 = \forall X\{A_2.T\}$ and $\forall X\{A_1.T, A_2.T\}\setminus A = \forall X\{A_1.T, A_2.T\}$. As in section 2.5.3, if T is a well-formed type $T \setminus A$ may be not well formed since A might be necessary to assure the $\cap$-closure of T. Though when we restrain our attention to overloaded types whose bounds form a set of isolated types this problem does not persist since $\cap$-closure is always trivially satisfied. Therefore we know that applying the metanotation $\setminus$ to an overloaded type encoding a record type we will always obtain a well formed type. Finally note that if $T \setminus A$ is a well formed type then $T \leq T \setminus A$.

To encode the full calculus of [Wan87] it suffices to modify the definition of $\langle r \leftarrow \ell_i = a\rangle$ in the following way:

$$\langle r \leftarrow \ell_i = a\rangle \quad = \quad (r \mathbin{\&}^I \Lambda X{\leq} L_i.a) \qquad \begin{array}{l}\text{where}\\ I{\equiv}[L_1.T_1\|...\|L_n.T_n\|L_i.T_i] \text{ if } r : T \text{ and}\\ T \setminus L_i = \forall X\{L_1.T_1, \dots, L_n.T_n\}\end{array}$$

Therefore the record in (9.1) is now encoded by:

$$(\varepsilon \mathbin{\&}^{[L.T_1]} \Lambda X{\leq} L.a_1 \mathbin{\&}^{[L.T_2]} \Lambda X{\leq} L.a_2)$$

of type $\forall X\{L.T_2\}$.

Unfortunately, as in the case of $\lambda\&$, the use of the indexes in the encoding precludes the polymorphism of the updating operation. In this framework this for example implies that we cannot have an operation of updating that returns a record whose type is a type variable, since we do not have variable indexes (just indexes containing variables). This could be an important lack: in object-oriented programming one wants that the following method

$$\texttt{init_x} = (\varepsilon \& \Lambda Mytype{\leq} \langle\!\langle x : Int\rangle\!\rangle.\lambda self^{Mytype}.\langle self \leftarrow x = 0\rangle)$$

has type

$$\forall Mytype\{\langle\!\langle x : Int\rangle\!\rangle.Mytype \to Mytype\}$$

while with the encodings above it has type

$$\forall Mytype\{\langle\!\langle x : Int\rangle\!\rangle.Mytype \to \langle\!\langle x : Int\rangle\!\rangle\}$$

Fortunately $F^{\&}_{\leq}$ includes $F_{\leq}$, thus we can use the encodings defined for this last one to obtain records types with the wanted properties. In particular we can use the encoding of extensible records defined in [Car92]. The calculus of record values encoded in that paper is less powerful than the records encoded above: for example it is not possible to add a field to a record that is not known to possess it. However it offers more polymorphism since the encoding of `init_x` would have the desired type.

In conclusion $F^{\&}_{\leq}$ offers a rather wide choice of record calculi, surely enough to cope with the modeling of object-oriented programming.

9.4.2 Generalized Subject Reduction

In this section we prove that the type-checking system of $F^{\&}_{\leq}$ well behaves w.r.t. the reduction rules. More precisely we prove that every (well-typed) term rewrites to another (well-typed) term, whose type is smaller than or equal to the type of the former. The proof of subject

reduction is very technical and complex. The crux of the problem is to prove that the property of $\cap$-closure is conserved under reductions, more precisely under (feasible) substitutions. For this reason we suggest the reader to skip at first reading the proofs of the three lemmas that follows.

We need first some notation:

Notation 9.4.3 *Let* $C\cup\{X\leq T\}$ *be a tcs. Define* $(C\cup\{X\leq T\})[Y:=S]$ *as* $(C[Y:=S]\cup\{X\leq T[Y:=S]\})$ *and* $\emptyset[X:=S]$ *as* $\emptyset$*. Let* $C\vdash\Delta$ *be a type judgment. Then* $C\vdash\Delta[X:=S]$ *is defined as* $C\vdash T[X:=S]$ type *if* $\Delta\equiv T$ type*, as* $C\vdash T_1[X:=S]\leq T_2[X:=S]$ *if* $\Delta\equiv T_1\leq T_2$*.*

The proof of subject reduction requires an assumption and three technical lemmas:

Assumption 9.4.4 *Recall that the proof of* $C\vdash\{A_i\}_{i=1..n}\cap$*-closed is indeed an appropriate set of proofs with final judgments of the form* $C\vdash A_h\leq A_k$ *proving the meet closure of* $\{A_i\}_{i=1..n}$*. In particular we suppose that this set contains* at least *one proof of* $C\vdash A_i\leq A_j$ *for every* i,j *in* $[1..n]$ *for which such a proof exists.*

Lemma 9.4.5 (main lemma) *If* $C\cup\{X\leq S\}\vdash\Delta$ *is a provable type judgment,* $X\notin FV(S')$ *and* $C[X:=S']\vdash S'\leq S$ *is also provable, then* $C[X:=S']\vdash\Delta[X:=S']$ *is provable, too.*

Before proving the lemma, we want clarify a point: indeed the reader may wonder why in this lemma, as well as in lemma 9.4.7, we used the tcs $C[X:=S']$ rather than C (note that $X\notin dom(C)$). Actually if you replace $C[X:=S']$ by C the theorem can no longer be proved, since at some points it is not possible to use the induction hypothesis (more precisely when you introduce a new variable in the tcs). The intuitive reason is that even if $C\cup\{X\leq S\}$ and $C[X:=S']$ are well-formed tcs's this does not imply the good formation of C. For example take $S'\equiv S\equiv B$ and $C\equiv\{Y\leq X\}$: C is not well-formed but $C[X:=S']\equiv Y\leq B$ and $C\cup\{X\leq S\}\equiv\{Y\leq X\}\cup\{X\leq B\}$ are well-formed.[8] We can now prove the lemma.

Proof. By induction on the depth of the proof of $C\cup\{X\leq S\}\vdash\Delta$. For depth=1 there are only two possible cases: $\Delta\equiv B$ type or $\Delta\equiv$ Top type. In both cases the result is trivially satisfied. For depth>1 we perform a case analysis on the last rule of the proof:

(refl) a straightforward use of the induction hypothesis

(trans) a straightforward use of the induction hypothesis

(taut) suppose that $\Delta\equiv Y\leq T$ then there are two possible subcases:

1. $Y\not\equiv X$: a straightforward use of the induction hypothesis
2. $Y\equiv X$: then the hypothesis gets $C\cup\{X\leq S\}\vdash X\leq S$; since $X\notin FV(S)$ the result reduces to $C[X:=S']\vdash S'\leq S$ which holds by hypothesis

(top) a straightforward use of the induction hypothesis

($\rightarrow$) a straightforward use of the induction hypothesis

[8]The order in tcs is not important

($\forall$) suppose that $\Delta \equiv \forall(Y{\leq}T_1)T_2 \leq \forall(Y{\leq}T'_1)T'_2$. Recall that $C[X{:=}S'] \vdash S'{\leq}S$. Thus by theorem 9.2.1 $C[X{:=}S'] \vdash S'$ type and therefore $FV(S') \subseteq dom(C[X{:=}S'])$. By hypothesis we have that both $C \cup \{Y \leq T'_1\} \cup \{X \leq S\}$ and $C[X{:=}S']$ are tcs's. Since $X \notin FV(S')$ then $FV(T'_1[X{:=}S']) = (FV(T'_1) \cup FV(S'))\backslash\{X\}$; thus also $C[X{:=}S'] \cup \{Y{\leq}T'_1[X{:=}S']\}$ is a tcs. Once this remark done, then the result follows by a straightforward use of the induction hypothesis.

($\{\}$) As the previous case.

(**Var**$_{type}$) suppose that $\Delta \equiv Y$ type. Then there are two possible subcases:

1. $Y \not\equiv X$: a straightforward use of the induction hypothesis
2. $Y \equiv X$: then the result reduces to $C[X{:=}S'] \vdash S'$ type which follows from $C[X{:=}S'] \vdash S'{\leq}S$ and theorem 9.2.1

($\rightarrow_{type}$) a straightforward use of the induction hypothesis

($\forall_{type}$) After having done the same remark as in the case ($\forall$) the thesis follows from a straightforward use of the induction hypothesis.

($\{\}_{type}$) This is the hard case. The pattern of the proof of this case is essentially the same as that of the case ($\forall$). The hard task is to prove that $C \cup \{X{\leq}S\} \vdash \{A_i\}_{i=1..n}\cap$-closed, $C[X{:=}S'] \vdash S'{\leq}S$ and the induction hypothesis imply $C[X{:=}S'] \vdash \{A_i[X{:=}S']\}_{i=1..n}\cap$-closed. This is equivalent to prove that whenever

$$\mathcal{B}(A_i[X{:=}S'])_{C[X:=S']} \Downarrow \mathcal{B}(A_j[X{:=}S'])_{C[X:=S']} \tag{9.2}$$

then there exists $h \in [1..n]$ such that

$$C[X{:=}S'] \vdash A_h[X{:=}S'] = A_i[X{:=}S'] \cap A_j[X{:=}S']$$

Suppose that (9.2) holds, and examine all the possible cases for A_i and A_j:

i. (**A_i and A_j basic**). Then $A_i[X{:=}S'] = A_i = \mathcal{B}(A_i[X{:=}S'])_{C[X:=S']} = \mathcal{B}(A_i)_{C\cup\{X\leq S\}}$ and the same for j. From the meet-closure of $\{A_i\}_{i=1..n}$ follows that there exists a basic type $A_h = A_h[X{:=}S'] = A_i \cap A_j = A_i[X{:=}S'] \cap A_j[X{:=}S']$ independently from the tcs we are taking into account.

ii. ($A_i{\equiv}A_j{\equiv}X$) trivial

iii. ($A_i{\equiv}X$ **and** $A_j \not\equiv X$). Then the hypothesis gets

$$\mathcal{B}(S')_{C[X:=S']} \Downarrow \mathcal{B}(A_j)_{C[X:=S']} \tag{9.3}$$

We proof the result by showing that $S' \cap A_j$ is always either S' or A_j.
From $C[X{:=}S'] \vdash S'{\leq}S$ and proposition 9.2.3 we deduce that $\mathcal{B}(S')_{C[X:=S']} \leq \mathcal{B}(S)_{C[X:=S']}$ and then from (9.3) follows that

$$\mathcal{B}(S)_{C[X:=S']} \Downarrow \mathcal{B}(A_j)_{C[X:=S']} \tag{9.4}$$

Now, first of all note that by definition of $\mathcal{B}$ one has $\mathcal{B}(X)_{C\cup\{X\leq S\}}=\mathcal{B}(S)_{C\cup\{X\leq S\}}$. Then observe that $\mathcal{B}(S)_{C\cup\{X\leq S\}} = \mathcal{B}(S)_{C[X:=S']}$: this is obvious if S is a basic type; when S is a variable this follows from the fact that the substitution $[X:=S']$ does not affect the definition of $\mathcal{B}(S)$. Indeed if

$$C[X:=S'] \equiv C' \cup \{S\leq X_1\} \cup \{X_1\leq X_2\} \cup \ldots \cup \{X_n\leq \mathcal{B}(S)\} \qquad (n \geq 0)$$

then $X \not\equiv X_i$ for all $i \in [1..n]$ otherwise $C \cup \{X\leq S\}$ would not be a tcs.
Thus from $\mathcal{B}(X)_{C\cup\{X\leq S\}}=\mathcal{B}(S)_{C\cup\{X\leq S\}}$ and $\mathcal{B}(S)_{C\cup\{X\leq S\}} = \mathcal{B}(S)_{C[X:=S']}$ we deduce

$$\mathcal{B}(X)_{C\cup\{X\leq S\}} = \mathcal{B}(S)_{C[X:=S']} \tag{9.5}$$

Now there are two possible subcases:

a. A_j is a basic type: then $\mathcal{B}(A_j)_{C\cup\{X\leq S\}} = A_j = \mathcal{B}(A_j)_{C[X:=S']}$ and thus (9.4) gets

$$\mathcal{B}(X)_{C\cup\{X\leq S\}} \Downarrow \mathcal{B}(A_j)_{C\cup\{X\leq S\}}$$

but since $C \cup \{X\leq S\} \vdash \{A_i\}_{i=1..n}\cap$-closed (and $X \in \{A_i\}_{i=1..n}$) we have that $C \cup \{X\leq S\} \vdash X\leq A_j$ (by proposition 9.2.5 the variable must be smaller than the basic type) and therefore $\mathcal{B}(S)_{C[X:=S']} = \mathcal{B}(X)_{C\cup\{X\leq S\}} \leq \mathcal{B}(A_j)_{C\cup\{X\leq S\}} = A_j$.
Thus $C[X:=S'] \vdash S'\leq S\leq\mathcal{B}(S)_{C[X:=S']}\leq A_j$ whence we can conclude that

$$C[X:=S'] \vdash S' = S' \cap A_j \tag{9.6}$$

b. A_j is a variable: then we have that

$$C[X:=S'] \equiv C'' \cup \{A_j\leq X_1\} \cup \{X_1\leq X_2\} \cup \ldots \cup \{X_n\leq \mathcal{B}(A_j)\} \qquad (n \geq 0)$$

If $S' \equiv X_i$ for some $i \in [1..n]$ then $C[X:=S'] \vdash A_j \leq S'$ and therefore

$$C[X:=S'] \vdash A_j = S' \cap A_j \tag{9.7}$$

Otherwise if $S' \not\equiv X_i$ for all $i \in [1..n]$ then the substitution $[X:=S']$ does not affect the definition of $\mathcal{B}(A_j)$ and thus

$$\mathcal{B}(A_j)_{C\cup\{X\leq S\}} = \mathcal{B}(A_j)_{C[X:=S']}$$

Thus once more (9.4) and (9.5) yield

$$\mathcal{B}(X)_{C\cup\{X\leq S\}} \Downarrow \mathcal{B}(A_j)_{C\cup\{X\leq S\}}$$

Recall that both X and A_j are variables contained in $\{A_i\}_{i=1..n}$ and that $C \cup \{X\leq S\} \vdash \{A_i\}_{i=1..n}\cap$-closed. Thus by proposition 9.2.5 either $C \cup \{X\leq S\} \vdash X \leq A_j$ or $C \cup \{X\leq S\} \vdash A_j \leq X$ must hold. Whichever judgment holds, we supposed in the assumption 9.4.4 that its proof is contained in the proof of meet closure of $\{A_i\}_{i=1..n}$; thus we can apply the induction hypothesis obtaining either (9.6) or (9.7), respectively.

iv. **(A_j and A_j are both different from X and at least one of them is a variable)** Thus $A_i[X{:=}S'] = A_i$ and $A_j[X{:=}S'] = A_j$ and the hypothesis becomes

$$\mathcal{B}(A_i)_{C[X:=S']} \Downarrow \mathcal{B}(A_j)_{C[X:=S']}$$

Let us open a short parenthesis: suppose to have a type variable $Y \not\equiv X$ with $Y \in dom(C)$ and consider $\mathcal{B}(Y)_{C\cup\{X\leq S\}}$. Then if
$C \cup \{X{\leq}S\} \equiv C' \cup \{Y{\leq}X_1\} \cup \{X_1{\leq}X_2\} \cup \ldots \cup \{X_n{\leq}\mathcal{B}(Y)_{C\cup\{X\leq S\}}\} \qquad (n \geq 0)$

there are two possible cases

(1) $X \equiv X_h$ for some $h \in [1..n]$ and in this case note that
$$\mathcal{B}(Y)_{C[X:=S']} = \mathcal{B}(S')_{C[X:=S']}$$
(2) $X \not\equiv X_h$ for all $h \in [1..n]$ and in this case
$$\mathcal{B}(Y)_{C[X:=S']} = \mathcal{B}(Y)_{C\cup\{X\leq S\}}$$

After this short remark we can now consider the various cases for A_i and A_j

a. A_i is a variable in the situation like Y in **(1)** and A_j is a basic type. But then by the point **(1)** the hypothesis becomes
$$\mathcal{B}(S')_{C[X:=S']} \Downarrow \mathcal{B}(A_j)_{C[X:=S']}$$
which has already been solved in (iii).

b. A_i is a variable in a situation like Y in **(2)** and A_j is a basic type. By the meet-closure of $\{A_i\}_{i=1..n}$ and by the point **(2)** we deduce that
$$\mathcal{B}(A_i)_{C[X:=S']} = \mathcal{B}(A_i)_{C\cup\{X\leq S\}} \leq A_j = \mathcal{B}(A_j)_{C\cup\{X\leq S\}}$$
and thus $C[X{:=}S'] \vdash A_i \leq A_j$

c. A_i is a variable in the situation like in **(1)** and A_j is a variable in the situation like in **(2)**; but then we are in a case similar to the one of (a.)

d. A_i and A_j are both variables in the situation like in **(1)**. Then $\mathcal{B}(A_i)_{C[X:=S']} = \mathcal{B}(S')_{C[X:=S']} = \mathcal{B}(A_j)_{C[X:=S']}$. Thus either $C[X{:=}S'] \vdash A_i \leq A_j$ or $C[X{:=}S'] \vdash A_j \leq A_i$ holds.

e. A_i and A_j are both variables in the situation like in **(2)**. Thus $\mathcal{B}(A_i)_{C\cup\{X\leq S\}} \Downarrow \mathcal{B}(A_j)_{C\cup\{X\leq S\}}$ and by the meet-closure either $C \cup \{X{\leq}S\} \vdash A_i \leq A_j$ or $C \cup \{X{\leq}S\} \vdash A_j \leq A_i$ holds. But since they are variables like in **(2)** this come to say that either $C[X{:=}S'] \vdash A_i \leq A_j$ or $C[X{:=}S'] \vdash A_j \leq A_i$ holds.

□

Lemma 9.4.6 (term substitution) *If $C \vdash b{:}T' \leq T$ and $C \vdash a{:}S$ then $C \vdash a[x^T{:=}b] : S'{\leq}S$.*

Proof. By induction on the structure of a:

$a{\equiv}y$ if $y \equiv x$ then $S{\equiv}T$ and $S'{\equiv}T'$; else if $y \not\equiv x$ the result trivially holds.

$a{\equiv}\varepsilon$ trivial

$a{\equiv}\mathsf{Top}$ trivial

$a\equiv\lambda y^{S_1}.a'$ if $y\equiv x$ then the result trivially holds; otherwise $S\equiv S_1\to S_2$ and $C\vdash a'{:}\,S_2$. By induction hypothesis $C\vdash a'[x^T{:}=b] : S_2'\leq S$ thus

$$C\vdash a[x^T{:}=b]\equiv\lambda y^{S_1}.a'[x^T{:}=b] : S_1\to S_2'\leq S_1\to S_2$$

$a\equiv(a_1\&^I a_2)$ just note that by induction hypothesis $(a_1[x^T{:}=b]\&^I a_2[x^T{:}=b])$ is well-typed, and that its type is S.

$a\equiv\Lambda X{\leq}S_1.a'$ then $C\cup\{X{\leq}S_1\}\vdash a'{:}\,S_2$ with $S\equiv\forall(X{\leq}S_1)S_2$. By induction hypothesis $C\cup\{X{\leq}S_1\}\vdash a'[x^T{:}=b] : S_2'\leq S_2$. Thus

$$C\vdash a[x^T{:}=b]\equiv\Lambda X{\leq}S_1.a'[x^T{:}=b] : \forall(X{\leq}S_1)S_2'\leq\forall(X{\leq}S_1)S_2$$

$a\equiv a_1(a_2)$ then $C\vdash a_1{:}\,S_3$, $\mathcal{B}(S_3)_C=S_1\to S$ and $C\vdash a_2{:}\,S_2{\leq}S_1$. By induction hypothesis $C\vdash a_1[x^T{:}=b] : U_3{\leq}S_3$ and $C\vdash a_2[x^T{:}=b] : U_2{\leq}S_2{\leq}S_1$. By proposition 9.2.3 $C\vdash\mathcal{B}(U_3)_C\leq\mathcal{B}(S_3)_C$. Since $\mathcal{B}(U_3)_C$ is not a type variable then by proposition 9.2.2 it is of the form $U_1\to U$ with $C\vdash S_1{\leq}U_1$ and $C\vdash U{\leq}S$. Thus we have:
- $C\vdash a_1[x^T{:}=b] : U_3$
- $C\vdash a_3[x^T{:}=b] : U_2{\leq}U_1$
- $\mathcal{B}(U_3)_C=U_1\to U$

Then by $[\to\text{ELIM}_{(\leq)}]$ we obtain

$$C\vdash a[x^T{:}=b]\equiv a_1[x^T{:}=b](a_2[x^T{:}=b]) : U{\leq}S$$

$a\equiv a'(U)$ then $C\vdash a'{:}\,S_3$, $\mathcal{B}(S_3)_C=\forall(X{\leq}S_1)S_2$, $C\vdash U{\leq}S_1$ and $S\equiv S_2[X{:}=U]$. Note that $X\notin dom(C)$ and thus $X\notin FV(U)$. By induction hypothesis $C\vdash a'[x^T{:}=b] : U_3{\leq}S_3$ and by proposition 9.2.3 $C\vdash\mathcal{B}(U_3)_C\leq\mathcal{B}(S_3)_C$. Since $\mathcal{B}(U_3)_C$ is not a type variable then by proposition 9.2.2 it is of the form $\forall(X{\leq}S_1')S_2'$.
Since $C\cup\{X{\leq}S_1\}\vdash S_2'\leq S_2$, $C\vdash U{\leq}S_1{\leq}S_1'$ and $X\notin FV(U)$ we can apply the main lemma and obtain

$$C[X:=U]\vdash S_2'[X:=U]\leq S_2[X:=U]$$

But $X\notin dom(C)$ thus $C[X:=U]=C$, from which it follows

$$C\vdash a[x^T{:}=b] : S_2'[X:=U]\leq S_2[X:=U]$$

$a\equiv a'[A]$ then $C\vdash a'{:}\,S_3$, $\mathcal{B}(S_3)_C=\forall X\{A_i.T_i\}_{i\in I}$ and $S\equiv T_h[X{:}=A]$ where $C\vdash A_h=\min_{i\in I}\{A_i\,|\,C\vdash A{\leq}A_i\}$.
By induction hypothesis $C\vdash a'[x^T{:}=b] : U_3\leq S_3$ and by proposition 9.2.3 $C\vdash\mathcal{B}(U_3)_C\leq\mathcal{B}(S_3)_C$. Since $\mathcal{B}(U_3)_C$ is not a type variable then it is of the form $\forall X\{A_j'.T_j'\}_{j\in J}$. Thus by the subtyping rule ($\{\}$) there exists $\tilde{h}\in J$ such that $C\vdash A\leq A_h\leq A_{\tilde{h}}'$. Therefore the set $\{A_j'\,|\,C\vdash A\leq A_j', j\in J\}$ is not empty, and by the meet-closure of $\{A_j'\}_{j\in J}$ it has also a minimum. Call this minimum A_k'. Then $C\vdash a'[x^T{:}=b] : T_k'[X{:}=A]$. Since $S{\equiv}T_h[X{:}=A]$ we have to prove that

$$C\vdash T_k'[X{:}=A]\leq T_h[X{:}=A]$$

Take again the previous $\tilde{h}$; by the rule ($\{\}$) we have

$$C\vdash\forall(X{\leq}A_{\tilde{h}}')T_{\tilde{h}}'\leq\forall(X{\leq}A_h)T_h \tag{9.8}$$

By the definition of A_h:

$$C \vdash A \leq A_h \tag{9.9}$$

From (9.8):

$$C \vdash A_h \leq A'_{\tilde{h}}$$

From (trans):

$$C \vdash A \leq A'_{\tilde{h}} \tag{9.10}$$

From (9.8):

$$C \cup \{X \leq A_h\} \vdash T'_{\tilde{h}} \leq T_h \tag{9.11}$$

From the definition of A'_k and from (9.10) we obtain

$$C \vdash A'_k \leq A'_{\tilde{h}}$$

and from this and the rule $(\{\}_{type})$ applied to $\forall X\{A'_j.T'_j\}_{j\in J}$ it follows

$$C \cup \{X \leq A_h\} \vdash T'_k \leq T'_{\tilde{h}} \tag{9.12}$$

From $X \notin dom(C)$ and from (9.9) we deduce that $X \notin FV(A)$; by (9.9) and by the choice of k we respectively have that $C \vdash A \leq A_h$ and $C \vdash A \leq A'_k$; thus we can apply the main lemma to (9.11) and (9.12) to obtain:

$$C[X{:=}A] \vdash T'_{\tilde{h}}[X{:=}A] \leq T_h[X{:=}A]$$
$$C[X{:=}A] \vdash T'_k[X{:=}A] \leq T'_{\tilde{h}}[X{:=}A]$$

But $X \notin dom(C)$, thus the judgements above get

$$C \vdash T'_{\tilde{h}}[X{:=}A] \leq T_h[X{:=}A]$$
$$C \vdash T'_k[X{:=}A] \leq T'_{\tilde{h}}[X{:=}A]$$

Finally by (trans) we obtain the result:

$$C \vdash T'_k[X{:=}A] \leq T_h[X{:=}A]$$

□

Lemma 9.4.7 (type substitution) *If $C \cup \{X{\leq}S\} \vdash a{:}T$, $C[X{:=}S'] \vdash S' \leq S$ and $X \notin FV(S')$ then $C[X{:=}S'] \vdash a[X{:=}S']{:}T' \leq T[X{:=}S']$*

Proof. By induction on the structure of a:

$a{\equiv}x^T$ then $T' \equiv T[X{:=}S']$.

$a{\equiv}\varepsilon$ trivial

$a{\equiv}\mathsf{Top}$ trivial

$a{\equiv}\lambda x^{T_1}.a'$ where $T{\equiv}T_1 \to T_2$ and $C \cup \{X{\leq}S\} \vdash a'{:}T_2$. Thus by induction hypothesis we deduce that $C[X{:=}S'] \vdash a'[X{:=}S'] : T'_2{\leq}T_2[X{:=}S']$. Therefore $C[X{:=}S'] \vdash a[X{:=}S'] = \lambda x^{T_1[X:=S']}.a'[X{:=}S'] : T_1[X{:=}S'] \to T'_2{\leq}T[X{:=}S']$.

$a \equiv \Lambda Y{\leq}T_1.a'$ First of all note that $Y \not\equiv X$, since by hypothesis $C \cup \{X{\leq}S\} \vdash \Lambda Y{\leq}T_1.a'{:}T$ and we have made the assumption of having all the type variables different in a tcs. Thus $C \cup \{X{\leq}S\} \cup \{Y{\leq}T_1\} \vdash a'{:}T_2$ and $T \equiv \forall(Y{\leq}T_1)T_2$. Note that $C[X{:=}S']$ and $C \cup \{X{\leq}S\}$ are tcs's, and also that $FV(S') \subseteq C[X{:=}S']$ (since $C[X{:=}S'] \vdash S'{\leq}S$). Thus we can conclude that also $C[X{:=}S'] \cup \{Y{\leq}T_1[X{:=}S']\}$ is a tcs, being $dom(C) = dom(C[X{:=}S'])$ and $FV(T_1[X{:=}S']) = (FV(T_1) \cup FV(S')) \backslash \{X\}$ (the latter because $X \notin FV(S)$). Then by a weakening we can prove that $(C \cup \{Y{\leq}T_1\})[X{:=}S'] \vdash S'{\leq}S$. By induction hypothesis thus we have $C[X{:=}S'] \cup \{Y{\leq}T_1[X{:=}S']\} \vdash a'[X{:=}S']{:}T_2' \leq T_2[X{:=}S']$. Thus by [$\forall$INTRO] and ($\forall$) we have that

$$C[X{:=}S'] \vdash \Lambda Y{\leq}T_1[X{:=}S'].a'[X{:=}S']{:}\forall(Y{\leq}T_1[X{:=}S'])T_2' \leq T[X{:=}S']$$

$a \equiv (a_1 \&^{[A_1.T_1\|\ldots\|A_n.T_n]} a_2)$ Thus $T \equiv \forall Y\{A_1.T_1, \ldots, A_n.T_n\}$, $C \cup \{X{\leq}S\} \vdash a_1 : S_1{\leq}\forall Y\{A_i.T_i\}_{i=1..n-1}$ and $C \cup \{X{\leq}S\} \vdash a_2{:}\,S_2{\leq}\forall(Y{\leq}A_n)T_n$. Since $C \cup \{X{\leq}S\} \vdash S'{\leq}S$ we can apply the main lemma (9.4.5) to the two judgements above obtaining respectively

$$C[X{:=}S'] \vdash S_1[X{:=}S']{\leq}\forall Y\{A_i[X{:=}S'].T_i[X{:=}S']\}_{i=1..n-1}$$
$$C[X{:=}S'] \vdash S_2[X{:=}S']{\leq}\forall(Y{\leq}A_n[X{:=}S'])(T_n[X{:=}S'])$$

Furthermore by induction hypothesis

$$C[X{:=}S'] \vdash a_i[X{:=}S'] : S_i' \leq S_i[X{:=}S'] \qquad i = 1, 2$$

Recall that by definition

$$a[X{:=}S'] = (a_1[X{:=}S'] \&^{[A_1[X:=S'].T_1[X:=S']\|\ldots\|A_n[X:=S'].T_n[X:=S']]} a_2[X{:=}S'])$$

Therefore using transitivity and the rule [{}INTRO] we can conclude that

$$C[X{:=}S'] \vdash a[X{:=}S'] : \forall Y\{A_i[X{:=}S'].T_i[X{:=}S']\}_{i=1..n} = T[X{:=}S']$$

$a \equiv a_1(a_2)$ Let $C \cup \{X{\leq}S\} \vdash a_1{:}W$, $C \cup \{X{\leq}S\} \vdash a_2{:}U'{\leq}U$ and $\mathcal{B}(W)_{C \cup \{X \leq S\}} = U \to T$. By induction hypothesis we have:

$$C[X{:=}S'] \vdash a_1[X{:=}S'] : W' \leq W[X{:=}S']$$
$$C[X{:=}S'] \vdash a_2[X{:=}S'] : U'' \leq U'[X{:=}S']$$

Applying the main lemma (9.4.5) to $C \cup \{X{\leq}S\} \vdash U'{\leq}U$ and (trans) we obtain

$$C[X{:=}S'] \vdash U'' \leq U[X{:=}S']$$

By proposition 9.2.3

$$C[X{:=}S'] \vdash \mathcal{B}(W')_{C[X:=S']} \leq \mathcal{B}(W[X{:=}S'])_{C[X:=S']} \tag{9.13}$$

Set $\overline{W} \equiv W[X{:=}S']$. We want to prove that

$$C[X{:=}S'] \vdash \mathcal{B}(\overline{W})_{C[X:=S']} \leq \mathcal{B}(W)_{C \cup \{X \leq S\}}[X{:=}S'] \tag{9.14}$$

If $\overline{W}$ is not a variable this follows from (refl). Otherwise let

$$C \cup \{X{\leq}S\} \equiv C' \cup \{\overline{W}{\leq}X_1\} \cup \{X_1{\leq}X_2\} \cup \ldots \cup \{X_n{\leq}\mathcal{B}(\overline{W})_{C \cup \{X \leq S\}}\} \qquad (n \geq 0)$$

There are two subcases:

1. $X \not\equiv X_i$ for all $i \in [1..n]$; then $\mathcal{B}(\overline{W})_{C\cup\{X\leq S\}} = \mathcal{B}(\overline{W})_{C[X:=S']}$
2. $X \equiv X_i$ for some $i \in [1..n]$; then

$$\mathcal{B}(\overline{W})_{C[X:=S']} = \mathcal{B}(S')_{C[X:=S']}$$
$$\mathcal{B}(\overline{W})_{C\cup\{X\leq S\}} = \mathcal{B}(S)_{C\cup\{X\leq S\}}$$

Now it is easy to check that $\mathcal{B}(S')_{C[X:=S']} = \mathcal{B}(S')_{C\cup\{X\leq S\}}$ (otherwise $C[X:=S']$ and $C \cup \{X \leq S\}$ could not both satisfy the conditions of tcs). Thus by proposition 9.2.3 we obtain

$C\cup\{X\leq S\} \vdash \mathcal{B}(\overline{W})_{C[X:=S']} = \mathcal{B}(S')_{C[X:=S']} = \mathcal{B}(S')_{C\cup\{X\leq S\}} \leq \mathcal{B}(S)_{C\cup\{X\leq S\}} = \mathcal{B}(\overline{W})_{C\cup\{X\leq S\}}$

Thus in both cases we have that

$$C\cup\{X\leq S\} \vdash \mathcal{B}(\overline{W})_{C[X:=S']} \leq \mathcal{B}(\overline{W})_{C\cup\{X\leq S\}}$$

We can then apply the main lemma and obtain

$$C[X:=S'] \vdash \mathcal{B}(\overline{W})_{C[X:=S']}[X:=S'] \leq \mathcal{B}(\overline{W})_{C\cup\{X\leq S\}}[X:=S']$$

By hypothesis $X \notin FV(S')$; this implies that $X \notin FV(C[X:=S'])$ and thus $\mathcal{B}(\overline{W})_{C[X:=S']}\,[X:=S'] = \mathcal{B}(\overline{W})_{C[X:=S']}$. Therefore to conclude the proof of (9.14) it just remains to prove the following equation:

$$C[X:=S'] \vdash \mathcal{B}(\overline{W})_{C\cup\{X\leq S\}}[X:=S'] \leq \mathcal{B}(W)_{C\cup\{X\leq S\}}[X:=S'] \tag{9.15}$$

This is obvious if W is not a variable (since $X \notin FV(S')$ then the substitution $[X:=S']$ is idempotent) or if it is a variable different from X (then $\overline{W} = W$). If $W{\equiv}X$ then just note that (9.15) gets

$$C[X:=S'] \vdash \mathcal{B}(S')_{C\cup\{X\leq S\}}[X:=S'] \leq \mathcal{B}(X)_{C\cup\{X\leq S\}}[X:=S']$$

by observing that $\mathcal{B}(X)_{C\cup\{X\leq S\}} = \mathcal{B}(S)_{C\cup\{X\leq S\}}$ this judgments becomes:

$$C[X:=S'] \vdash \mathcal{B}(S')_{C\cup\{X\leq S\}}[X:=S'] \leq \mathcal{B}(S)_{C\cup\{X\leq S\}}[X:=S'] \tag{9.16}$$

To prove it first apply proposition 9.2.3 to the hypothesis $C[X:=S'] \vdash S'{\leq}S$ and obtain

$$C[X:=S'] \vdash \mathcal{B}(S')_{C[X:=S']} \leq \mathcal{B}(S)_{C[X:=S']} \tag{9.17}$$

Assume now that we have proved that $\mathcal{B}(S')_{C[X:=S']} = \mathcal{B}(S')_{C\cup\{X\leq S\}}[X:=S']$ and $\mathcal{B}(S)_{C[X:=S']} = \mathcal{B}(S)_{C\cup\{X\leq S\}}[X:=S']$. In this case (9.17) implies (9.16). So let us prove the assumption: we start with S'. When S' is not a type variable then the result follows from the definition of $\mathcal{B}$ and the fact that $X \notin FV(S')$. If S' is a type variable then

$$C[X:=S'] \equiv C' \cup \{S'{\leq}X_1\} \cup \ldots \cup \{X_n{\leq}\mathcal{B}(S')_{C[X:=S']}\}$$

Since $C\cup\{X{\leq}S\}$ is a tcs then $X \not\equiv X_i$ for all $i \in [1..n]$. By this

$$C\cup\{X{\leq}S\} \equiv C'' \cup \{S'{\leq}X_1\} \cup \ldots \cup \{X_n{\leq}T''\} \cup \{X{\leq}S\}$$

Note that T'' cannot be a type variable: it cannot be X otherwise $\mathcal{B}(S')_{C[X:=S']} = S'$ (a loop in a tcs); it cannot be another variable otherwise $C[X:=S'](X_n)$ would be a

variable, too. Therefore T'' is not a type variable which implies that $T'' = \mathcal{B}(S')_{C\cup\{X\leq S\}}$ and thus $\mathcal{B}(S')_{C[X:=S']} = \mathcal{B}(S')_{C\cup\{X\leq S\}}[X{:}= S']$. A similar proof holds for S, too.

This ends the proof of (9.14)

From (9.13) and (9.14) we obtain:

$$C[X{:}= S'] \vdash \mathcal{B}(W')_{C[X:=S']} \leq U[X{:}= S'] \rightarrow T[X{:}= S']$$

Since $\mathcal{B}(W')_{C[X:=S']}$ is not a variable then it must be of the form $U''' \rightarrow T'$ with $C[X{:}= S'] \vdash U[X{:}= S'] \leq U'''$ and $C[X{:}= S'] \vdash T' \leq T[X{:}= S']$.
Summing up we have:
- $C[X{:}= S'] \vdash a_1[X{:}= S'] : W'$
- $C[X{:}= S'] \vdash a_2[X{:}= S'] : U''{\leq}U'''$
- $\mathcal{B}(W')_{C[X:=S']} = U''' \rightarrow T'$

Then by[$\rightarrow$ELIM$_{(\leq)}$] we obtain

$$C[X{:}= S'] \vdash a[X{:}= S'] \equiv a_1[X{:}= S'](a_2[X{:}= S']) : T' \leq T[X{:}= S']$$

$a{\equiv}a'(U)$ Let $C \cup \{X{\leq}S\} \vdash a' : W$, $C \cup \{X{\leq}S\} \vdash U \leq U'$, $\mathcal{B}(W)_{C\cup\{X\leq S\}} = \forall(Y \leq U')U''$ and $T \equiv U''[Y{:}= U]$. First of all note that $Y \notin dom(C) \cup \{X\}$; then by induction hypothesis

$$C[X{:}= S'] \vdash a'[X{:}= S'] : W' \leq W[X{:}= S']$$

By the main lemma we have that

$$C[X{:}= S'] \vdash U[X{:}= S'] \leq U'[X{:}= S'] \tag{9.18}$$

Proceeding exactly as in the previous case we can prove that

$$C[X{:}= S'] \vdash \mathcal{B}(W')_{C[X:=S']} \leq \mathcal{B}(W)_{C\cup\{X\leq S\}}[X{:}= S']$$

Since $\mathcal{B}(W')_{C[X:=S']}$ is not a variable then it is of the form $\forall(Y{\leq}V')V''$ with

$$C[X{:}= S'] \vdash U'[X{:}= S'] \leq V'$$

$$C[X{:}= S'] \cup \{Y{\leq}U'[X{:}= S']\} \vdash V'' \leq U''[X{:}= S'] \tag{9.19}$$

Thus we have:
- $C[X{:}= S'] \vdash a'[X{:}= S'] : W'$
- $C[X{:}= S'] \vdash U'[X{:}= S'] \leq V'$
- $\mathcal{B}(W')_{C[X:=S']} = \forall(Y{\leq}V')V''$

Therefore by [$\forall$ELIM] we obtain:

$$C[X{:}= S'] \vdash a[X{:}= S'] = a'[X{:}= S'](U[X{:}= S']) : V''[Y{:}= U[X{:}= S']]$$

Now from the hypothesis $C[X{:}= S'] \vdash S'{\leq}S$ and from $Y \notin dom(C)$ we deduce that $Y \notin FV(S')$; from $C \cup \{X{\leq}S\} \vdash U \leq U'$ and $Y \notin (dom(C) \cup \{X\})$ we deduce that $Y \notin FV(U)$. Thanks to this and to (9.18) we can apply the main lemma to (9.19) and obtain

$$C[X{:}= S'][[Y{:}= U[X{:}= S']] \vdash V''[Y{:}= U[X{:}= S']] \leq U''[X{:}= S'][Y{:}= U[X{:}= S']] \tag{9.20}$$

Since $Y \notin FV(S')$ then

$$([X{:=}S'][Y{:=}U[X{:=}S']]) = ([Y{:=}U][X{:=}S'])$$

and then (9.20) rewrites to

$$C[Y{:=}U][X{:=}S'] \vdash V''[Y{:=}U[X{:=}S']] \leq U''[Y{:=}U][X{:=}S'] = T[X{:=}S']$$

and since $Y \notin dom(C)$ it becomes

$$C[X{:=}S'] \vdash V''[Y{:=}U[X{:=}S']] \leq T[X{:=}S']$$

i.e. the result.

$a \equiv a'[A]$ Let $C \cup \{X{\leq}S\} \vdash a' : W$, $C \cup \{X{\leq}S\} \vdash A_h = \min_{i\in I}\{A_i \,|\, C \cup \{X{\leq}S\} \vdash A{\leq}A_i\}$, $\mathcal{B}(W)_{C\cup\{X\leq S\}} = \forall Y\{A_i.T_i\}_{i\in I}$ and $T \equiv T_h[Y{:=}A]$. Again $Y \notin dom(C) \cup \{X\}$. By induction hypothesis

$$C[X{:=}S'] \vdash a'[X{:=}S'] : W' \leq W[X{:=}S']$$

Applying the main lemma we also obtain that

$$C[X{:=}S'] \vdash \min_{i\in I}\{A_i[X{:=}S'] \,|\, C[X{:=}S'] \vdash A[X{:=}S'] {\leq} A_i[X{:=}S']\} \leq A_h[X{:=}S']$$

Proceeding as in the two previous cases we have that

$$C[X{:=}S'] \vdash \mathcal{B}(W')_{C[X:=S']} = \forall Y\{A'_j.T'_j\}_{j\in J} \leq \forall Y\{A_i[X{:=}S'].T_i[X{:=}S']\}_{i\in I} \quad (9.21)$$

By the rule ($\{\}$) for each $i \in I$ there exists $j \in J$ such that $C[X{:=}S'] \vdash A_i[X{:=}S'] \leq A'_j$. Thus by the main lemma we have that $\{A'_j \,|\, C[X{:=}S'] \vdash A[X{:=}S'] \leq A'_j, j \in J\}$ is not empty, and by the meet-closure of $\{A'_j\}_{j\in J}$ it has also a minimum. Therefore if

$$C[X{:=}S'] \vdash A'_k = \min_{j\in J}\{A'_j \,|\, C[X{:=}S'] \vdash A[X{:=}S'] \leq A'_j\}$$

then $C[X{:=}S'] \vdash a[X{:=}S'] : T'_k[Y{:=}A[X{:=}S']]$. Consider now (9.21); by the rule ($\{\}$) one has that there exists $\tilde{h} \in J$ such that

$$C[X{:=}S'] \vdash \forall(Y{\leq}A'_{\tilde{h}})T'_{\tilde{h}} \leq \forall(Y{\leq}A_h[X{:=}S'])(T_h[X{:=}S']) \qquad (9.22)$$

Since $C \cup \{X{\leq}S\} \vdash A \leq A_h$ then by the main lemma

$$C[X{:=}S'] \vdash A[X{:=}S'] \leq A_h[X{:=}S'] \qquad (9.23)$$

From (9.22):

$$C[X{:=}S'] \vdash A_h[X{:=}S'] \leq A'_{\tilde{h}}$$

From (trans):

$$C[X{:=}S'] \vdash A[X{:=}S'] \leq A'_{\tilde{h}} \qquad (9.24)$$

From (9.22):

$$C[X{:=}S'] \cup \{Y \leq A_h[X{:=}S']\} \vdash T'_{\tilde{h}} \leq T_h[X{:=}S'] \qquad (9.25)$$

From the definition of A'_k and from (9.24) we obtain

$$C[X{:=}S'] \vdash A'_k \leq A'_{\tilde{h}}$$

and from this and the rule $(\{\}_{type})$ applied to $\forall Y\{A'_j.T'_j\}_{j\in J}$ it follows that

$$C[X{:=}S']\cup\{Y\leq A'_k\}\vdash T'_k\leq T'_{\tilde h} \qquad (9.26)$$

From (9.23) and $Y\notin dom(C)$ (and thus $Y\notin dom(C[X{:=}S'])$) follows that $Y\notin FV(A[X{:=}S'])$; by (9.23) and by the choice of k we respectively have that $C[X{:=}S']\vdash A[X{:=}S']\leq A_h[X{:=}S']$ and $C[X{:=}S']\vdash A[X{:=}S']\leq A'_k$; thus we can apply the main lemma to (9.25) and (9.26) to obtain:

$$C[X{:=}S'][Y{:=}A[X{:=}S']]\vdash T'_{\tilde h}[Y{:=}A[X{:=}S']]\leq T_h[X{:=}S'][Y{:=}A[X{:=}S']]$$
$$C[X{:=}S'][Y{:=}A[X{:=}S']]\vdash T'_k[Y{:=}A[X{:=}S']]\leq T'_{\tilde h}[Y{:=}A[X{:=}S']]$$

But $Y\notin dom(C)$, thus $Y\notin dom(C[X{:=}S'])$ and whence, by the definition of tcs, $Y\notin FV(C[X{:=}S'])$. Then the judgements above get

$$C[X{:=}S']\vdash T'_{\tilde h}[Y{:=}A[X{:=}S']]\leq T_h[X{:=}S'][Y{:=}A[X{:=}S']]$$
$$C[X{:=}S']\vdash T'_k[Y{:=}A[X{:=}S']]\leq T'_{\tilde h}[Y{:=}A[X{:=}S']]$$

By (trans):

$$C[X{:=}S']\vdash T'_k[Y{:=}A[X{:=}S']]\leq T_h[X{:=}S'][Y{:=}A[X{:=}S']]$$

From $C[X{:=}S']\vdash S'\leq S$ and $Y\notin dom(C[X{:=}S'])$ follows that $Y\notin FV(S')$. Thus the last judgement becomes:

$$C[X{:=}S']\vdash T'_k[Y{:=}A[X{:=}S']]\leq T_h[Y{:=}A][X{:=}S']=T[X{:=}S']$$

□

Lemmas 9.4.5 and 9.4.7 constituted the hard part of the proof. It is then rather straightforward to prove the theorem of generalized subject reduction by using the same technique used for $\lambda\&$.

Theorem 9.4.8 (generalized subject reduction) *If $C\vdash a{:}T$ and $C\vdash a \rhd b$ then $C\vdash b{:}T'$ and $C\vdash T'\leq T$*

Proof. The proof is by induction on the depth of the proof of $C\vdash a\rhd b$. Instead of presenting the proof for the base case (the rules (β), $(\beta_\forall)$ and $(\beta_{\{\}})$) and for the inductive case (the context rules), we think that it is more intelligible if we do a case analysis on the structure of a:

$a{\equiv}x^T$ trivial.

$a{\equiv}\varepsilon$ trivial

$a{\equiv}\mathsf{Top}$ trivial

$a{\equiv}\lambda x^{T_1}.a'$, $C\vdash a'\rhd b'$ and $b\equiv\lambda x^{T_1}.b'$. This case is solved by a straightforward use of the induction hypothesis.

$a{\equiv}\Lambda X{\leq}T_1.a'$ $C\cup\{X{\leq}T_1\}\vdash a'\rhd b'$ and $b\equiv\Lambda X{\leq}T_1.b'$. This case is solved by a straightforward use of the induction hypothesis.

$a \equiv (a_1 \&^I a_2)$ just note that whichever reduction is performed the reductum is well-typed and the type does not change

$a \equiv a_1(a_2)$ where $C \vdash a_1 : W$, $C \vdash a_2 : S' \leq S$ and $\mathcal{B}(W)_C = S \to T$. Then there are three possible subcases:

1. $a_1 \equiv \lambda x^S.a_3$ and $b \equiv a_3[x^S := a_2]$. this case follows from lemma 9.4.6
2. $C \vdash a_1 \rhd a_1'$. Then by induction hypothesis we have $C \vdash a_1' : T'' \leq W$. By proposition 9.2.3 $C \vdash \mathcal{B}(T'')_C \leq \mathcal{B}(W)_C$. Since $\mathcal{B}(T'')_C$ is a not a type variable then it is of the form $S'' \to T'$ with $C \vdash S' \leq S \leq S''$ and $C \vdash T' \leq T$. Thus b is well-typed and with type $T' \leq T$.
3. $C \vdash a_2 \rhd a_2'$. Then by induction hypothesis we have $C \vdash a_2' : S'' \leq S' \leq S$. Thus $C \vdash b : T$

$a \equiv a'(S)$ where $C \vdash a' : W$, $C \vdash S \leq S'$, $\mathcal{B}(W)_C = \forall(X \leq S')S''$ and $T \equiv S''[X := S]$. Since $\mathcal{B}(W)_C = \forall(X \leq S')S''$, then

$$C \vdash \forall(X \leq S')S'' \text{ type}$$

this holds only if

$$C \cup \{X \leq S'\} \vdash S'' \text{ type}$$

from which we deduce that $X \notin dom(C)$. From this and from $C \vdash S \leq S'$ we deduce that $X \notin FV(S)$.
Now there are two possible subcases:

1. $a' \equiv \Lambda X \leq S'.a''$ and $b \equiv a''[X := S]$. But since $C \vdash S \leq S'$ and $X \notin FV(S)$ we can apply lemma 9.4.7. The result follows from $X \notin dom(C)$.
2. $C \vdash a' \rhd b'$. thus by induction hypothesis and by proposition 9.2.3 we obtain $C \vdash b' : T'' \leq W$ and $C \vdash \mathcal{B}(T'')_C \leq \mathcal{B}(W)_C$. Since $\mathcal{B}(T'')_C$ is not a type variable then it is of the form $\forall(X \leq U')U''$ with $C \vdash S \leq S' \leq U'$ and $C \cup \{X \leq S'\} \vdash U'' \leq S''$. Thus b is well-typed and $C \vdash b : U''[X := S]$. The result follows from the main lemma applied to $C \vdash U'' \leq S''$ and the fact that $X \notin dom(C)$

$a \equiv a'[A]$ where $C \vdash a' : W$ and $\mathcal{B}(W)_C = \forall X\{A_i.T_i\}_{i \in I}$. As in the case before it is possible to prove that $X \notin dom(C)$ and that $X \notin FV(A)$. Let $A_h = \min_{i \in I}\{A_i \mid C \vdash A \leq A_i\}$. Then $T \equiv T_h[X := A]$. Again we have two subcases:

1. $a' \equiv (a_1 \&^{[A_1.T_1 \| \ldots \| A_n.T_n]} a_2)$ and $A, A_1, \ldots, A_n$ are closed and a $\beta_{\{\}}$-reduction is performed. Then either $b \equiv a_1[A]$ (case $A_h \neq A_n$) or $b \equiv a_2(A)$ (case $A_h = A_n$). In both cases, by [{}ELIM] or by [∀ELIM] according to the case, it is easy to prove that the terms have type $T' \leq T_h[X := A]$: just use the induction hypothesis and then apply the main lemma.
2. $C \vdash a' \rhd a''$. Then by induction hypothesis $C \vdash a'' : W' \leq W$ and by proposition 9.2.3 $C \vdash \mathcal{B}(W')_C \leq \mathcal{B}(W)_C$. Since $\mathcal{B}(W')_C$ is not a type variable $\forall X\{A_j'.T_j'\}_{j \in J}$. Thus by the subtyping rule ({}) there exists $\tilde{h} \in J$ such that $C \vdash A \leq A_h \leq A_{\tilde{h}}'$. Therefore the set $\{A_j' \mid C \vdash A \leq A_j', j \in J\}$ is not empty, and by the meet-closure

of $\{A'_j\}_{j\in J}$ it has also a minimum. Call this minimum A'_k. Then $C \vdash b : T'_k[X{:=}\,A]$. Since $S\equiv T_h[X{:=}\,A]$ we have to prove that

$$C \vdash T'_k[X{:=}\,A] \leq T_h[X{:=}\,A]$$

Take again the previous $\tilde{h}$; by the rule ({}) we have

$$C \vdash \forall(X{\leq}A'_{\tilde{h}})T'_{\tilde{h}} \leq \forall(X{\leq}A_h)T_h \tag{9.27}$$

By the definition of A_h:

$$C \vdash A \leq A_h \tag{9.28}$$

From (9.27):

$$C \vdash A_h \leq A'_{\tilde{h}}$$

From (trans):

$$C \vdash A \leq A'_{\tilde{h}} \tag{9.29}$$

From (9.27):

$$C \cup \{X \leq A_h\} \vdash T'_{\tilde{h}} \leq T_h \tag{9.30}$$

From the definition of A'_k and from (9.29) we obtain

$$C \vdash A'_k \leq A'_{\tilde{h}}$$

and from this and the rule $(\{\}_{type})$ applied to $\forall X\{A'_j.T'_j\}_{j\in J}$ it follows

$$C \cup \{X \leq A_h\} \vdash T'_k \leq T'_{\tilde{h}} \tag{9.31}$$

By (9.28) and by the choice of k we respectively have that $C \vdash A \leq A_h$ and $C \vdash A \leq A'_k$; thus we can apply the main lemma to (9.30) and (9.31) to obtain:

$$C[X{:=}\,A] \vdash T'_{\tilde{h}}[X{:=}\,A] \leq T_h[X{:=}\,A]$$
$$C[X{:=}\,A] \vdash T'_k[X{:=}\,A] \leq T'_{\tilde{h}}[X{:=}\,A]$$

But $X \notin dom(C)$, thus the judgements above get

$$C \vdash T'_{\tilde{h}}[X{:=}\,A] \leq T_h[X{:=}\,A]$$
$$C \vdash T'_k[X{:=}\,A] \leq T'_{\tilde{h}}[X{:=}\,A]$$

Finally by (trans) we obtain the result:

$$C \vdash T'_k[X{:=}\,A] \leq T_h[X{:=}\,A]$$

□

9.4.3 Church-Rosser

In section 9.2.3 we proved the syntactic coherence of the *proof system* of $F^{\&}_{\leq}$. In this section we prove the syntactic coherence of the *reduction system* of $F^{\&}_{\leq}$.

In the reductions that follow we omit, without loss of generality, all the tcs[9]. As in section 2.4 we use the Hindley-Rosen lemma:

[9] The only place where this ommission really matters is in the lemma 9.4.12 whose complete statement should be *If* $C \cup \{X{\leq}S\} \vdash a \;\rhd_{\beta_{\{\}}}\; a'$ *then* $C \vdash a[X{:=}\,T] \;\rhd^*_{\beta_{\{\}}}\; a'[X{:=}\,T]$.

Lemma 9.4.9 (Hindley-Rosen) *Let R_1,R_2 be two notions of reduction. If R_1, R_2 are* **CR** *and $\triangleright^*_{R_1}$ commutes with $\triangleright^*_{R_2}$ then $R_1 \cup R_2$ is* **CR**.

Set now $R_1 \equiv \beta_{\{\}}$ and $R_2 \equiv \beta \cup \beta_\forall$; if we prove that these notions of reduction satisfy the hypotheses of the lemma above then we proved **CR**. It is easy to prove that $\beta \cup \beta_\forall$ is **CR**: indeed in [Ghe90] it is proved that $\beta \cup \beta_\forall$ is terminating; by a simple check of the conflicts it is possible to prove that it is also locally confluent; since it has no critical pair then by the Knuth-Bendix lemma ([KB70]) it is locally confluent; finally by applying the Newman's Lemma ([New42]) we obtain **CR**.

Lemma 9.4.10 *$\beta_{\{\}}$ is* **CR**.

Proof. By lemma 3.2.2 of [Bar84] it suffices to prove that the reflexive closure of $\triangleright_{\beta_{\{\}}}$ (denoted by $\triangleright^{=}_{\beta_{\{\}}}$) satisfies the diamond property. Thus by induction on $a \triangleright^{=}_{\beta_{\{\}}} a_1$ we show that for all $a \triangleright^{=}_{\beta_{\{\}}} a_2$ there exists a common $\triangleright^{=}_{\beta_{\{\}}}$ reduct a_3 of a_1 and a_2. We can assume that $a_1 \not\equiv a$, $a_2 \not\equiv a$ and $a_1 \not\equiv a_2$, otherwise the proof is trivial. Let examine all the possible cases:

1. $(b_1\&b_2)[A] \triangleright^{=}_{\beta_{\{\}}} b_1[A]$. If $a_2 \equiv (b_1\&b'_2)[A]$ then $a_3 \equiv a_1$; else $a_2 \equiv (b'_1\&b_2)[A]$ then $a_3 \equiv b'_1[A]$.
2. $(b_1\&b_2)[A] \triangleright^{=}_{\beta_{\{\}}} b_2(A)$. If $a_2 \equiv (b'_1\&b_2)[A]$ then $a_3 \equiv a_1$; else $a_2 \equiv (b_1\&b'_2)[A]$ then $a_3 \equiv b'_2(A)$.
3. $b_1(b_2) \triangleright^{=}_{\beta_{\{\}}} b'_1(b_2)$. If $a_2 \equiv b_1(b'_2)$ then $a_3 \equiv b'_1(b'_2)$; else $a_2 \equiv b''_1(b_2)$: then by induction hypothesis there exists b_3 common $\triangleright^{=}_{\beta_{\{\}}}$ reduct of b'_1 and b''_1; thus $a_3 \equiv b_3(b_2)$
4. $b_1(b_2) \triangleright^{=}_{\beta_{\{\}}} b_1(b'_2)$ as the case before.
5. $(b_1\&b_2)[A] \triangleright^{=}_{\beta_{\{\}}} (b'_1\&b_2)$ as the case before.
6. $(b_1\&b_2)[A] \triangleright^{=}_{\beta_{\{\}}} (b_1\&b'_2)$ as the case before.
7. $\lambda x^T.a \triangleright^{=}_{\beta_{\{\}}} \lambda x^T.a'$. Then $a_2 \equiv \lambda x.a''$ and by induction hypothesis there exists b_3 common $\triangleright^{=}_{\beta_{\{\}}}$ reduct of a' and a''. Thus $a_3 \equiv \lambda x^T.b_3$.
8. $\Lambda X{\leq}T.a \triangleright^{=}_{\beta_{\{\}}} \Lambda X{\leq}T.a'$. as the case before (apart from the changement of tcs in the induction hypothesis).
9. $a(T) \triangleright^{=}_{\beta_{\{\}}} a'(T)$ as the case before.
10. $a[A] \triangleright^{=}_{\beta_{\{\}}} a'[A]$ as the case before.

□

To prove that the two notions of reduction commute we need three technical lemmas:

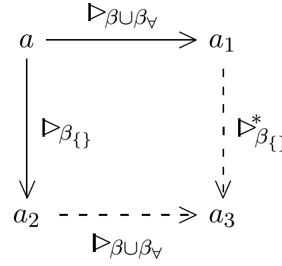
Lemma 9.4.11 *If $a \triangleright_{\beta_{\{\}}} a'$ then $a[x{:=}b] \triangleright^*_{\beta_{\{\}}} a'[x{:=}b]$*

Lemma 9.4.12 *If $a \triangleright_{\beta_{\{\}}} a'$ then $a[X{:=}T] \triangleright^*_{\beta_{\{\}}} a'[X{:=}T]$*

Lemma 9.4.13 *If $b \triangleright_{\beta_{\{\}}} b'$ then $a[x{:=}b] \triangleright^*_{\beta_{\{\}}} a[x{:=}b']$*

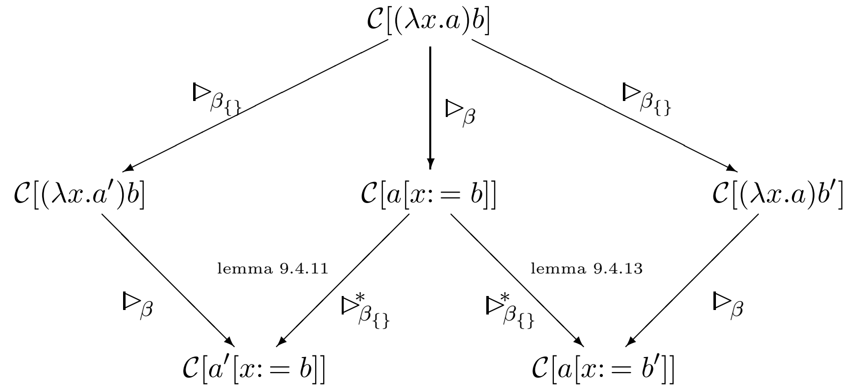
These lemmas can be proved by a straightforward use of induction (on $a \triangleright_{\beta_{\{\}}} a'$ for the first two and on a for the third). Just for the proof of the second, note that in $\beta_{\{\}}$ $A, A_1, \ldots, A_n$ are required to be closed. We can now prove that the two notions of reduction commute.

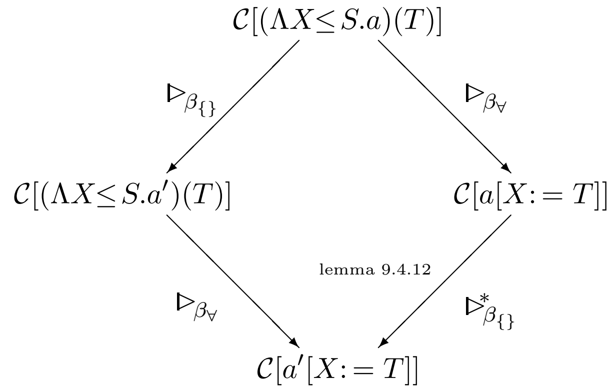
Lemma 9.4.14 *If* $a \rhd_{\beta\cup\beta_\forall} a_1$ *and* $a \rhd_{\beta_{\{\}}} a_2$ *then there exists* a_3 *such that* $a_1 \rhd^*_{\beta_{\{\}}} a_3$ *and* $a_2 \rhd_{\beta\cup\beta_\forall} a_3$*. Pictorially:*



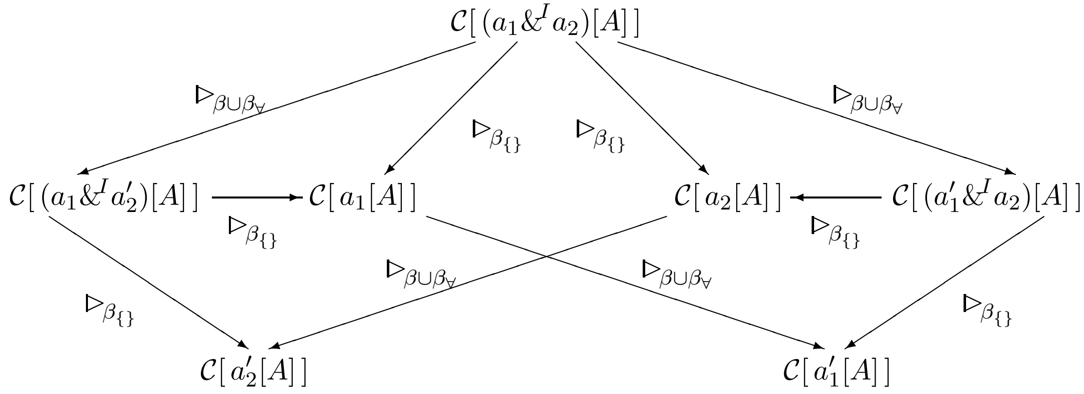
(Where full arrows are used for the hypotheses and dashed arrows for the theses.)

Proof. A proof of this lemma can be given by a simple diagram chase. Let $\mathcal{C}[\,]$ be a context (in the sense of [Bar84])[10]. Then we have the following cases:





[10] Avoid confusion between a context, denoted by $\mathcal{C}[\,]$ and a type constraint system, denoted by C.



□

Corollary 9.4.15 $\triangleright^*_{\beta_{\{\}}}$ *commutes with* $\triangleright^*_{\beta\cup\beta_\forall}$

Proof. By lemma 3.3.6 in [Bar84]. □

In conclusion all the hypotheses of lemma 9.4.9 are satisfied, and we can conclude that $F^{\&}_{\leq}$ is **CR**.

9.5 Decidable subtyping

It is obvious that the undecidability of $F_{\leq}$ is inherited by $F^{\&}_{\leq}$. In chapter 8 we have shown how to recover many of the syntactical properties that $F_{\leq}$ lacks, foremost decidability of the subtyping relation: just replace the incriminated ($\forall$) rule by the following one

$$(\forall\text{-new}) \qquad \frac{C \vdash T_1' \leq T_1 \qquad C \cup \{X \leq \mathsf{Top}\} \vdash T_2 \leq T_2'}{C \vdash \forall(X \leq T_1)T_2 \leq \forall(X \leq T_1')T_2'} \qquad X \notin dom(C)$$

Thus one may wonder whether by doing the same modification in $F^{\&}_{\leq}$ one obtains decidability. That is indeed the case, as we show in this section. We denote the resulting system by $F^{\&\top}_{\leq}$. As we already did when dealing with the transitivity elimination of $F^{\&}_{\leq}$ (see section 9.2.2), we concentrate our attention on the subtyping rules, forgetting those of type good formation; thus, once more, we suppose that all the types that appear below are well-formed.

9.5.1 Subtyping algorithm

The first thing that one has to do is to define a subtyping algorithm which is sound and complete with respect to the subtyping system. This is obtained by replacing the (Alg$\forall$) rule in the algorithm of section 9.2.3 by the new rule (Alg$\forall$-new):

$$(\text{AlgRefl}) \qquad C \vdash_{\mathcal{A}} X \leq X$$

(AlgTrans) $$\frac{C \vdash_{\mathcal{A}} C(X) \leq T}{C \vdash_{\mathcal{A}} X \leq T}$$

(AlgTop) $$C \vdash_{\mathcal{A}} T \leq \mathsf{Top}$$

(Alg→) $$\frac{C \vdash_{\mathcal{A}} T_1' \leq T_1 \qquad C \vdash_{\mathcal{A}} T_2 \leq T_2'}{C \vdash_{\mathcal{A}} T_1 \to T_2 \leq T_1' \to T_2'}$$

(Alg∀-new) $$\frac{C \vdash_{\mathcal{A}} T_1' \leq T_1 \qquad C \cup \{X \leq \mathsf{Top}\} \vdash_{\mathcal{A}} T_2 \leq T_2'}{C \vdash_{\mathcal{A}} \forall (X \leq T_1) T_2 \leq \forall (X \leq T_1') T_2'} \qquad X \notin dom(C)$$

(Alg{}) $$\frac{\text{for all } i \in I \text{ exists } j \in J \text{ s.t.} C \vdash A_i' \leq A_j \quad C \cup \{X \leq A_i'\} \vdash T_j \leq T_i'}{C \vdash \forall X \{A_j . T_j\}_{j \in J} \leq \forall X \{A_i' . T_i'\}_{i \in I}} \qquad X \notin dom(C)$$

Note that we have not changed the rule (Alg{}); one might expect that also the bound in $C \cup \{X \leq A_i'\} \vdash T_j \leq T_i'$ should be changed from A_i' to Top. This is not necessary to obtain decidability: indeed the bounds used in the overloaded quantification are far less general than those used in standard quantification, since the former can range only over constant types while the latter can range over all types, whence the undecidability (in a sense the overloaded types of $F_{\leq}^{\&\top}$ constitute a system that satisfies the restriction of [KS92], since Top cannot appear as a bound of an overloaded type). Thus we leave (Alg{}) as it is; in section 10.1 we will show how to use its full expressiveness, by defining some methods that would not be typed if (Alg{}) used the bound Top in comparing the types of different branches (see the definition of *move* at page 269). This however has a minor drawback, since once more we are not allowed to use the simple technique of chapter 8 to prove transitivity elimination and thus the completeness of the algorithm. We are obliged to use the technique of [CG92] and prove again all the theorems of sections 9.2.2 and 9.2.3 from scratch.

We will not rewrite them here since actually very few modifications to the proofs in section 9.2.2 suffice to do the work. The main modification is in the rewriting system of section 9.2.2 where you have to substitute the rules

$$\begin{array}{lll} (\forall') & (\forall (X \leq c) d)\, (\forall (X \leq c') d') & \leadsto \; \forall (X \leq c'\, c)(d\, d'[X_T := c\, X_S]) \\ (\forall'') & (\forall (X \leq c) d)\, ((\forall (X \leq c') d')\, e) & \leadsto \; (\forall (X \leq c'\, c)(d\, d'[X_T := c\, X_S]))\, e \end{array}$$

by the following ones

$$\begin{array}{lll} (\forall') & (\forall (X \leq c) d)\, (\forall (X \leq c') d') & \leadsto \; \forall (X \leq c'\, c)(d\, d') \\ (\forall'') & (\forall (X \leq c) d)\, ((\forall (X \leq c') d')\, e) & \leadsto \; (\forall (X \leq c'\, c)(d\, d'))\, e \end{array}$$

Of course, now $(\forall (X \leq c) d)$ codifies the new (∀) rule:

(∀-new) $$\frac{C \vdash c_1 : T_1' \leq T_1 \qquad C \cup \{X \leq \mathsf{Top}\} \vdash c_2 : T_2 \leq T_2'}{C \vdash \forall (X \leq c_1) c_2 : \forall (X \leq T_1) T_2 \leq \forall (X \leq T_1') T_2'}$$

The reader can now move across the proofs of section 9.2.2 and check the obvious modifications; the proofs of section 9.2.3 are essentially unchanged.

9.5.2 Termination

We now prove that this algorithm terminates.

Definition 9.5.1 Let C be a tcs and T a type such that $FV(T) \subseteq dom(C)$ then define

$$\mathcal{L}(T)_C = \begin{cases} 0 & \text{if } T \text{ is not a type variable} \\ \mathcal{L}(C(T))_C + 1 & \text{otherwise} \end{cases}$$

($\mathcal{L}$ stays for "length") □

Notation 9.5.2 *Let C be a tcs; we denote by $\widehat{C}$ a type variable $Y \in dom(C)$ such that*

$$\mathcal{L}(Y)_C = \max_{X \in dom(C)} \{\mathcal{L}(X)_C\}$$

If there is more than one such a variable then choose any of them (e.g. use the textual order)

We can now define a weight $\mathcal{T}$ for a type T with respect to a tcs C (such that T is well-formed in C):

$$\begin{aligned}
\mathcal{T}(B)_C &\stackrel{def}{=} 1 \\
\mathcal{T}(\mathsf{Top})_C &\stackrel{def}{=} 1 \\
\mathcal{T}(X)_C &\stackrel{def}{=} \mathcal{T}(C(X))_C + 1 \\
\mathcal{T}(S_1 \to S_2)_C &\stackrel{def}{=} \mathcal{T}(S_1)_C + \mathcal{T}(S_2)_C \\
\mathcal{T}(\forall (X \leq S_1) S_2)_C &\stackrel{def}{=} \mathcal{T}(S_1)_C + \mathcal{T}(S_2)_{C \cup \{X \leq S_1\}} \\
\mathcal{T}(\forall X \{A_i.T_i\}_{i \in I})_C &\stackrel{def}{=} \max_{i \in I} \{\mathcal{T}(A_i)_C, \mathcal{T}(T_i)_{C \cup \{X \leq \widehat{C}\}}\} + 1
\end{aligned}$$

Lemma 9.5.3 *For each type T well-formed in a tcs C, the weight $\mathcal{T}(T)_C$ is finite and positive.*

Proof. First, it is obvious that the weight $\mathcal{T}(T)_C$ is always positive. Now to prove that it is also finite, we give a well-founded rank for $\mathcal{T}(T)_C$ (i.e. we define a weight for the definition of the weight) and we show that it decreases at each stage in the definition of $\mathcal{T}$. To define the rank of $\mathcal{T}(T)_C$ consider all the variables that appear in T and C (no matter whether they appear free or bounded, only in a quantifier or in a bound). Since T is well-formed in C, every variable is associated to a unique bound (either in C or in T) apart those appearing in T as a quantification of an overloaded type; to these variables associate as bound $\widehat{C}$. Furthermore it is also possible to totally order these variables in a way that if X_i is defined in the bound of X_j then X_i precedes X_j (C is a tcs so it is $C \cup \{X \leq \widehat{C}\}$ —with $X \notin dom(C)$—, T is well-formed in C, thus loops are not possible). If there is more than one order satisfying this condition then choose one arbitrarily. Define the *depth* of each variable as the number of variables that precede it in this order. Then the rank of $\mathcal{T}(T)_C$ is the lexicographical size of the pair (D, L), where D is the maximum depth of any of the variables that appear in T, and L is the textual length of T. This rank is well-founded (the least element is $(0, 1)$). Take now the definition of $\mathcal{T}$: it easy to see that for the subproblems on the right-hand side of $\mathcal{T}(S_1 \to S_2)_C$, $\mathcal{T}(\forall (X \leq S_1) S_2)_C$ and $\mathcal{T}(\forall X \{A_i.T_i\}_{i \in I})_C$, the component D either is the same or it decreases, while the L component always strictly decreases; for the case $\mathcal{T}(X)_C$, the component D strictly decreases. □

The weight of the types is extended to a weight for type judgments in the obvious way:

$$\mathcal{J}(C \vdash S_1 \leq S_2) = \mathcal{T}(S_1)_C + \mathcal{T}(S_2)_C.$$

Now we can show the termination of the algorithm.

Lemma 9.5.4 *Given a tcs C and a type variable X, for all types T_1, T_2 such that $FV(T_i) \subseteq dom(C)$ $(i = 1, 2)$* $\quad \mathcal{T}(T_1)_{C\cup\{X\leq\mathrm{Top}\}} \leq \mathcal{T}(T_1)_{C\cup\{X\leq T_2\}}$.

Proof. A simple induction on the definition of $\mathcal{T}(T_1)$ (note that one of the consequences of lemma 9.5.3 is that it is possible to use induction on $\mathcal{T}$). □

Lemma 9.5.5 *Given a tcs C, a type variable $X \notin dom(C)$, two atomic types A and A' such that $\mathcal{B}(A)_C$ and $\mathcal{B}(A')_C$ are constant types, if $\mathcal{L}(A)_C \leq \mathcal{L}(A')_C$ then:*
(a) $\mathcal{T}(A)_C \leq \mathcal{T}(A')_C$
(b) *for every type T such that $FV(T) \subseteq dom(C) \cup \{X\}$, $\mathcal{T}(T)_{C\cup\{X\leq A\}} \leq \mathcal{T}(T)_{C\cup\{X\leq A'\}}$*

Proof. there are three possible cases:

1. Both A and A' are constant types: (a) is trivial; (b) follows by a straightforward induction on $\mathcal{T}(T)_{C\cup\{X\leq A\}} + \mathcal{T}(T)_{C\cup\{X\leq A'\}}$, performing a case analysis on T.

2. A is a constant type and A' is a type variable: as the previous case

3. Both A and A' are type variables: we prove (a) by induction on $\mathcal{L}(A)_C + \mathcal{L}(A')_C$. The base case is when $\mathcal{L}(A)_C = \mathcal{L}(A')_C = 1$. In that case it is easy to check that $\mathcal{T}(A)_C = \mathcal{T}(A')_C = 2$. When that sum is strictly larger than 2 then by definition of $\mathcal{T}$

   $$\mathcal{T}(A)_C \leq \mathcal{T}(A')_C \iff \mathcal{T}(C(A))_C \leq \mathcal{T}(C(A'))_C$$

   By definition of $\mathcal{L}$, $\mathcal{L}(A)_C \leq \mathcal{L}(A')_C$ implies $\mathcal{L}(C(A))_C \leq \mathcal{L}(C(A'))_C$; therefore we can apply the induction hypothesis to obtain the result.

   Once more, (b) follows by a straightforward induction on $\mathcal{T}(T)_{C\cup\{X\leq A\}} + \mathcal{T}(T)_{C\cup\{X\leq A'\}}$, performing a case analysis on T: use the case (a) of this lemma when $T \equiv X$.

□

Theorem 9.5.6 *At every step of the subtyping algorithm, the weight of each of the premises is strictly smaller than the weight of the conclusion.*

Proof. The verification is easy in most cases. The only non-trivial cases are (Alg∀) and (Alg{}). The first case is proved by the following inequations:

$$\begin{array}{rcll}
\mathcal{J}(C \cup \{X \leq \mathsf{Top}\} \vdash S_2 \leq T_2) & = & \mathcal{T}(S_2)_{C\cup\{X\leq\mathsf{Top}\}} + \mathcal{T}(T_2)_{C\cup\{X\leq\mathsf{Top}\}} & \\
& \leq & \mathcal{T}(S_2)_{C\cup\{X\leq S_1\}} + \mathcal{T}(T_2)_{C\cup\{X\leq T_1\}} & \text{by lemma 9.5.4} \\
& < & \mathcal{T}(S_1)_C + \mathcal{T}(T_1)_C + \mathcal{T}(S_2)_{C\cup\{X\leq S_1\}} + \mathcal{T}(T_2)_{C\cup\{X\leq T_1\}} & \\
& = & \mathcal{T}(\forall(X \leq S_1)S_2)_C + \mathcal{T}(\forall(X \leq T_1)T_2)_C & \\
& = & \mathcal{J}(C \vdash \forall(X \leq S_1)S_2 \leq \forall(X \leq T_1)T_2) &
\end{array}$$

For (Alg{}) the proof is given by these inequations:

$$\begin{array}{rcll}\mathcal{J}(C \cup \{X{\leq}A'_i\} \vdash T_j{\leq}T'_i) & = & \mathcal{T}(T_j)_{C\cup\{X\leq A'_i\}} + \mathcal{T}(T'_i)_{C\cup\{X\leq A'_i\}} & \\ & \leq & \mathcal{T}(T_j)_{C\cup\{X\leq\widehat{C}\}} + \mathcal{T}(T'_i)_{C\cup\{X\leq\widehat{C}\}} & \text{by lemma 9.5.5} \\ & \leq & \max_{j\in J}\{\mathcal{T}(A_j)_C, \mathcal{T}(T_j)_{C\cup\{X\leq\widehat{C}\}}\} + \max_{i\in I}\{\mathcal{T}(A'_i)_C, \mathcal{T}(T'_i)_{C\cup\{X\leq\widehat{C}\}}\} & \\ & < & \mathcal{J}(C \vdash \forall X\{A_j.T_j\}_{j\in J}{\leq}\forall X\{A'_i.T'_i\}_{i\in I}) & \end{array}$$

□

Corollary 9.5.7 *The algorithm terminates*

9.5.3 Terms and reduction

Up to now we dealt with the types of $F_{\leq}^{\&\top}$. To end with it, it still remains to describe its terms and reduction rules. The task is easy for the raw terms and the reduction rules, which are exactly the same as those for $F_{\leq}^{\&}$. More difficult is instead the case for the typing rules; we have two different choices: either we use the same typing rules as for $F_{\leq}^{\&}$ and we do not allow reductions involving free type variables, or we add to these rules the rule of subsumption (see page 90) and we leave the reduction unchanged. As for the case of $F_{\leq}^{\top}$, in the first case we are not able to prove the subject-reduction property, while in the second the minimal typing property does not hold.

Note that in both cases there will be less well-typed terms than in $F_{\leq}^{\&}$, since the subtyping relation of $F_{\leq}^{\&\top}$ (with or without subsumption) is strictly contained in that of $F_{\leq}^{\&}$: therefore there will be less well-formed types (some types well-formed in $F_{\leq}^{\&}$ may not satisfy the covariance rule) and some functional applications may no longer result well-typed.

Remark On the contrary of what happened in chapter 3, the contraction of the subtyping relation (on pretypes) does not imply the expansion of the set of well-formed types: there, you could have some pretypes that in one case do not satisfy the multiple inheritance condition, and in the other they do; here a type satisfies ∩-closure in one system if and only if it satisfies it in the other, since ∩-closure involves only (types bounded by) basic types, while the subtyping relations differs only on universally quantified types. A similar argument holds for the domains when checking the covariance rule. Thus $F_{\leq}^{\&\top}$*is* a subcalculus of $F_{\leq}^{\&}$.

Since subject-reduction is very important from both a theoretical and a practical point of view, we prefer to use the subsumption rule to define the type system of $F_{\leq}^{\&\top}$; in this case the property of subject-reduction still holds. It just requires some work to adapt to the subcalculus the proof of subject reduction of section 9.4.2: essentially you have to modify the various cases of $a{\equiv}\, a'(T)$ to take into account the new subtyping rule (∀-new), and use the subsumption rule in the cases $a{\equiv}\,\Lambda X \leq S_1.a'$ to show that the type *is preserved*; in the case of lemma 9.4.7 the proof results even simplified. The proof of Church-Rosser for $F_{\leq}^{\&\top}$ is then a consequence of its subject reduction property, and of the fact that $F_{\leq}^{\&}$ is **CR**: given a term M of $F_{\leq}^{\&\top}$, if $M \rhd^* N_1$ and $M \rhd^* N_2$ then there exists N_3 in $F_{\leq}^{\&}$ such that $N_1 \rhd^* N_3$ and $N_2 \rhd^* N_3$. But, since the notion of reduction is the same in both calculi, the subject reduction theorem for $F_{\leq}^{\&\top}$ guarantees that N_3 is a term of $F_{\leq}^{\&\top}$, too. On the line of

chapter 8, we conjecture that every pair of types in $F_{\leq}^{\&\top}$ has a least upper bound, and that, if they have a common lower bound, then they also have a greatest lower bound (not true for $F_{\leq}^{\&}$).

Of course, we would choose not to use the subsumption as soon as we proved that $F_{\leq}^{\&\top}$ without subsumption and reductions involving free type variables satisfies the subject-reduction property, since, in this system, the decidability of the subtyping relation implies the decidability of the typing relation. We are comforted in this choice by the fact that the tests we did to check the expressiveness of $F_{\leq}^{\top}$ (see section 8.3) have been performed by modifying the subtyping algorithm for $F_{\leq}$, i.e. by using a system that does not use the subsumption rule.

Clearly with $F_{\leq}^{\&\top}$ we lose in expressive power w.r.t. $F_{\leq}^{\&}$ since the terms (and the reductions) of $F_{\leq}^{\&\top}$ are strictly contained in those of $F_{\leq}^{\&}$. Thus some terms are lost; but, again, are those terms really interesting? We cannot answer to this question as we did in chapter 8, where we tried to type-check existing libraries of $F_{\leq}$ programs, using the $F_{\leq}^{\top}$ subtyping relation: there is no library for $F_{\leq}^{\&}$ since we have just defined it. However, we think that, for object-oriented programming, $F_{\leq}^{\&\top}$ is a good calculus to start from. We will give an idea of this in the next section, where we show how to use second order overloading to model object-oriented features; all the examples we will show are typable in $F_{\leq}^{\&\top}$ *without subsumption*; of course they are typable in $F_{\leq}^{\&}$, and in $F_{\leq}^{\top}$ with subsumption, too.

Chapter 10

Second order overloading and object-oriented programming

In this chapter we mimic, for our second order system, what we have done in section 2.6 for the $\lambda\&$-calculus. We show how object-oriented programming can be modeled by using second order overloading. All the examples of this chapter are typable both in $F^{\&}_{\leq}$ and $F^{\&\top}_{\leq}$ (with or without subsumption).

10.1 Object-oriented programming

Informally, from the point of view of our toy object-oriented language the gain of considering a second order system is that we can use in the class interfaces the reserved keyword `Mytype` which denotes the type of the receiver of a message. Note that by inheritance the type of the receiver of a message can be smaller than the class(-name) for which the method has been defined (`Mytype` is the keyword used in [Bru92]; examples of other keywords are "`like current`" [Mey88] and `myclass` [CCHO89]). To show the use of this keyword we slightly modify the definitions of the classes `2DPoint` and `2DColorPoint` of examples 1.1.3 and 1.1.4 (pages 69 and 72)

```
class 2DPoint
  {
    x:Int = 0;
    y:Int = 0
  }
    norm = sqrt(self.x^2 + self.y^2);
    erase = (update{x = 0});
    move = fn(dx:Int,dy:Int) => (update{x=self.x+dx; y=self.y+dy})
  [[
    norm: Real;
    erase: Mytype;
    move: (Int x Int) -> 2DPoint
  ]]
```

```
class 2DColorPoint is 2DPoint
  {
    x:Int = 0;
    y:Int = 0;
    c:String = "black"
  }
    isWhite = (self.c == "white")
    move = fn(dx:Int,dy:Int) =>
                      (update{x=self.x+dx; y=self.y+dy; c="white"})
  [[
    isWhite: Bool
    move: (Int x Int) -> Mytype
  ]]
```

Remark that we have modified only the interfaces, using in two places the keyword `Mytype`. Recall that at the beginning of chapter 9 we had remarked that in our original toy language the type system would have assigned to the term `[new(2DColorPoint) erase]` the type `2DPoint`. Now, the keyword `Mytype` in the interface says that the type returned by the passage of `erase` is the same as the type of the receiver. In the case of `[new(2DColorPoint) erase]`, therefore, the type inferred is `2DColPoint`. Note also that, in the interface of `2DColorPoint`, the message `move` returns `Mytype` instead of `2DPoint`. The other way round is not allowed, i.e. it is not possible to replace `Mytype` by a class-name. For example the following definition

```
extend 2DColorPoint
   erase = new(2DColorPoint)
[[ erase: 2DColorPoint ]]
```

would not be well-typed since the method `erase` in `2DPoint` returned `Mytype`. Indeed, `Mytype` in the interface of `2DPoint` might assume any type smaller than `2DPoint` and thus, in particular, also a type smaller than `2DColorPoint`. In that case, covariance would not be respected[1].

Let see how this behavior is formalized in $F_{\leq}^{\&}$.

As usual we use the name of a class to type the objects of that class, and a message is (an identifier of) an overloaded function whose branches are the methods associated to that message. The method to be executed is selected according to the type (the class-name) passed as argument (and not the type of the argument) which will be the class of the object the message is sent to. Thus the sending of a message *mesg* to an object a of class A is here modeled by $(mesg[A])a$.

Class-names are *basic types* that we associate to a *representation type*. We use the same conventions as in section 2.6, thus there is an operation $_^{classType}$ to transform a record value $r\colon R$ into a class type value $r^{classType}$ of type *classType*, provided that the representation type of *classType* is R.

[1]Of course in the previous example it would have been more reasonable that `move` in `2DPoint` returned `(Int x Int) -> Mytype` rather than `(Int x Int) -> 2DPoint`.

Again we declare

$2DPoint \doteq \langle\!\langle x : \mathrm{Int}; y : \mathrm{Int}\rangle\!\rangle$
$2DColorPoint \doteq \langle\!\langle x : \mathrm{Int}; y : \mathrm{Int}; c : \mathrm{String}\rangle\!\rangle$

and impose that on the types *2DColorPoint* and *2DPoint* we have the following relation $2DColorPoint \leq 2DPoint$

The message `norm` is translated into

$$norm \equiv (\varepsilon \,\&\, \Lambda Mytype {\leq} 2DPoint\,.\lambda self^{\,Mytype}.\sqrt{self.x^2 + self.y^2})$$

whose type is $\forall Mytype.\{2DPoint.Mytype \to \mathrm{Real}\}$
We have used the variable *self* to denote the receiver of the message and, following the notation of [Bru92], the type variable *Mytype* to denote the type of the receiver. Note however that we do not need, as in [Bru92], recursion for these features since they are just parameters of the message.

The meaning of the covariance condition of section 9.2 in this framework can be shown by the translation of the message *move*

$$\begin{array}{ll} move \equiv & (\ \Lambda Mytype {\leq} 2DPoint.\lambda self^{\,Mytype}.\\ & \qquad \lambda(dx,dy)^{Int\times Int}.\langle self \leftarrow x = self.x + dx, y = self.y + dy\rangle^{2DPoint}\\ & \&\,\Lambda Mytype {\leq} 2DColorPoint.\lambda self^{\,Mytype}.\\ & \qquad \lambda(dx,dy)^{Int\times Int}\langle self \leftarrow x = self.x + dx, y = self.y + dy, c = \text{``}white''\rangle^{Mytype}\\ &) \end{array}$$

which has type

$$\begin{array}{l} \forall Mytype.\{2DPoint.Mytype \to (Int \times Int) \to 2DPoint,\\ \qquad 2DColorPoint.Mytype \to (Int \times Int) \to Mytype\} \end{array}$$

Since $2DColorPoint \leq 2DPoint$ we check that the covariance condition is satisfied:

$$\{Mytype \leq 2DColorPoint\} \vdash Mytype \to (Int \times Int) \to Mytype \leq Mytype \to (Int \times Int) \to 2DPoint$$

In general if a method has been defined for the message m in the classes B_i for $i \in I$ then its type is of the form $\forall Mytype.\{B_i.Mytype \to T_i\}_{i\in I}$. If $B_h {\leq} B_k$ that means that the method defined for m in the class B_h overrides the one defined in B_k. Since *Mytype* is the same in both branches then the covariance condition reduces to prove that

$$\{Mytype {\leq} B_h\} \vdash T_h {\leq} T_k \tag{10.1}$$

Thus as for the case of simple typing the covariance condition requires that an overriding method returns a type smaller than or equal to the type returned by the overridden one. Note that if a method returns a result of type *Mytype* then a method that overrides it has to return *Mytype* too and it is not allowed to return, say, the class-name of the class in which the method is being defined (since, by inheritance, this could be a type larger then the actual value of *Mytype*), as we anticipated for the toy language.

For the problem of loss of information note that `erase` is translated into

$\mathit{erase} \equiv$ ($\Lambda \mathit{Mytype}{\leq}\, \mathit{2DPoint}.\lambda \mathit{self}^{\;\mathit{MyType}}.\langle \mathit{self} \leftarrow x = 0, y = 0\rangle^{\mathit{Mytype}}$)

which has type:

$$\forall \mathit{Mytype}.\{\mathit{2DPoint}.\mathit{Mytype} \rightarrow \mathit{Mytype}\}$$

If an object b of type *2DColorPoint* receives the message *erase* then this is translated into $\mathit{erase}[\mathit{2DColorPoint}](b)$; but since $\mathit{erase}[\mathit{2DColorPoint}](b) : \mathit{2DColorPoint}$ the loss of information is avoided.

In this framework bounds are always basic types (more precisely class-names); thus the $\cap$-closure reduces to impose that if a message has type $\forall X.\{B_i.T_i\}_{i\in I}$ and there exists $h, k \in I$ such that B_h and B_k have a common subclass then there must be a method defined for the message, in the class that is the g.l.b. of B_h and B_k. In other terms, we obtain the same condition of multiple inheritance as in chapter 2. To pass to real object-oriented languages the condition of meet-closure should be weakened in a way similar to the one of section 4.1.1, i.e. we have to consider maximal elements of the set of lower bounds rather than the glb.

10.1.1 Extending classes

As in λ_object method addition or redefinition are implemented by branch concatenation. Thus for example the expression

```
extend 2DPoint
    isOrigin = (self.x == 0) and (self.y == 0)
  [[ isOrigin : Bool ]]
```

is implemented in $F^{\&}_{\leq}$ by

let $\mathit{isOrigin} = (\mathit{isOrigin}\ \&\ \Lambda \mathit{Mytype}{\leq} 2DPoint.\lambda \mathit{self}^{\mathit{Mytype}}.(\mathit{self}.x = 0)\ \text{AND}\ (\mathit{self}.y = 0))$

10.1.2 First class messages, super and coerce

Clearly also in $F^{\&}_{\leq}$ messages are first class. This is not the same for super and coerce:

On one hand in $F^{\top}_{\leq}$ we have a full control on the types the selection of a branch is based on, since overloaded functions take as argument directly types. A trivial example is the implementation of a super-like function: suppose that in the definition of a method you want to send a message to *self* but that the method selected must be the one defined for the objects of a certain class C. This can be obtained by the following function:

let super[C] $= \lambda m^{\forall X\{C.T\}}.m[C]\mathit{self}$

This function takes a message m accepting objects of class C and send it to *self* but selecting the method defined for the object of class C (of course this function is well typed if $\mathit{Mytype} \leq C$).

On the other hand one does not have direct constructs that can be applied to an object to change its type, but types have to be changed directly in the code. Thus it is not evident how to translate for example $\mathbf{coerce}^A(a)$.

10.1.3 Typing rules for the toy language

Let see how this interpretation of the constructs is reflected in the type discipline for our toy language. We do not give here, as we did for λ_object, a formal translation of the language into $F_{\leq}^{\&}$ and a proof of the correctness of the type discipline: we just follow the intuition suggested by $F_{\leq}^{\&}$, keeping the presentation very informal.

We just consider a restricted version of the toy language, without multiple dispatching (which is dealt with in the next section) and in which messages (overloaded functions) are not first class (thus they cannot be neither the argument nor the result of a function). We give this restriction in order to maintain to a minimum the modifications we have to make to the type system. We will say more about it below.

We modify the language by adding the following productions

Terms	$v ::=$ `new(Mytype)`
Raw Types	$T ::=$ `Mytype`

We do not detail the modifications to make to the definitions of the types and of the terms in order to exclude first class messages. They are very simple and the reader can easily find them out.

Recall that a method M defined in the class A is translated into

$$\Lambda Mytype \leq A.\lambda self^{Mytype}.M \tag{10.2}$$

Thus we have to modify the rule of type-checking in order to take into account the new type of `self`; we split the rule [Taut] (see section 5.1.2) in two rules

[TautVar] $\qquad C;S;\Gamma \vdash x : \Gamma(x) \qquad$ for $x \in Vars$

[TautSelf] $\qquad C;S;\Gamma \vdash \texttt{self} : \texttt{Mytype}$

As before $\Gamma(\texttt{self})$ records the current class (see the rule [Class] below).

Then we must modify the rule [Write] since now the update of the internal state returns a value of type `Mytype`.

[Write] $$\frac{C;S;\Gamma \vdash r : R}{C;S;\Gamma \vdash \texttt{(update } r\texttt{):}\ \texttt{Mytype}} \qquad \text{if } C \vdash R \Subset S(\Gamma(\texttt{self}))$$

We must also extend the rule [New] in order to include the case `new(Mytype):Mytype`.

Consider again the translation of a method given by the equation (10.2): the body M of the method is evaluated in a tcs in which `Mytype` is smaller than or equal to the current class. That is in the typing of a method we have to record that $Mytype \leq \Gamma(\texttt{self})$. This constraint is for example used to type the expressions `[`v *message*`]` and `super[`A`](`v`)` when v:`Mytype`. This corresponds to transforming the rule [Class] into the following one

$$\frac{C;S;\Gamma \vdash r : R \quad C' \cup \{\texttt{Mytype}\leq\Gamma(\texttt{self})\};S';\Gamma'[\texttt{self} \leftarrow A] \vdash exp_j : V_j \ {\scriptstyle (j=1..m)} \quad C';S';\Gamma' \vdash p : T}{C;S;\Gamma \vdash \texttt{class}\ A\ \texttt{is}\ A_1\texttt{,}\ldots\texttt{,}A_n\ r : R\ \ m_1\texttt{=}exp_1\texttt{;}\ldots\texttt{;}m_m\texttt{=}exp_m\ \ I\ \texttt{in}\ p : T}$$

if $A \notin dom(S)$, for $i = 1..n$ $C \vdash R \leq_{strict} S(A_i)$ and for $i = 1..m$ $\Gamma(m_i) \cup \{A \rightsquigarrow V_i\} \in_{C'}$ **Types** Where C', S' and Γ' are defined as at page 150. The same modifications are required for the rule [Extend].

Finally during the message passing we have to substitute `Mytype`, by the type of the receiver

$$[\textsc{OvAppl}]\qquad \frac{C;S;\Gamma\vdash exp_1\colon\{A_i\to T_i\}_{i\in I}\qquad C;S;\Gamma\vdash exp_2\colon A}{C;S;\Gamma\vdash \texttt{[}exp_2\ \ exp_1\ \texttt{]}\colon T_h[\texttt{Mytype}\colon= A]}\qquad A_h=\min_{i\in I}\{A_i\mid C\vdash A\le A_i\}$$

Note that since overloaded functions are not first class then we are sure that the type T_h does not contains overloaded types. This avoids possible name-clashes in the substitution $[\texttt{Mytype}\colon= A]$. However there is no conceptual difficulty in allowing also first class overloaded functions, but it would require us to change our notation for overloaded types, since each overloaded type should bear a type variable along with it. Thus for example we should redefine the definition of $\cup$ for the rule [Class]

$$\forall X\{A_1\to T_1,...,A_n\to T_n\}\cup\{A\to T\}=\forall X\{A_1\to T_1,...,A_n\to T_n,A\to T[\texttt{Mytype}\colon= X]\}$$

and the application rule would then become

$$[\textsc{OvAppl}]\qquad \frac{C;S;\Gamma\vdash exp_1\colon\forall X\{A_i\to T_i\}_{i\in I}\qquad C;S;\Gamma\vdash exp_2\colon A}{C;S;\Gamma\vdash \texttt{[}exp_2\ \ exp_1\ \texttt{]}\colon T_h[X\colon= A]}\qquad A_h=\min_{i\in I}\{A_i\mid C\vdash A\le A_i\}$$

We end this section by the good formation of the types. First of all note that if a term of the toy language has type

$$\{A_1\to T_1,\dots,A_n\to T_n\}$$

this corresponds in $F^{\&}_{\le}$ to the type

$$\forall Mytype\{A_1.Mytype\to T_1,\dots,A_n.Mytype\to T_n\}$$

Then recall the judgment (10.1) at page 269: the covariance condition must be verified under the hypothesis that *Mytype* is smaller than the least bound.

Thus one has to modify the condition **(b)** in the definition 5.1.3 of type good formation in the following way

(b) if $C\vdash D_i\le D_j$ then $C\cup\{Mytype\le D_i\}\vdash T_i\le T_j$

And that is all. As we have seen the introduction of polymorphism requires very few modifications, which are very sensitive once we have fixed our intuition thanks to the formal model suggested by $F^{\&}_{\le}$.

10.1.4 Multiple dispatch

In this chapter we have studied a very kernel calculus. A simple extension of this calculus suffices to model multiple dispatch. The simplest extension of $F^{\&}_{\le}$ to obtain it consists in allowing as bounds of an overloaded function products of basic types. Thus we redefine atomic types in the following way

$$A\ ::=\ X\ \mid\ B\ \mid\ B\times\dots\times B\qquad\text{(atomic types [}B\text{ basic types])}$$

we modify the condition in the rule of good formation for overloaded types as follows:

$$(\{\}_{type})\quad \frac{\begin{array}{c} C \vdash A_i \text{ type} \\ C \vdash \{A_i\}_{i=1..n} \cap\text{-closed} \\ C \cup \{X \leq A_i\} \vdash T_i \text{ type} \\ \text{if } C \vdash A_i \leq A_j \text{ then } C \cup \{X \leq A_i\} \vdash T_i \leq T_j \end{array}}{C \vdash \forall X\{A_1.T_1, \ldots, A_n.T_n\} \text{ type}} \quad \begin{array}{c} X \notin dom(C) \\ \mathcal{B}(A_i)_C \text{ basic type} \\ \text{or } A_i = B_1 \times \ldots \times B_m \\ \text{for } i,j \in [1..n] \end{array}$$

and of course we add tuples to terms:

$$a ::= < a, \ldots, a >$$

There are other more general extensions: for example we can change the condition in the good formation of overloaded types into "$\mathcal{B}(A_i)_C$ basic type or product of atomic basic types" allowing as bounds variables ranging on the product of basic types; or we can allow as bounds products formed by type variables and basic types.

However the extension above largely suffices to model multi-methods, and furthermore it is very easy to check that it enjoys all the properties we have already proved for $F_{\leq}^{\&}$ (and $F_{\leq}^{\&}$) as as the generalized subject reduction or **(CR)**: just run through the proofs by taking into account that now proposition 9.2.5 has the following fourth case:

4. A_i and A_j are both products of basic types and their g.l.b. is in $\{A_i\}_{i \in I}$

One example of use of multiple dispatch is the method *equal*: you want to extend the class *2DPoint* with a method that compares two points and to redefine it for *2DColorPoint*; furthermore you want that in comparing a *2DPoint* with a *2DColorPoint* the method for *2DPoint* is used. In $\lambda\&$ we had that a function *equal* of type $\{2DPoint \to 2DPoint \to Bool, 2DColorPoint \to 2DColorPoint \to Bool\}$ would not have a well-formed type since covariance is not respected. So in $\lambda\&$ we defined

$$equal : \{(2DPoint \times 2DPoint) \to Bool, (2DColorPoint \times 2DColorPoint) \to Bool\}$$

obtaining in this way multiple dispatching. When applied to *2DPoint* and a *2DColorPoint* or viceversa it executes the first branch.

In $F_{\leq}^{\&}$ the difference is subtler: indeed $\forall X\{2DPoint.X \to X \to Bool, 2DColorPoint.X \to X \to Bool\}$ is well formed. However to select the right branch you have to pass to a function of this type the greater of the types of the two actual parameters. This is not what one would like to have: one would like to pass to the function both the types of the arguments and leave to the system the task to select the right branch. This can be done by using multi-methods and defining *equal* with the following type:

$$\forall X\{2DPoint \times 2DPoint.X \to Bool, 2DColorPoint \times 2DColorPoint.X \to Bool\}$$

A possible implementation of *equal* is then

$$\begin{array}{l} equal \equiv (\Lambda X \leq 2DPoint \times 2DPoint. \\ \qquad\qquad \lambda p^X.(\pi_1(p).x = \pi_2(p).x) \wedge (\pi_1(p).y = \pi_2(p).y) \\ \quad \& \;\; \Lambda X \leq 2DColorPoint \times 2DColorPoint. \\ \qquad\qquad \lambda p^X.(\pi_1(p).x = \pi_2(p).x) \wedge (\pi_1(p).y = \pi_2(p).y) \wedge (\pi_1(p).c = \pi_2(p).c) \\ \quad) \end{array}$$

10.1.5 Advanced features

$F_{\leq}^{\&}$ is not the mere formal version of an object-oriented language. For many aspects it is less powerful than object-oriented languages, but it possesses some features that existing object-oriented languages have not. Thus one can imagine enriching an object-oriented language by these features suggested by the model; for example one could design new object-oriented languages that handle parametricity and overloading, in which classes (class-names) could be passed as arguments to functions; these functions would also be able to dispatch to different codes according to class-name received as argument. It is possible to imagine many applications of this blend of parametric polymorphism and explicit overloading; for example suppose that you have to write a general installation routine for software products, working on various machines. Suppose that you distinguish your software products between graphic software and mathematical software, and the machines between b/w and color machines. Then you would probably have the following classes: *GraphSW, MathSW*$\leq$ *Software* and *Color, B&W*$\leq$ *Machine* and your general installation routine would have type

$$install : \forall(M{\leq}Machine)\forall(S{\leq}Software)(M \times S) \rightarrow \ldots$$

The body of this routine would include some parts common to all kinds of machines and software, and then some parts specialized according to the kinds of the parameters. This specialization could be obtained by using functions of type:

$$\begin{aligned}\forall X\{&Software \times Machine.\ ...,\\ &GraphSW \times B\&W.\ ...,\\ &GraphSW \times Color.\ ...,\\ &MathSW \times B\&W.\ ...,\\ &MathSW \times Color.\ ...\}\end{aligned}$$

10.2 Future work

In this and in the previous chapter we defined and studied $F_{\leq}^{\&}$ and a variant with decidable subtyping, and we sketched how they can be used to model object-oriented features. We showed that they account for many features of object-oriented programming and that they also suggest new features to add to the existing paradigms. However there are some features that are not easily handled (e.g. the keyword *super*; see for this purpose [Cas93b]).

The major restriction is that meet-closure allows overloading only on atomic types. In the last section we showed how to weaken this condition to model multiple dispatching; though also this definition prevents us to model the generic classes of Eiffel [Mey88]. A generic class is a class parameterized by a type variable. For example if X is a type variable, one would like to define a class $Stack[X]$ with methods *pop*: X and *push*: $X \rightarrow ()$, and then obtain a stack of integers by instantiating the type variable X in the following way: `new(Stack[Int])`. We believe that it is not difficult to further weaken meet-closure to allow among the bounds of an overloaded function, monotonic type constructors. But we are at a loss to see how to allow non monotonic type constructors. In the same way it should be possible to extend meet-closure to closed types and to add recursive types to implement recursive objects (even if we think that recursive types are not strictly indispensable in this model).

Meet closure constitutes an even more serious limitation from a proof-theoretical point of view. It would be very interesting to let bounds range over all the types; this would require a suitable definition of $\cap$-closure assuring consistency also for higher order bounds. Note that the proof theory would be greatly complicated since a new level of impredicativity would be added. However this would correspond to a major increase of the expressive power. In that case, indeed, by a slight weakening of the $\beta_{\{\}}$ rule, it would be possible to obtain parametric functions as a special case of overloaded functions with only one branch.

For what it concerns the decidable variant, there is the open problem whether the system without subsumption satisfies the weak subject reduction property. Only in that case one would have the formal proof that the subtyping algorithm is reliable.

Despite these problems $F_{\leq}^{\&}$ is a important step forward. It gives us the basic type checking rules to deal with the problem of the loss of information. At the moment of the writing of this thesis we are studying the integration of generic functions (i.e. overloaded functions with late binding) into the core of ML [MH88], in order to add object-oriented features to the languages of this family. Also we think that $F_{\leq}^{\&}$ already suffices to type languages like Dylan. Also underway is the definition of an object-oriented database programming language, whose type system is based on $F_{\leq}^{\&}$.

Chapter 11

Conclusion

...'Forse tu non pensavi ch'io loico fossi'!

DANTE ALIGHIERI
Inferno; XXVII,122-123

To conclude this thesis we place our work in a more general setting, comparing it with related work and drawing some directions for future research.

11.1 Proof Theory

In [Str67] Strachey classified the polymorphism into parametric and "ad hoc". In this thesis we mainly concentrated on the "ad hoc" polymorphism; this study leads to a classification of "ad hoc" polymorphism similar to the one defined for the parametric polymorphism: indeed for the parametric polymorphism one introduces the further distinction between *implicit* (parametric) polymorphism where the quantification on the types is an external metalinguistic operation that accounts for the use of proposition schemas in the corresponding proof theory [Hin69, Mil78], and *explicit* (parametric) polymorphism where the same quantification becomes a linguistic operation and the multiplicity of instances of the schema proved by a term is linguistically expressed by a unique syntactic type [Gir72, Rey83].

We introduce in the "ad hoc" polymorphism the same classification but with a slightly different meaning: we say that the "ad hoc" polymorphism is *implicit* when the selection of the branch is based on the type *of* the argument; we say that the ad hoc polymorphism is *explicit* when the selection of the branch is based on the type *at* the argument

The use of the same words as for the parametric polymorphism is justified by the fact that the *implicit* "ad hoc" polymorphism consists in a metalinguistic quantification that ranges on the subtypes of the "input types" of the overloaded function, while in the *explicit* ad hoc polymorphism this quantification is promoted to the linguistic level. To push the resemblance even further we can consider the overloaded type $\{U_i \to T_i\}_{i\in I}$ of $\lambda\&$ as a different notation for the following type schema

$$\forall \alpha.\alpha \to \{U_i.T_i\}_{i\in I} \tag{11.1}$$

where, of course, the $\forall$ is a metalinguistic operator[1]. The meaning of this notation is that α is a generic variable that can range over (the subtypes of) the various U_i. According to the value α assumes, the corresponding T_i is returned. Maybe this notation is still a little obscure; it may help the reader to use instead of the notation of the schema (11.1) the following one.

$$\forall\alpha \in \widehat{\{U_i\}}_{i\in I}.\alpha \to \widehat{out}(\alpha) \qquad (11.2)$$

where $\{U_i \to T_i\}_{i\in I} \equiv (K, out)$ and $\widehat{\{U_i\}}_{i\in I}$ and $\widehat{out}$ are defined as in section 6.2.

Thus a $\lambda\&$-term of type $\{U_i \to T_i\}_{i\in I}$ is indeed a term whose set of types is formed by all the instances of the schema (11.1). And at the moment of application a particular instance of that schema is used

$$[\{\}\text{ELIM}] \qquad \frac{M\colon U \to \{U_i.T_i\}_{i\in I} \qquad N\colon U}{M\bullet N\colon T_j} \qquad U_j = min_{i\in I}\{U_i | U \leq U_i\}$$

Note that the domain of the overloaded function is the same as the type of its argument and that by the side condition of the rule the range of the metavariable α is indeed the set of the subtypes of the input types.

This is made explicit in the semantics by the definition of completion: there the metalinguistic quantification ranging over the set of the subtypes of the input types is expanded into its instances, to be then interpreted by a product indexed on the set of the codes of the types in its range. And indeed if we use the notation of (11.2) the rule of elimination for the overloaded functions becomes precisely the usual [$\to$ELIM][2]

$$[\{\}\text{ELIM}] \qquad \frac{M\colon U \to \widehat{out}(U) \qquad N\colon U}{M\bullet N\colon \widehat{out}(U)} \qquad (M\colon \forall\alpha \in \widehat{\{U_i\}}_{i\in I}.\alpha \to \widehat{out}(\alpha))$$

We can thus introduce a classification of the various calculi on the base of whether they have implicit, explicit or no "ad hoc" polymorphism, and implicit, explicit or no parametric polymorphism. This classification introduces nine classes of languages some of which (denoted by (1) and (2)) are empty yet as shown by the table below

	polymorphism	
	parametric	**ad hoc**
simply typed λ-calc.	none	none
$\lambda\&$-calculus	none	implicit
(1)	none	explicit
ML	implicit	none
ML+ generic functions	implicit	implicit
(2)	implicit	explicit
System $F_{(\leq)}$	explicit	none
$\lambda\&+F_{(\leq)}$	explicit	implicit
$F^{\&}_{\leq}$	explicit	explicit

[1] The T_i's cannot contain α; otherwise we enter in the field of implicit parametric polymorphism

[2] One can equivalently use [$\to$ELIM$_{\leq}$] and make α to range over $\{U_i\}_{i\in I}$ instead of $\widehat{\{U_i\}}_{i\in I}$

Note that in every calculus the two kinds of polymorphism are syntactically distinguished, for they are implemented by different terms. Thus in the table above each column refers to the characteristic of the corresponding pure calculus of terms (since as soon as we introduce ad hoc polymorphism the system does no longer satisfy parametricity, in the sense of Reynolds [Rey83, MR91]).

It should not surprise the reader the fact that the table above mixes calculi with and without a subtyping relation: indeed all the calculi we have studied here use a subtyping relation, since this implies a great increase in the expressive power. Though, this study and this classification are valid also for languages without subtyping; to reshape them without subtyping, it just suffices to use syntactic equality for the subtyping relation ($S \leq T \Leftrightarrow S \equiv T$): of course this introduces many possible simplifications in the definitions of the calculi.

We comment more in detail the various entries of this table. In this thesis we studied first of all $\lambda\&$ which has implicit ad hoc polymorphism and no parametric polymorphism. We also studied $F^{\&}_{\leq}$, which integrates the explicit forms of the two kind of polymorphism.

Very interesting, and presently under investigation, is the extension of ML by generic functions and which we classed as characterized by the two implicit forms of polymorphism. In this thesis we partially met it: indeed the type system of the toy language with `Mytype` is an half way implicit-implicit polymorphism: it uses the implicit ad hoc polymorphism (there is no type variable in the language) and a restricted form of implicit parametric polymorphism (implicit parametric polymorphism is used to type the branches of an overloaded function but not to type the λ-abstractions[3]). This combination of polymorphism, or even better this half-way, seems the right mechanism to use to type object-oriented languages. We already used it for our toy language and we are planning to use it again to define a type system for the language Dylan [App92], or more generally for strongly-typed object-oriented database programming languages.

We have not studied in this thesis a calculus with explicit parametric and implicit ad hoc polymorphism, but it can be easily obtained by adding System F to $\lambda\&$ (or vice-versa adding &-terms to F). Finally there are two empty classes of languages denoted in the table by (1) and (2). However it is not very hard to imagine how a language of these classes can be defined: for (1) just take the BNF definition of the terms of $F^{\&}_{\leq}$ at page 240 and erase the productions of the second line. At type level this correspond to erase the parametric quantification which implies for $\forall X\{A_i \to T_i\}$ that the variable X cannot appear free in the T_i's (as a matter of fact it cannot appear free anywhere). To obtain a language for the class (2) just take the language of the class (1) above and add implicit polymorphism on the λ-terms. The problem with these last three languages is that they do not seem to have any practical motivation or justification.

11.2 Object-oriented programming

In 1984 Cardelli defined the so called "objects as records" analogy (see [Car88]). According to it an object can be modeled by a record whose fields contain the methods defined for that object. Message passing is then reduced to field selection. On the base of that intuition

[3] This corresponds to have in the type schema (11.1) above some free occurrences of α in the T_i's, which are obtained by using the ML typing technique in the typing of the branches.

a lot of work has been done and new areas of research opened (see [CW85, CL91a, CG92, CMMS91, CM91, Rém89, BL90, Wan91, CCH+89, CP89]). This work has culminated in the proposal of some powerful models such as those in [PT93] and [Bru92].

In this thesis we took a different attitude, taking as basic construction overloading with late binding rather than record types. The resulting model is rather different from the record based ones: it is completely orthogonal to the two models above even if somewhat closer to the one of [PT93], where the internal state of an object is neatly separated from its methods.

Bruce's work strongly relies on the use of recursion both in types (to define methods that return a value of the same type of the object the method belongs to) and in terms (to define the self-reference in a method to its own object and its superobjects). The internal state encapsulation is implemented by an existential quantification on the instance variables that are then passed as an argument to the methods. If a method "modifies" the internal state, then it returns a new object of the same type of the object it belongs to, whence the use of recursive types.

In [PT93], Pierce and Turner further stress the role of encapsulation by giving a different encoding of object-oriented programming: again they existentially quantify the internal state but the methods which modify the internal state return a new state rather than a new object. This state is then packed into an object *externally* to the method definition. In this way they avoid the use of recursive types and give a very neat and simple model of object-oriented programming.

Both approaches accent the encapsulation offered by object-oriented programming in virtue of the tight relation between an object and its methods. In the overloading approach we focus more on the flexibility made available by the run-time dispatching of methods and, as we said, we obtain a model orthogonal to the record-based one. This orthogonality is shown by the fact that some mechanisms of OOP are easily accounted for in one model and nearly impossible in the other and vice versa. Such a difference of behavior leads to the conclusion that the two models yield two different styles of object-oriented programming. Surely, object-oriented paradigms based on the generic functions (such as CLOS) fit very well the overloading model. But also, in the record model everything seems to be delegated to objects while in overloading the leading role is played by the classes. Thus it seems that the overloading-based model is closer to class-based languages (like Smalltalk: [GR83]) where methods are defined for classes and a modification on a class is reflected to existing objects. And on the contrary the record-based model looks closer to delegation-based languages (like the first version of Modula-3: [CDG+89]), where methods are peculiar to an object and may differ in objects of the same class and where classes are not strictly necessary and play the role of object generators.

Even if overloading draws an orthogonal model with respect to the record-based one, its study is very useful also to understand the "object as record analogy" (at least we found it very useful). For example one of the most important contributions of the model is to clarify the roles of covariance and contravariance: contravariance is the right rule when you want to substitute a function of a given type by another one of a different type; covariance is the right rule when you want to specialize a branch of an overloaded function by one with a smaller input type. It is important to notice that, in this case, the new branch *does not substitute* the old branch but rather it *masks* it to the objects of some given classes. Indeed

our formalization shows that the issue of "contravariance vs. covariance" was a false problem caused by the confusion of two mechanisms that have very few in common : substitutivety and overriding. The substitutivety establishes when an expression of a given type S can be use *in the place of* an expression of a different type T. This information is used by the application: let f be a function of type $T \to U$, we want to singularize a category of types whose values can be passed as arguments to f; it must be noted that these arguments will *substitute* in the body of the function, the formal parameter of type T. To this end we define a subtyping relation such that f accepts every argument of type S smaller than T. Therefore the category at issue is the set of the subtypes of T. When T is $T_1 \to T_2$ then it may happen that in the body of f the formal parameter is applied to an expression of type T_1; hence we deduce two facts: the actual parameter must be a function too (thus if $S \leq T_1 \to T_2$ then S has the shape $S_1 \to S_2$), and furthermore it must be a function to which we can pass an argument of type T_1 (thus $T_1 \leq S_1$, yes! ... contravariance). It is clear that if one is not interested in passing functions as arguments then there is no reason to define the subtyping relation also for the arrows (this is the reason why O_2 [BDe92] works very well even without contravariance). Overriding is a totally different feature: we have an identifier m (in the circumstances, a message) which identifies, say, two functions $f : A \to C$ and $g : B \to D$ with A and B incomparable; this identifier can be applied to an expression e; the meaning of this application is the passage of e to f if e has a type smaller than A (in the sense of the substitutivety explained above), to g if it has type smaller than B. Suppose now that $B \leq A$; the application in this case is solved by selecting f if the type of e is included between A and B, g if the type is smaller than or equal to B; but there is a further problem: the types may decrease during the computation; thus it may happen that the type checker see e of type A and infers that m applied to e has type C (f is selected); but if during the computation the type of e decreases to B, the application has type D; thus D must be a type that can substitute C (in the sense of the substitutivety above), i.e. $D \leq C$. You can call it covariance, if you like, but it must be clear that it is not a subtyping rule: g does not substitute f since g will be never applied to arguments of type A; g and f are independent functions which perform two precise and different tasks: f handle the arguments of m whose type is included between A and B, g those arguments whose type is smaller than or equal to B. In this case we are not defining the substitutivety, but we are giving a formation rule for sets of functions in order to assure the type consistency of the computation.

But maybe all these arguments are still too abstract for the object-oriented practitioners. Thus let us write it in "plain" object-oriented terms: a message may have some parameters; the type (class) of each parameter may or may not be taken into account to select the right method; if a method for that message is overridden then the parameters that are taken into account for the selection must be covariantly overridden (i.e. the corresponding parameters in the overriding method must have a lesser type) and those which are not taken into account for the selection must be contravariantly overridden (i.e. the corresponding parameters in the overriding method must have a greater type). In the record-based models no argument is taken into account to select the method: the method is uniquely determined by the record you apply the dot selection to. Thus in these models you can have only contravariance (yes, we the type theorists, we were sure about it, but ...).

To be more precise, the record-based model does possess a very limited form of covariance

but it is hidden by the encoding of the objects: consider a label ℓ; by the subtyping rule for record types, if you "send" this label to two records of type S and T with $S \leq T$ then the result returned by the record of type S must have a type smaller than or equal to the type of the one returned by T. This exactly corresponds to the covariance rule, but its form is much more limited because it applies but to the record types (since we "send" a label), and not to products (i.e. multiple dispatch) nor to arrows.

Among the works that closely resemble the one contained in this thesis we have to recall the already cited thesis of Hideki Tsuiki [Tsu92], in which overloading and late binding are used to model object-oriented languages. Though, the basic mechanism of the various calculi there, is a parallel reduction: the argument of an overloaded function is applied in parallel to all the branches that have a compatible input type. This requires the further condition that the result of all these branches must be the same. Thus what it is modeled is just *coherent overloading* and it is not enough to model full object-oriented programming[4].

Finally very recently (when the draft version of this thesis was already written) we discovered O_2 [BDe92]. O_2 is an object-oriented data model for databases. The bases of this model are set in [ABW$^+$92], where the basic mechanisms are individuated in overloading and late binding. The model is typed, and it resembles the one of $\lambda\&$ (the data description language is quite similar to our toy language). However we guess that its conception was not type-theoretically driven. Indeed its type systems (see [KLR92]) has some flaws (structural subtyping is used for comparing instance variable which yields some possible run-time type errors in case of methods updating the internal states: cf page 81 of this thesis), only the objects are first class values and it suffers with the problem of loss of information.

11.2.1 Inheritance

With respect to the formalism based on records the overloading-based model lacks some features. For example this thesis does not account for pure inheritance without subtyping. Let us recall the problem we already introduced in section 1.1.5. In object-oriented languages there are two distinct hierarchies on types called subtyping and inheritance. Subtyping concerns the computation on the values and establish when the values of a given type can be used where values of another type are required. Inheritance concerns the definition of the operations for the values of a certain type and establish when the values of a given type can use some operations originally defined for another type.

According to this definition subtyping implies inheritance, since when the values of a certain type can substitute those of another type then they can also use some (indeed all) operations defined for that type. Though the reverse implication does not hold, since in some cases you can have code reuse but not substitutivity. One example of this is the method copy: suppose we have defined the following class

```
class C
  {...}
   copy = self
  [[ copy: Mytype ]]
```

[4]This correspond to requiring that an overriding method and the overridden one return the same result for equal arguments.

Clearly the code of copy can be reused by *every* class, but this does not imply that every class can be a subtype of C. If you have just subtyping and you want to define a new class C' completely different from C but with a copy method you have to redefine a method for copy in C'. With inheritance instead one can specify that C' inherits the (code of the) methods of C; a possible syntax is

```
class C' is ... inherits from C
  {...}
    :
    :
```

This can be probably obtained in $F_{\leq}^{\&}$ by allowing union types to appear as bounds of overloaded types. For example the method copy above could be modeled by

$$copy \equiv (\varepsilon \& \Lambda Mytype \leq C \cup C'.\lambda self^{Mytype}.self) : \forall Mytype\{C \cup C'.Mytype \rightarrow Mytype\}$$

The type above indicates that the branch is shared between C and C'. This branch will be selected whenever one of these two types is the selected input type. Thus the branch must be tested separately for each type, i.e.

$$\frac{X \leq C \vdash a : T \qquad X \leq C' \vdash a : T}{\vdash \Lambda X \leq C \cup C'.a : \forall (X \leq C \cup C')T}$$

Therefore the $\cup$ represents more an exclusive or than an union. Note that inheritance here is a mechanism local to the particular operation whose code must be shared; this does not imply the sharing for all the operations defined on the concerned types. Thus in our formalism we can introduce a new and partial form of inheritance according to which a type inherits only *some* operations of another type. For example one may want that C' inherits from C only the method copy. In our toy language this could be expressed by the following syntax

```
class C' is ...
  {...}
   copy = inherited from C
  [[ ... ]]
```

In this case the implementation of copy in $F_{\leq}^{\&}$ persists unchanged; note that such an operation would be impossible in the record based model where all the methods of a class must be inherited[5].

This new view of the inheritance mechanism suggests another possible improvement w.r.t. the record-based model. In section 1.1.5 we said that in object-oriented programming inheritance is a relation defined only on classes (i.e. atomic types). Now, there is no apparent reason to restrict unions in the bounds only to atomic types. For example in $F_{\leq}^{\&}$ we could also allow bounds formed by unions of products of atomic types, obtaining in this way inheritance also for multi-methods. But even more general unions are worth to be studied.

In conclusion we think that the extension by union types in the bounds may be worth to be studied in view of the connection shown above with the mechanism of inheritance. However this study was not among the objectives of this thesis for mainly two reasons:

[5] At least it would be impossible in an incremental way: we would surely loose the late binding

1. Covariance, and thus multiple dispatching already solves many cases that in the record based approach requires the use of pure inheritance

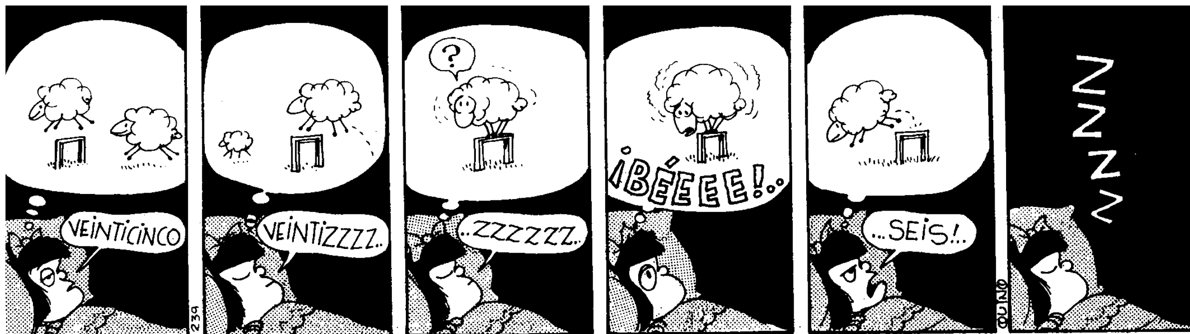
2. We strongly suspect that for this model a clever compiler and type checker could check inheritance as well (adding some fake branches implemented by code sharing), without using union types or other fancy type disciplines.

11.2.2 Higher-order bounds

We already explained in the conclusion of chapter 10 (see section 10.2) that another lack of $F_{\leq}^{\&}$ was the impossibility of model generic classes. These on the contrary are very naturally handled in the record based formalisms. Though this lack is not peculiar to model of overloading but it is due only to the fact that we were not able to deal with overloaded functions whose bounds range over any type. We want to tackle this problem in the immediate future, trying to introduce intersection types to define a $\cap$-closure property for general types.

11.2.3 Beyond object-oriented programming

Since the very beginning of this thesis we affirmed that the combination of overloading and late binding permits a high degree of incremental programming and of code reuse. Such characteristics are not and must not be exclusive to object-oriented programming. Thus it is very interesting to try to export them to other paradigms by the integration of overloading and late binding. At the moment of the writing of this thesis we have already begun to apply these mechanism to the languages of modules; in particular the study of the module system of SML with overloaded functors that use late binding is under way. However it seems that there is no limit to their application and thus we plan to extend these techniques to completely different paradigms such as concurrent systems or logic programming.



twenty-five twenty-zzzz.. ..six!..

Part III

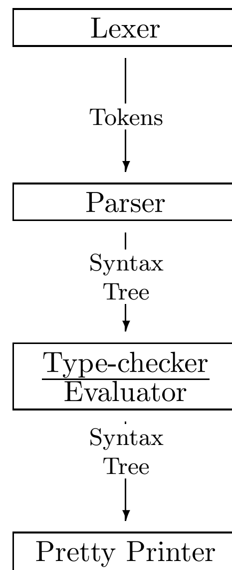
Appendixes

Appendix A

Implementation of λ_object

In this appendix we present an interpreter of λ_object written in Caml Light 0.5. The text of the program has been formatted by the program `tgrind` running under UNIX and the macro package `tgrindmac.tex` a patched by the author. Part of the code of the interpreter has been produced in the context of a project developed under the author's direction by Jean-Christophe Filliatre and François Pottier at the Ecole Normale Supérieure.

The structure of the program is quite classical:



The input is passed to a lexer, which generates a list of tokens. This is processed by the parser which returns the syntax tree corresponding to the term. The syntax tree is first type checked and then evaluated by the evaluator which applies the rules of the operational semantics. The result is the syntax tree of the term obtained after the rewriting. The syntax trees of the type and of the term are finally passed to a pretty printer which prints them in a readable form. The pretty printer is also used by the other parts of the program to print errors.

A.1 The language

The language implemented is λ_object without product types, to which we have added:

1. natural numbers (`1, 2, 3...` of type `int`) with the operators `succ` (successor), `pred` (predecessor) and `eqz` (zero equality).

2. booleans (`true` and `false` of type) with the polymorphic operator `ite` (of type schema $\forall\alpha.(\texttt{bool}\times\alpha\times\alpha)\rightarrow\alpha$)

3. A special value `?` of type `*`

4. A substitution operation denoted by `[ ]` (the expression `[x:T = M] N` is a different syntax for `let x:T = M in N`

Lambda abstractions are denoted by `\`, the fixpoint operator by `fix` and the empty overloaded function ε by `@`. We make no syntactic distinction between usual application and overloaded one. Every program must end by a semi-colon. The parser is described by the rules in figure A.1 where identifiers are denoted by the non terminal x. From the productions one can deduce the rules for associativity and precedence: application is left associative, arrows and & are right associative, application has the highest precedence, then follow in the order λ-abstractions and &-abstractions. For example the following expression:

```
@ &{int->int} \x:int.x&{int->int, (int->int->int)->int}\x:int->int->int.x 3 5
```

is parsed as:

$$((\varepsilon\ \&^{\{int\rightarrow int\}}\lambda x^{int}.x)\&^{\{int\rightarrow int,(int\rightarrow(int\rightarrow int))\rightarrow int\}}(\lambda x^{int\rightarrow(int\rightarrow int)}.((x)3)5))$$

The module `lambdaobject.zo` exports the following functions:

```
ParseString : string -> terme
ParseFile : string -> terme
TypeString : string -> unit
TypeFile : string -> unit
CalcString : string -> unit
CalcFile : string -> unit
InternalCalcString : string -> unit
InternalCalcFile : string -> unit
```

These functions return respectively the syntax tree, the type, the type and the result of a program which can be contained either in a string or in a file (`InternalCalc` evaluates also inside λ-terms and &-terms).

We show its use by a short terminal session: first we open the module `lambdaobject.zo`

```
#load_object "lambdaobject";;
- : unit = ()
##open "lambdaobject";;
```

We then perform some simple commands (in Caml Light the reverse slash in strings must be written `\\`)

```
NON TERMINAL                                                NAME USED IN THE PROGRAM

A    ::=  *  | int  | bool  | x

T    ::=  {L}W  | A W  | (T)W                               Pretype

W    ::=  -> T  | emptyword                                 rest

L    ::=  x -> T L  | , x -> T L  | emptyword               ListTypes


E    ::=  FR'  | [ x:T = E ]                                Expression

R'   ::=  &{L} FR'  | emptyword                             rest

F    ::=  SR"                                               Factor

R"   ::=  SR"  | emptyword                                  restTwo

S    ::=  ?  | succ  | pred  | eqz                          SmallExpression
      |   x  | @  | \x:T.F  | fix x:T.F
      |   true  | false  | ite(E,E,E)
      |   coerce[A](E)  | super[A](E)
      |   in[A](E)  | out[A](E)
      |   (E)  | 1  | 2  | 3  | ...


P    ::=  let x X  | E ;                                    Program

X    ::=  < A,..., A in P                                   suite
      |   hide T in P
```

Figure A.1: Parsing rules

```
#ParseString "\\x:int.x;";;
- : terme = Expression (Lambda ("x", Int, Var "x"))
#CalcString "(\\x:int->int. x 5)pred;";;
- : int
4
- : unit = ()
#CalcString "(\\x:int->bool.true)5;";;
Erreur : [type-checker] Impossible d'appliquer (int -> bool) -> bool a int
dans (\x:(int -> bool).true) 5
- : unit = ()
```

For more complex expressions it is more convenient to use a file. Suppose that the file `example` contains the following program:

```
let X hide * in
let Y hide * in
let Z hide * in

let TwoPoint hide {X->int, Y->int} in
let ThrPoint hide {X->int, Y->int , Z->int} in
let ThrPoint < TwoPoint in


[plus: int -> int -> int =
   (fix p: int -> int -> int .  \x:int. \y:int. ite(eqz x, y, succ(p (pred x) y)))]
[mult: int -> int -> int =
   (fix m: int -> int -> int . \x:int.\y:int. ite(eqz x, 0 , plus y (m (pred x) y)))]
[square: int -> int = \x:int. mult x x ]

[getx: X = in[X](?)]
[gety: Y = in[Y](?)]
[getz: Z = in[Z](?)]

[norm: {TwoPoint -> int , ThrPoint -> int}=
(@ &{TwoPoint -> int}
      \self:TwoPoint.plus (square ((out[TwoPoint](self))(getx)))
                          (square ((out[TwoPoint](self))(gety)))
    &{TwoPoint -> int , ThrPoint -> int}
      \self:ThrPoint.  plus(square ((out[ThrPoint](self))(getx)))
                      (plus(square ((out[ThrPoint](self))(gety)))
                           (square ((out[ThrPoint](self))(getz)))))]

[sameclass: TwoPoint -> TwoPoint -> bool =
     \p: TwoPoint. \q: TwoPoint .
          (@&{TwoPoint -> bool}
               \p:TwoPoint.(@ &{TwoPoint ->bool}\q:TwoPoint.true
                              &{TwoPoint ->bool,ThrPoint ->bool}\q:ThrPoint.false)q
            &{TwoPoint -> bool, ThrPoint -> bool}
               \p:ThrPoint.(@ &{TwoPoint ->bool}\q:TwoPoint.false
                              &{TwoPoint ->bool,ThrPoint ->bool}\q:ThrPoint.true)q)p]
```

```
[newTwoPoint:int -> int -> TwoPoint =
   \x:int. \y: int.
          in[TwoPoint](@&{X-> int}\l:X.x
                       &{X -> int,Y-> int}\l:Y.y)]
[newThrPoint:int -> int -> int -> ThrPoint =
   \x:int. \y: int. \z:int.
          in[ThrPoint](@&{X-> int}\l:X.x
                       &{X -> int,Y-> int}\l:Y.y
                       &{X -> int,Y -> int,Z-> int}\l:Z.z)]

[p:ThrPoint = (newThrPoint 3 5 2)]
[q:TwoPoint = (newTwoPoint 5 2)]

ite(sameclass p q , 0, plus (norm (coerce[TwoPoint](p))) (norm p));
```

In this program we have defined the class `TwoPoint` with instances variables `x`,`y` of type `int` and its subclass `ThrPoint` with instance variables `x`,`y`,`z` of type `int`. For these classes we have defined the method `norm`. We have also a function `sameclass:TwoPoint -> TwoPoint -> bool` which checks whether its arguments have the same class.

We then define two objects

```
[p:TwoPoint = (newThrPoint 3 5 2)]
[q:TwoPoint = (newTwoPoint 5 2)]
```

Note that even if `p` is declared to be an object of class `TwoPoint` it indeed contains an object of class `ThrPoint`. This is recognized by the function `sameclass` thanks to late binding; indeed `ite(sameclass p q , 0, plus (norm (coerce[TwoPoint](p))) (norm p))` executes the "else" branch:

```
#CalcFile "file.3";;
- : int
72
- : unit = ()
```

A.2 The module

Here it follows the commented (in french) listing of the module `lambdaobject.ml` and of its interface `labdaobject.mli`. The sources of the module, of its interface and the file `example` can be retrieved by anonymous ftp from

`theory.stanford.edu:/pub/castagna/LAMBDAOBJ/`

The source and the documentation of CAML LIGHT can be retrieved by anonymous ftp from

`ftp.inria.fr:/INRIA/Projects/cristal/caml-light/`

```
(*______________________________________________________________________*)
(*                                                                      *)
(*                               LEXER                                  *)
(*                                                                      *)
(*______________________________________________________________________*)

type token = LAMBDA | FIX | COLON | DOT | AMPERSEND | EPSILON
           | LPAR | RPAR | IN | OUT | COERCE | SUPER | LET
           | SMALLER | HIDE | EQUAL | IDENT of string | ARROW
           | LBRACK | RBRACK | COMMA | INT | VAL of int | SUCC
           | PRED | EQZ | BOOL | TRUE | FALSE | ITE | SEMIC
           | STAR | QUEST | LBR | RBR;;


(* Elimine les espaces et les retours chariot *)

let rec spaces = function
  [< ’‘ ‘ | ‘\t‘ | ‘\n‘; spaces _ >] -> ()
| [< >] -> ();;

let int_of_digit =
  function  ‘0‘..‘9‘ as c    -> (int_of_char c)-(int_of_char ‘0‘)
      |        _  ->  failwith "Not a Digit";;

let rec integer n = function
  [< ’ ‘0‘..‘9‘ as c; (integer (10*n+int_of_digit  c)) r>]  ->  r
| [<>]  -> VAL n;;


(* Reconnait les identificateurs et les mots reserves *)

let ident_buf = make_string 20 ‘ ‘;;
let rec ident len = function
  [< ’(‘a‘..‘z‘ | ‘A‘..‘Z‘) as c;
     (if len >= 20 then ident len
                   else (set_nth_char ident_buf len c; ident (succ len))) s >] -> s
| [< >] -> (match sub_string ident_buf 0 len with
       "in" -> IN
     | "out" -> OUT
     | "coerce" -> COERCE
     | "super" -> SUPER
     | "fix" -> FIX
     | "let" -> LET
     | "hide" -> HIDE
     | "bool" -> BOOL
     | "int" -> INT
     | "ite" -> ITE
     | "true" -> TRUE
     | "false" -> FALSE
     | "succ" -> SUCC
     | "pred" -> PRED
     | "eqz" -> EQZ
     | s -> IDENT s);;
```

```
(* Analyseur lexical *)

let rec lexer str = spaces str;
match str with
  [< '` `|`\t`|`\n`; spaces _>]  -> [< lexer str>]
| [< '`(`; spaces _ >] -> [< 'LPAR; lexer str >]        (* Reconnaissance des divers tokens *)
| [< '`)`; spaces _ >] -> [< 'RPAR; lexer str >]
| [< '`[`; spaces _ >] -> [< 'LBR; lexer str >]
| [< '`]`; spaces _ >] -> [< 'RBR; lexer str >]
| [< '`\\`; spaces _ >] -> [< 'LAMBDA; lexer str >]
| [< '`.`; spaces _ >] -> [< 'DOT; lexer str >]
| [<'`-`; '`>`; spaces _ >]  ->  [< 'ARROW; lexer str >]
| [< '`?`; spaces _ >] -> [< 'QUEST; lexer str >]
| [< '`*`; spaces _ >] -> [< 'STAR; lexer str >]
| [< '`=`; spaces _ >] -> [< 'EQUAL; lexer str >]
| [< '`&`; spaces _ >] -> [< 'AMPERSEND; lexer str >]
| [< '`@`; spaces _ >] -> [< 'EPSILON; lexer str >]
| [< '`<`; spaces _ >] -> [< 'SMALLER; lexer str >]
| [< '`{`; spaces _ >] -> [< 'LBRACK; lexer str >]
| [< '`}`; spaces _ >] -> [< 'RBRACK; lexer str >]
| [< '`,`; spaces _ >] -> [< 'COMMA; lexer str >]
| [< '`:`; spaces _ >] -> [< 'COLON; lexer str >]
| [<' `0`..`9` as c; (integer (int_of_digit c)) tok; spaces _>]
                            -> [<'tok; lexer str >]
| [< '(`a`..`z` | `A`..`Z`) as c; (set_nth_char ident_buf 0 c; ident 1) tok; spaces _ >]
                            -> [< 'tok; lexer str >]
| [< '`;` >] -> [< 'SEMIC >]
| [< 't >] -> failwith ("Erreur lexicale : le caractere "^(make_string 1 t)^" est illegal.")
| [< >] -> failwith "Point-virgule attendu.";;


(*______________________________________________________________________*)
(*                                                                      *)
(*                              PARSER                                  *)
(*                                                                      *)
(*______________________________________________________________________*)

type
  pretype = Bool | Star | Int | Atom of atomictype
          | Arrow of pretype*pretype
          | Method of (atomictype*pretype) list
and
  atomictype == string;;

type
  expression = Var of string
             | Quest
             | Const of bool
             | Val of int
             | Succ
             | Pred
             | Eqz
```

```
                | Lambda of string*pretype*expression
                | Fix of string*pretype*expression
                | Ite of expression*expression*expression
                | App of expression*expression
                | Epsilon
                | Ampersend of expression*((atomictype*pretype) list)*expression
                | Coerce of atomictype*expression
                | Super of atomictype*expression
                | In of atomictype*expression
                | Out of atomictype*expression
and
  terme = Expression of expression
                | Subtyping of atomictype*(atomictype list)*terme
                | Declaration of atomictype*pretype*terme;;


(* Quelques sous-fonctions destinees a attraper les exceptions *)

let rec printToken n s = if n = 0 then "" else let char = (match s with
  [< 'LAMBDA >] -> "\\"
| [< 'COLON >] -> ":"
| [< 'DOT >] -> "."
| [< 'AMPERSEND >] -> "&"
| [< 'EPSILON >] -> "@"
| [< 'LPAR >] -> "("                          (* Imprime n tokens du stream s *)
| [< 'RPAR >] -> ") "
| [< 'IN >] -> "in "
| [< 'OUT >] -> "out "
| [< 'COERCE >] -> "coerce "
| [< 'SUPER >] -> "super "
| [< 'LET >] -> "let "
| [< 'EQUAL >] -> "= "
| [< 'FIX >] -> "fix "
| [< 'SMALLER >] -> "< "
| [< 'HIDE >] -> "hide "
| [< 'IDENT a >] -> a^" "
| [< 'ARROW >] -> "-> "
| [< 'LBRACK >] -> "{"
| [< 'RBRACK >] -> "} "
| [< 'COMMA >] -> ", "
| [< 'BOOL >] -> "bool "
| [< 'INT >] -> "int "
| [< 'SEMIC>] -> ";"
| [< 'STAR >] -> "* "
| [< 'QUEST >] -> "? "
| [< 'PRED >] -> "pred "
| [< 'SUCC >] -> "succ "
| [< 'EQZ >] -> "eqz "
| [< 'TRUE >] -> "true "
| [< 'FALSE >] -> "false "
| [< 'ITE >] -> "ite"
| [< >] -> "") in if char = "" then "" else char^(printToken (n-1) s);;

let aulieu s = " au lieu de '" ^ (printToken 8 s) ^ "...'.";;
```

```
(* Verifie la presence d'un IDENT dans s *)
let ifIDENT s = match s with
  [< 'IDENT A >] -> A
| [< >] -> failwith ("Identificateur attendu"^(aulieu s));;

(* Verifie la presence du token t *)
let ifToken t s =
  let error debut t =
      failwith ((match t with
                    IN -> "in"
                  | LBRACK -> "{"
                  | LBR -> "["
                  | COLON -> ":"
                  | SEMIC -> ";"
                  | EQUAL -> "="
                  | DOT -> "."
                  | COMMA -> ","
                  | RPAR -> ")"
                  | ARROW -> "->"
                  | RBRACK -> "}"
                  | RBR -> "]"
                  | _ -> failwith "Impossible!")^" attendu"^(aulieu [< debut; s >]))
  in match s with
       [< 'u >] -> if u = t then () else error [< 'u >] t
     | [< >] -> error [< >] t;;


(* Reconnaissace des types  *)

exception InvalidPretype;;

let rec parseListeTypes s = match s with
  [< 'IDENT nom; (ifToken ARROW) _; parsePretype t; parseListeTypes l  >]
       -> (nom, t)::l
| [< 'COMMA; ifIDENT nom; (ifToken ARROW) _; parsePretype t; parseListeTypes l >]
       -> (nom, t)::l
| [< '(STAR | BOOL | INT) ;(ifToken ARROW) _; parsePretype t; parseListeTypes l >]
       -> failwith "Le dispatching n'est pas permis sur les types predefinis"
| [< >] -> []
and
rest t1 = function
  [< 'ARROW; parsePretype t2 >] -> Arrow (t1, t2)
| [< >] -> t1
and
parsePretype s = match s with
  [< 'LBRACK; parseListeTypes l; (ifToken RBRACK) _; (rest (Method l)) t >] -> t
| [< 'STAR; (rest Star) t >] -> t
| [< 'BOOL; (rest Bool) t >] -> t
| [< 'INT; (rest Int) t >] -> t
| [< 'IDENT atomtyp; (rest (Atom atomtyp)) t >] -> t
| [< 'LPAR; parsePretype t; (ifToken RPAR) _; (rest t) u >] -> u
| [< >] -> raise InvalidPretype;;
```

```
(* Les fonctions find* rattrapent les exceptions generees par les fonctions ci-dessus *)

let findListeTypes s = try parseListeTypes s
                       with InvalidPretype -> failwith ("Pretype attendu"^(aulieu s));;

let findpretype s = try parsePretype s
                    with InvalidPretype -> failwith ("Pretype attendu"^(aulieu s));;

(* Suit le parser pour les expressions. Il comprend plusieurs sous-fonctions :
   - parseProgram est charge d'evaluer un terme. Si il trouve "let A ...", il distingue (grace a la
     sous-fonction suite) les constructions "let A hide ..." et "let A < A1, ... An in ...". Si le
     terme ne commence pas par "let", il s'agit d'une expression.
   - parseSmallExpression evalue une expression simple (i.e. qui soit d'un seul morceau, donc n'etant
     pas formee d'une application ou d'un ampersend. La fonction rest relie les expressions calculees
     par parseSmallExpression pour former les applications et les ampersends. Elle realise un parenthesage
     implicite a gauche : "f x y" est interprete comme "(f x) y".
       E   ->  F R'                  Expression
       R'  ->  &{ } F R' | emptyword  rest
       F   ->  S R"                  Factor
       R"  ->  S R" | emptyword      restTwo
       S   ->  (E) | \x:T.F | ....   SmallExpression *)

let isempty s = try (end_of_stream s); true with Parse_failure -> false;;

(* Analyse une clause "let a < b, c, ... " *)

let renvoieListeAtomes s =
  let rec itere = function
    [< 'IDENT nom; itere l >] -> (nom::l)
  | [< 'COMMA; ifIDENT nom; itere l >] -> (nom::l)
  | [< >] -> []
in let l = itere s
in if l = [] then failwith ("Type atomique attendu"^(aulieu s)) else l;;

let rec parseProgram s =
  let suite A = function
    [< 'SMALLER; renvoieListeAtomes l; (ifToken IN) _; parseProgram p >] -> Subtyping (A, l, p)
  | [< 'HIDE; findpretype T; (ifToken IN) _; parseProgram p >] -> Declaration (A, T, p)
  | [< >] -> failwith ("'<' ou 'HIDE' attendu"^(aulieu s))
in match s with
  [< 'LET; ifIDENT A; (suite A) s >] -> s
| [< parseExpression M; (ifToken SEMIC) _ >] -> Expression M

and parseExpression s =
  let rec rest e1 s = match s with
    [< 'AMPERSEND; (ifToken LBRACK) _; findListeTypes l;
        (ifToken RBRACK) _; parseFactor e2; (rest (Ampersend (e1, l, e2))) u >] -> u
  | [< >] -> e1
  in match s with
    [< 'LBR; ifIDENT x; (ifToken COLON) _; findpretype T; (ifToken EQUAL) _; parseExpression M
         ; (ifToken RBR) _; parseExpression N >] -> App(Lambda(x, T, N), M)
  | [< parseFactor e0; (rest e0) u >] -> u
  | [< >] -> failwith ("Expression attendue"^(aulieu s))
```

```
and parseFactor s =
  let rec restTwo  e1 s = match s with
    [< parseSmallExpression e2; (restTwo (App(e1, e2))) u >] -> u
  | [< >] ->  e1
  in match s with [< parseSmallExpression e0; (restTwo e0) u >] -> u
                | [< >] -> failwith ("Expression attendue"^(aulieu s))

and parseSmallExpression s = match s with
  [< 'QUEST >] -> Quest
| [< 'TRUE >] -> Const true
| [< 'FALSE >] -> Const false
| [< 'SUCC >] -> Succ
| [< 'PRED >] -> Pred
| [< 'EQZ >] -> Eqz
| [< 'VAL n >] -> Val n
| [< 'IDENT x >] -> Var x
| [< 'LAMBDA; ifIDENT x; (ifToken COLON) _; findpretype T; (ifToken DOT) _; parseFactor M >]
              -> Lambda (x, T, M)
| [< 'FIX; ifIDENT x; (ifToken COLON) _; findpretype T; (ifToken DOT) _; parseExpression M >]
              -> Fix (x, T, M)
| [< 'ITE; (ifToken LPAR) _; parseExpression e1; (ifToken COMMA) _; parseExpression e2
                ; (ifToken COMMA) _; parseExpression e3; (ifToken RPAR) _ >] -> Ite(e1,e2,e3)
| [< 'EPSILON >] -> Epsilon
| [< 'COERCE; (ifToken LBR) _; ifIDENT A; (ifToken RBR) _;
                (ifToken LPAR) _; parseExpression M  ; (ifToken RPAR) _ >] -> Coerce (A, M)
| [< 'SUPER; (ifToken LBR) _; ifIDENT A;  (ifToken RBR) _;
                (ifToken LPAR) _; parseExpression M ; (ifToken RPAR) _ >] -> Super (A, M)
| [< 'IN; (ifToken LBR) _; ifIDENT A; (ifToken RBR) _;
                (ifToken LPAR) _; parseExpression M  ; (ifToken RPAR) _ >] -> In (A, M)
| [< 'OUT; (ifToken LBR) _; ifIDENT A; (ifToken RBR) _;
                (ifToken LPAR) _; parseExpression M  ; (ifToken RPAR) _ >] -> Out (A, M)
| [< 'LPAR; parseExpression M; (ifToken RPAR) _ >] -> M
| [< '(COLON | DOT | LET | SMALLER | HIDE | ARROW |
       LBRACK | RBRACK | BOOL | EQUAL | STAR) as bidule >]
         -> failwith ("'" ^ (printToken 8 [< 'bidule; s >]) ^ "...' n'est pas une expression!");;

(*_________________________________________________________________________*)
(*                                                                         *)
(*                            PRETTY PRINTER                               *)
(*                                                                         *)
(*_________________________________________________________________________*)


(* La fonction imprime renvoie une chaine representant un pretype *)
let rec imprime = function
  Bool -> "bool"
| Star -> "*"
| Int -> "int"
| Atom a -> a
| Arrow (t, u) -> "("^(imprime t)^" -> "^(imprime u)^")"
| Method l -> "{"^(imprimeListe l)^"}"
and imprimeListe = function
  (a, t)::l -> (imprime (Arrow(Atom a, t)))^(if l = [] then "" else ", "^(imprimeListe l))
| [] -> "";;
```

```
let rec exprToString = function
    Var x -> x
  | Val n -> (string_of_int n)
  | Const t -> if t then "true" else "false"
  | Eqz -> "eqz"
  | Pred -> "pred"
  | Succ -> "succ"
  | Quest -> "?"
  | Fix (s, t, e) -> "fix "^s^":"^(imprime t)^"."^(exprToString e)
  | Lambda (s, t, e) -> "\\"^s^":"^(imprime t)^"."^(exprToString e)
  | App (e, f) -> "("^(exprToString e)^") "^(exprToString f)
  | Epsilon -> "@"
  | Ampersend (e, l, f) -> "("^(exprToString e)^" &{"^(imprimeListe l)^"} "^(exprToString f)^")"
  | Coerce (A, e) -> "coerce["^A^"]("^(exprToString e)^")"
  | Super (A, e) -> "super["^A^"]("^(exprToString e)^")"
  | In (A, e) -> "in["^A^"]("^(exprToString e)^")"
  | Out (A, e) -> "out["^A^"]("^(exprToString e)^")"
  | Ite (e1,e2,e3) -> "ite("^(exprToString e1)^","^(exprToString e2)^","^(exprToString e3)^")";;

let prettyprint s = print_string ((exprToString s)^"\n");;



(*______________________________________________________________________*)
(*                                                                      *)
(*                            TYPE CHECKER                              *)
(*                                                                      *)
(*______________________________________________________________________*)


(* Renvoie la classe d'un objet afin de pouvoir appliquer une fonction surchargee *)

let getTag = function
  In (a, _) -> a
| Coerce (a, _) -> a
| Super (a, _) -> a
| _ -> failwith "Ceci n'est pas un objet.";;

(* La fonction smaller compare deux types atomiques. Elle tire ses renseignements de la structure hier
  (hierarchie), qui est de type (atomictype*(atomictype list)) list : a chaque type atomique elle associe
  la liste de ceux qui sont plus grands que lui.*)

let smaller hier A B = (A = B) or (mem B (try assoc A hier with Not_found -> []));;

(* La fonction findMin extrait d'une liste de types atomiques le type le moins general qui soit plus
   general que D. En fait ici liste est une (atomictype*pretype) list, mais on oublie le pretype qui ne
   nous sert a rien. La fonction findGreater ne conserve dans la liste que les types plus generaux que D.
   La fonction search en extrait le minimum. Une exception est produite si celui-ci n'existe pas.  *)

let rec findMin hier liste D =
  let rec findGreater = function
    A::l -> (if (smaller hier D A) then [A] else [])@(findGreater l)
  | [] -> []
```

```
  and search curMin = function
    A::l -> if (smaller hier A curMin) then search A l
                else search curMin l
  | [] -> curMin
in
  match findGreater (map (fun (a, t) -> a) liste) with
    A::l -> search A l
  | [] -> failwith ("Cette methode ne peut s'appliquer au type "^D^".");;

(* Suivent quelques fonctions auxiliaires ... *)


(* Renvoie le dernier element d'une liste *)
let dernier l = match (rev l) with a::l -> a | [] -> failwith "Liste vide!";;


(* Equivalent de la fonction mem de caml : prend une liste de couples et indique si
   A apparait en tant que premier element de l'un de ces couples *)
let rec customMem A = function
  B::l -> if (fst B = A) then true else (customMem A l)
| [] -> false;;

(* Prend une liste de types et verifie que chacun de ces types apparait dans decl *)
let rec megaMem decl = function
  A::l -> (customMem A decl) & (megaMem decl l)
| [] -> true;;

(* La fonction moinsGeneral compare deux pretypes *)
let rec existe hier decl fleche = function
  (a, t)::l -> (moinsGeneral hier decl (Arrow(Atom a, t), fleche)) or (existe hier decl fleche l)
| [] -> false
and moinsGeneral hier decl = function
  (Bool, Bool) -> true
| (Star, Star) -> true
| (Int, Int) -> true
| (Atom a, Atom b) -> smaller hier a b
| (Arrow(t1, t2), Arrow(u1, u2))
    -> (moinsGeneral hier decl (t2, u2)) & (moinsGeneral hier decl (u1, t1))
| (Method l, Method m)
    -> it_list (fun x y -> x & y) true (map (fun (a, t) -> existe hier decl (Arrow(Atom a, t)) l) m)
| _ -> false;;

(* wellformed controle les conditions de bonne formation des types *)

let rec wellformed hier decl =
    let  atomictypes = (map (fun(a, t) -> a) decl)      (*ensemble des types atomiques*)
in function
  Bool -> true
| Star -> true
| Int -> true
| Atom a -> true
| Arrow(t1,t2) -> (wellformed hier decl t1) & (wellformed hier decl t2)
| Method l -> let domain_de_l = (map (fun(a,t) -> a) l)
              in (covariance l) & (multinheritance domain_de_l domain_de_l)
```

```
(* il controle la condition de covariance *)

where rec covariance = function
 (a,t):: l1 -> (check a t l1) & (covariance l1)
| []  -> true

and check a t = function
  (b,s)::l2 -> (if (smaller hier a b) then moinsGeneral hier decl (t,s)
                else if (smaller hier b a) then moinsGeneral hier decl (s,t)
                else true) & (check a t l2)
| [] -> true


(* il controle l'heritage multiple pour tout couple du domain *)

and multinheritance domain = function
  a::l -> (apply domain a l) & (multinheritance domain l)
| [] -> true


(* il controle l'heritage multiple pour tout couple (a, reste_du_domain) *)

and apply domain a = function
  b::l -> let LBab = lowerbounds a b atomictypes
          in ((list_length LBab)=0 or (contains domain (maximalElements LBab)))
             & (apply domain a l)
| [] -> true

and lowerbounds a b = function
  c::l -> (if (smaller hier c a) & (smaller hier c b) then [c] else [])@(lowerbounds a b l)
| [] -> []


(* il donne la liste des elements maximaux *)

and maximalElements liste =
    let rec isMaximal a = function
          b::l -> ((b=a) or not(smaller hier a b)) & (isMaximal a l)
        | [] -> true
    and explore l1 = function
          a::l2 -> (if (isMaximal a l1) then [a] else [])@(explore l1 l2)
        | [] -> []
    in explore liste liste


(* il dit si la liste l1 contient la liste l2 *)

and contains l1 = function
 a::l2 -> (if (mem a l1) then true else false) & (contains l1 l2)
| [] -> true ;;
```

```
(* typexpression renvoie le type d'une expression. Genere une erreur en cas d'incompatibilites *)

let rec typexpression hier decl env = function
   Val n -> Int
 | Const x -> Bool
 | Quest -> Star
 | Epsilon -> Method []
 | Eqz -> Arrow(Int, Bool)
 | Pred -> Arrow(Int, Int)
 | Succ -> Arrow(Int, Int)
 | Var x -> (try assoc x env with Not_found
               -> failwith ("[type-checker] Pretype absent de l'environnement: variable \""
                           ^(x)^"\" non liee"))
 | Fix (s, t, f )
        -> if wellformed hier decl t then
              (if moinsGeneral hier decl ((typexpression hier decl ((s, t)::env) f), t) then t
               else failwith ("[type-checker] Le type "^(imprime t)^
                              " de la variable de recursion n'est pas "^
                              " compatible avec la definition"))
           else (failwith ("[type-checker] Le pretype "^(imprime t)^
                          " n'est pas bien forme"))
 | Lambda (s, t, f ) -> if wellformed hier decl t then Arrow(t, typexpression hier decl ((s, t)::env) f)
                        else (failwith ("[type-checker] Le pretype "^(imprime t)^
                                        " n'est pas bien forme"))
 | App (f, g) -> let v = typexpression hier decl env g
   in (match (typexpression hier decl env f) with
        Arrow(t, u) -> if moinsGeneral hier decl (v, t) then u
                       else failwith ("[type-checker] Impossible d'appliquer "^
                                      (imprime t)^" -> "^(imprime u)^" a "^
                                      (imprime v)^" dans "^
                                      (exprToString(App (f, g))))
      | Method l -> (match v with
                       Atom A -> assoc (findMin hier l A) l
                     | _ -> failwith ("[type-checker] Methode appliquee au pretype "
                                      ^(imprime v)^"!"))
      | _ as t -> failwith ("[type-checker] Le type "^(imprime t)^
                            " est utilise comme type fonctionnel!"))
 | Ampersend (f, l, g)
      -> if wellformed hier decl (Method l)  then
           (match (rev l) with
              (a,t)::l' ->
                     if
                       (moinsGeneral hier decl (typexpression hier decl env f , Method l')) &
                       (moinsGeneral hier decl (typexpression hier decl env g , Arrow(Atom a, t)))
                     then Method l
                     else failwith("[type-checker] Index incompatible: ...&"
                                   ^(imprime(Method l)))
            | []      -> failwith "[type-checker] Index vide?!?!")
         else failwith ("[type-checker] Le pretype "^(imprime (Method l))^
                        " n'est pas bien forme")
 | Coerce (a, f) -> let b = typexpression hier decl env f in
                       if moinsGeneral hier decl (b , Atom a) then (Atom a)
                       else failwith ("[type-checker] Coercion impossible:"^(imprime (Atom a))^
                                      " n'est pas plus grand que "^(imprime b))
```

```
| Super (a, f) -> let b = typexpression hier decl env f in
                    if moinsGeneral hier decl (b , Atom a) then (Atom a)
                    else failwith ("[type-checker] Super impossible:"^(imprime (Atom a))^
                              " n'est pas plus grand que "^(imprime b))
| In (a, f) -> let t = typexpression hier decl env f in
               let b = (try assoc a decl with Not_found
                         -> failwith ("[type-checker] Le type objet "
                                  ^a^" n'est pas defini!"))
               in if moinsGeneral hier decl (t, b)
                  then Atom a
                  else failwith ("[type-checker] `in` impossible: le type representaton de "^
                          (imprime (Atom a))^" n'est pas plus grand que "^(imprime t))
 | Out (a, f) -> let b = typexpression hier decl env f in
                   if moinsGeneral hier decl (b , Atom a)
                   then (try assoc a decl with Not_found
                          -> failwith ("[type-checker] Le type objet "
                                   ^a^" n'est pas defini!"))
                   else failwith ("[type-checker] Out impossible:"^(imprime (Atom a))^
                              " n'est pas plus grand que "^(imprime b))
| Ite (e1,e2,e3) -> if (typexpression hier decl env e1) = Bool then
                    let t2 = (typexpression hier decl env e2)
                    and t3 = (typexpression hier decl env e3) in
                       if moinsGeneral hier decl (t2, t3) then t3 else
                       if moinsGeneral hier decl (t3, t2) then t2 else
                       failwith("[type-checker] Types incompatibles "^
                               "dans ite dans l'expression "
                               ^(exprToString(Ite (e1,e2,e3))))
                    else failwith("[type-checker] le premier argument "^
                              "de ite n'est pas bool dans l'expression "
                              ^(exprToString (Ite(e1,e2,e3))));;


(* Mise a jour de la hierarchie des types lors d'une declaration "let A < A1, ... An". Il faut non seulement
   noter que A < A1 ... An  mais aussi en deduire les nouvelles relations qui en decoulent par transitivite.
   Deux types de cas se presentent :
   - si on declare a < b alors qu'on avait deja b < c, on doit noter a < c. C'est ce dont se charge updateDown
   - si on declare b < c alors qu'on avait deja a < b, on doit noter a < c. C'est ce dont se charge updateUp.
   La fonction updateHier, apres avoir verifie l'existence des types specifies, met a jour la hierarchie des
   types en appelant ces  deux sous-fonctions.*)


let updateDown hier A B =
  let greaterthanB = try assoc B hier with Not_found -> []
  in if customMem A hier
      then map (fun ((x, m) as elem) -> if x = A then (x, B::greaterthanB@m) else elem) hier
      else (A, B::greaterthanB)::hier;;


let rec updateUp A l = function
  (x, listex)::suite -> (x, listex@(if mem A listex then l else []))::(updateUp A l suite)
| [] -> [];;
```

```
let updateHier decl hier A l =
 if not (customMem A hier) then                    (* ici on controle si A a deja ete' declare' comme soustype *)
  if (customMem A decl) & (megaMem decl l) then    (* ici on controle si tous les types ont ete declares *)
    if it_list (fun x y -> x & y) true              (* controle des types representation *)
           (map (fun B -> moinsGeneral hier decl (assoc A decl, assoc B decl)) l)
    then
        let rec parcoursl h = function
          B::l -> parcoursl (updateDown h A B) l
          | [] -> h
        in updateUp A l (parcoursl hier l)
    else failwith "Sous-typage incompatible avec l'ordre sur les pretypes."
  else failwith ("Type objet non declare:  let "^A^"  < ...")
else failwith("Sous-typage pour "^A^" deja declare");;


(*pour faire marcher seulment le type-checker*)

let rec typeterme hier decl = function
  Expression e -> typexpression hier decl [] e
| Declaration (atomtyp, pretyp, term)
        -> if customMem atomtyp decl
              then failwith ("Le type "^atomtyp^" est deja declare")
              else typeterme hier ((atomtyp, pretyp)::decl) term
| Subtyping (atomtyp, l, term) -> typeterme (updateHier decl hier atomtyp l) decl term;;



(*______________________________________________________________________________*)
(*                                                                              *)
(*                               EVALUATEUR                                     *)
(*                                                                              *)
(*______________________________________________________________________________*)


(* L'evaluateur proprement dit se compose de deux sous-fonction recursives : evalterme et evalexpr,
   chargees d'evaluer respectivement termes et expressions. Elles prennent deux parametres : la hierarchie
   des types atomiques (de type (atomictype*(atomictype list)) list), et la liste des declarations de types
   atomiques (atomictype*pretype list).        *)


let eval = evalterme [] []

where rec evalterme hier decl = function
  Expression e -> (evalexpr hier decl e)
| Declaration (atomtyp, pretyp, term)
        -> if customMem atomtyp decl then failwith ("Le type "^atomtyp^" est deja declare")
           else evalterme hier ((atomtyp, pretyp)::decl) term
| Subtyping (atomtyp, l, term) -> evalterme (updateHier decl hier atomtyp l) decl term
```

```
(* pour effecter le type-checking nous on fait preceder l'interpretation de une expression par
                 (typexpression hier decl [] M)
   de facon que si le terme n'est pas bien type' la fonction typexpression souleve une failure  *)

and evalexpr hier decl M =
(match M with
  Epsilon -> Epsilon
| Var x -> failwith ("La variable "^x^" n'est pas definie.")
| Val n -> Val n
| Succ -> Succ
| Pred -> Pred
| Eqz -> Eqz
| Quest -> Quest
| Const x -> Const x
| Ite (e1, e2, e3) -> let b = (evalexpr hier decl e1) in (match b with
                          Const true    -> (evalexpr hier decl e2)
                        | Const false  -> (evalexpr hier decl e3)
                        | _ -> failwith "Type checker bug??")
| Lambda x -> Lambda x
| Fix(x, t, f) ->  evalexpr hier decl (remplace x (Fix(x, t, f)) f)
| Ampersend x -> Ampersend x
| Coerce (a, x) -> let x1 = (evalexpr hier decl x) in  Coerce (a, x1)
| Super (a, x) -> let x1 = (evalexpr hier decl x) in  Super (a, x1)
| In (a, x) -> In (a , x)
| Out (A, e) -> let e1 = evalexpr hier decl e
                  in (match e1 with
                        In (_, expr) -> (evalexpr hier decl expr)
                      | Coerce (_, expr) -> (evalexpr hier decl (Out (A, expr)))
                      | _ -> failwith ("J'attandais un tagged value et j'ai trouve' "
                                      ^(exprToString e1)))
| App (e, f) -> let e1 = evalexpr hier decl e
                    and f1 = evalexpr hier decl f
                    in (match e1 with
          Eqz -> (match f1 with
                      Val 0 -> Const true
                    | Val n -> Const false
                    | _  -> failwith "eqz: Type checker bug??")
        | Pred -> (match f1 with
                      Val 0 -> Val 0
                    | Val n -> Val (n-1)
                    | _  -> failwith " pred: Type checker bug??")
        | Succ -> (match f1 with
                      Val n -> Val (n+1)
                    | _ as y -> failwith "succ: Type checker bug??")
        | Lambda (x, t, M) -> evalexpr hier decl (remplace x f1 M)
        | Ampersend (M1, l, M2)
              -> let D' = findMin hier l (getTag f1)
                 and Dn = fst (dernier l)
                  in (match f1 with
                         Super (a, N) -> evalexpr hier decl
                                          (if D' = Dn then App(M2, N) else App(M1, f1))
                       | _ -> evalexpr hier decl (App ((if D' = Dn then M2 else M1), f1)))
        | _ -> failwith "Valeur non fonctionnelle utilisee comme fonction!"))
```

```
(* La fonction remplace prend en argument une expression et y remplace toutes les occurences de
   la variable x par l'expression e *)

and remplace x e = function
   Var y -> if y = x then e else Var y
| Lambda (s, t, f) -> Lambda (s, t, if s = x then f else (remplace x e f))
| Fix (s, t, f) -> Fix (s, t, if s = x then f else (remplace x e f))
| App (f, g) -> App (remplace x e f, remplace x e g)
| Ampersend (f, l, g) -> Ampersend (remplace x e f, l, remplace x e g)
| Coerce (a, f) -> Coerce (a, remplace x e f)
| Super (a, f) -> Super (a, remplace x e f)
| In (a, f) -> In (a, remplace x e f)
| Out (a, f) -> Out (a, remplace x e f)
| Ite (e1,e2,e3) -> Ite (remplace x e e1, remplace x e e2, remplace x e e3)
| _ as y -> y;;




(* pour evaluer a l'interieur des lambda et des In on a besoin d'une fonction remplace plus sofistique
   parce qu'il doit eviter la capture des varibles libres, et donc effectuer des alfa-conversions      *)


let internaleval = evalterme [] []

where rec evalterme hier decl = function
   Expression e -> ((typexpression hier decl [] e);  internalevalexpr hier decl e)
| Declaration (atomtyp, pretyp, term)
        -> if customMem atomtyp decl then failwith ("Le type "^atomtyp^" est deja declare")
           else evalterme hier ((atomtyp, pretyp)::decl) term
| Subtyping (atomtyp, l, term) -> evalterme (updateHier decl hier atomtyp l) decl term


and  internalevalexpr hier decl = function
   Epsilon -> Epsilon
| Var x -> Var x
| Val n -> Val n
| Succ -> Succ
| Pred -> Pred
| Eqz -> Eqz
| Quest -> Quest
| Const x -> Const x
| Fix(x, t, f) -> remplace x (Fix(x, t, f)) f
| Lambda (x, t, M) -> Lambda (x , t, (internalevalexpr hier decl M))
| Ampersend x -> Ampersend x
| Coerce (a, x) -> let x1 = (internalevalexpr hier decl x) in  Coerce (a, x1)
| Super (a, x) -> let x1 = (internalevalexpr hier decl x) in  Super (a, x1)
| In (a, x) -> In (a , internalevalexpr hier decl x)
| Ite (e1, e2, e3)
       -> let b = (internalevalexpr hier decl e1)
          in (match b with
                Const true   -> (internalevalexpr hier decl e2)
              | Const false  -> (internalevalexpr hier decl e3)
              | _ -> Ite (b,(internalevalexpr hier decl e2),(internalevalexpr hier decl e2)))
```

```
| Out (A, e) -> let e1 = internalevalexpr hier decl e
                    in (match e1 with
                      In (_, expr) -> (internalevalexpr hier decl expr)
                    | Coerce (_, expr) -> (internalevalexpr hier decl (Out (A, expr)))
                    | _ -> Out (A,e1))
| App (e, f) -> let e1 = internalevalexpr hier decl e
                    and f1 = internalevalexpr hier decl f
                    in  (match e1 with
          Eqz -> (match f1 with
                    Val 0 -> Const true
                  | Val n -> Const false
                  | _  -> App (e1,f1))
        | Pred -> (match f1 with
                    Val 0 -> Val 0
                  | Val n -> Val (n-1)
                  | _  -> App (e1,f1))
        | Succ -> (match f1 with
                    Val n -> Val (n+1)
                  | _  -> App (e1,f1))
        | Lambda (x, t, M) -> internalevalexpr hier decl (remplace x f1 M)
        | Ampersend (M1, l, M2) ->
              let D' = findMin hier l (getTag f1)
              and Dn = fst (dernier l)
               in (match f1 with
                      Super (a, N) -> internalevalexpr hier decl
                                          (if D' = Dn then App(M2, N) else App(M1, f1))
                    | _ -> internalevalexpr hier decl (App ((if D' = Dn then M2 else M1), f1)))
        | _ -> App (e1,f1))


and union = function
   (l, []) -> l
| ([], l) -> l
| (x::l1, l) -> if (mem x l ) then union(l1,l) else union(l1,x::l)


and diff = function
   ([], x) -> []
| (y::l, x) -> if (x=y) then l else (y::(diff(l,x)))


and FV = function
   Var x -> [x]
| Ite (e1, e2, e3) -> union(FV(e1), union(FV(e2),FV(e3)))
| Fix (x, t, e) -> diff(FV(e), x)
| Lambda (x, t, e) -> diff(FV(e), x)
| Ampersend (e1, l, e2)  ->  union(FV(e1),FV(e2))
| Coerce (a, e) -> FV(e)
| Super (a, e) -> FV(e)
| In (a, e) ->  FV(e)
| Out (A, e) -> FV(e)
| App (e1, e2) ->  union(FV(e1),FV(e2))
| _ -> []
```

```
and remplace x e = function
   Var y -> if y = x then e else Var y
 | Fix (y, t, f) -> if  y = x then  Fix (y, t, f)
                        else if not(mem x (FV(f))) then Fix (y, t, remplace x e f)
                        else remplace x e (Fix(y^"a", t, remplace y (Var(y^"a")) f))
 | Lambda (y, t, f) -> if  y = x then  Lambda (y, t, f)
                        else if not(mem x (FV(f))) then Lambda (y, t, remplace x e f)
                        else remplace x e (Lambda(y^"a", t, remplace y (Var(y^"a")) f))
 | App (f, g) -> App (remplace x e f, remplace x e g)
 | Ampersend (f, l, g) -> Ampersend (remplace x e f, l, remplace x e g)
 | Coerce (a, f) -> Coerce (a, remplace x e f)
 | Super (a, f) -> Super (a, remplace x e f)
 | In (a, f) -> In (a, remplace x e f)
 | Out (a, f) -> Out (a, remplace x e f)
 | Ite (e1,e2,e3) -> Ite (remplace x e e1, remplace x e e2, remplace x e e3)
 | _ as y -> y;;


(* Les fonctions exportees*)


let ParseString x = parseProgram (lexer(stream_of_string x));;

let ParseFile x =  (parseProgram(lexer (stream_of_string x)));;

let CalcString x = try   let syntaxtree = (parseProgram(lexer (stream_of_string x)))
                      in print_string ("- : "^imprime(typeterme [] [] syntaxtree)^"\n");
                         prettyprint(eval (syntaxtree))
                 with Failure s -> print_string ("Erreur : "^s^"\n");;

let TypeString x = try print_string (imprime(typeterme [] [] (parseProgram(lexer (stream_of_string x)))))
                 with Failure s -> print_string ("Erreur : "^s^"\n");;


let TypeFile x =
     try print_string (imprime(typeterme [] [] (parseProgram(lexer (stream_of_channel (open_in x))))))
     with Failure s -> print_string ("Erreur : "^s^"\n");;


let CalcFile x = try let syntaxtree = (parseProgram(lexer (stream_of_channel (open_in x))))
                      in  print_string ("- : "^imprime(typeterme [] [] syntaxtree)^"\n");
                          prettyprint(eval (syntaxtree))
                 with Failure s -> print_string ("Erreur : "^s^"\n");;

let InternalCalcString x = try let syntaxtree = (parseProgram(lexer (stream_of_string x)))
                              in  print_string ("- : "^imprime(typeterme [] [] syntaxtree)^"\n");
                                  prettyprint(internaleval (syntaxtree))
                           with Failure s -> print_string ("Erreur : "^s^"\n");;

let InternalCalcFile x = try  let syntaxtree = (parseProgram(lexer (stream_of_channel (open_in x))))
                                  in  print_string ("- : "^imprime(typeterme [] [] syntaxtree)^"\n");
                                      prettyprint(internaleval (syntaxtree))
                              with Failure s -> print_string ("Erreur : "^s^"\n");;
```

```
(*_________________________________________________________________*)
(*                                                                 *)
(*                  le ficher lambdaobject.mli                     *)
(*                                                                 *)
(*_________________________________________________________________*)

type
  pretype = Bool | Star | Int | Atom of atomictype
          | Arrow of pretype*pretype
          | Method of (atomictype*pretype) list
and
  atomictype == string;;

type
  expression = Var of string
             | Quest
             | Const of bool
             | Val of int
             | Succ
             | Pred
             | Eqz
             | Lambda of string*pretype*expression
             | Fix of string*pretype*expression
             | Ite of expression*expression*expression
             | App of expression*expression
             | Epsilon
             | Ampersend of expression*((atomictype*pretype) list)*expression
             | Coerce of atomictype*expression
             | Super of atomictype*expression
             | In of atomictype*expression
             | Out of atomictype*expression
and
  terme = Expression of expression
             | Subtyping of atomictype*(atomictype list)*terme
             | Declaration of atomictype*pretype*terme;;


value ParseString : string -> terme ;;
value ParseFile : string -> terme ;;
value TypeString : string -> unit ;;
value TypeFile : string -> unit ;;
value CalcString : string -> unit ;;
value CalcFile : string -> unit ;;
value InternalCalcString : string -> unit ;;
value InternalCalcFile : string -> unit ;;
```

Appendix B

Type system of λ_object

B.1 Types

1. $A \in_{C,S}$ **Types** for each $A \in dom(S)$
2. if $T_1, T_2 \in_{C,S}$ **Types** then $T_1 \to T_2 \in_{C,S}$ **Types** and $T_1 \times T_2 \in_{C,S}$ **Types**
3. if for all $i, j \in I$

   (a) $(D_i, T_i \in_{C,S}$ **Types**$)$
   (b) if $C \vdash D_i \leq D_j$ then $C \vdash T_i \leq T_j)$
   (c) for all maximal type D in $LB_C(\{D_i, D_j\})$ there exists $h \in I$ such that $D_h = D$
   (d) if $i \neq j$ then $D_i \neq D_j$

   then $\{D_i \to T_i\}_{i \in I} \in_{C,S}$ **Types**

B.2 Typing rules

[NewType] $$\frac{C\,,\ S[A \leftarrow T] \vdash P\colon U}{C, S \vdash \textbf{let } A \textbf{ hide } T \textbf{ in } P\colon U}$$

$A \notin dom(S), T \in_{C,S}$ **Types** and T not atomic

[Constraint] $$\frac{C \cup (A \leq A_i), S \vdash P\colon T}{C, S \vdash \textbf{let } A \leq A_1, \ldots, A_n \textbf{ in } P\colon T}$$

if $C \vdash S(A) \leq S(A_i)$ and A do not appear in C

[Taut] $$C, S \vdash x^T\colon T$$

[$\to$ Intro] $$\frac{C, S \vdash M\colon T'}{\lambda x^T.M\colon T \to T'}$$ $T \in_{C,S}$ **Types**

[$\to$ Elim$_{(\leq)}$] $$\frac{C, S \vdash M\colon U \to T \quad N\colon W}{C, S \vdash M\,N\colon T}$$ $C \vdash W \leq U$

[TAUT$_\varepsilon$] $$C,S \vdash \varepsilon\colon \{\}$$

[{}INTRO+] $$\frac{C,S \vdash M\colon W_1 \leq \{U_i \to V_i\}_{i\in I} \quad C,S \vdash N\colon W_2 \leq U \to V}{C,S \vdash (M\&^{\{U_i\to V_i\}_{i\in I}\oplus(U\to V)}N)\colon \{U_i \to V_i\}_{i\in I} \oplus (U \to V)} \quad \{U_i \to V_i\}_{i\in I} \oplus (U \to V) \in_{C,S} \mathbf{Types}$$

The rules for the expressions that do not belong to the syntax of λ& are:

[{}ELIM] $$\frac{C,S \vdash M\colon \{U_i \to T_i\}_{i\in I} \quad C,S \vdash N\colon U}{C,S \vdash M \bullet N\colon T_j} \quad U_j = \min_{i\in I}\{U_i | C \vdash U \leq U_i\}$$

[PAIR] $$\frac{C,S \vdash M\colon T_1 \quad C,S \vdash N\colon T_2}{C,S \vdash \langle M, N\rangle\colon T_1 \times T_2}$$

[PROJ] $$\frac{C,S \vdash M\colon T_1 \times T_2}{C,S \vdash \pi_i(M)\colon T_i} \quad \text{for } i = 1,2$$

[COERCE] $$\frac{C,S \vdash M\colon B}{C,S \vdash \mathbf{coerce}^A(M)\colon A} \quad C \vdash B \leq A \text{ and } A \in_{C,S} \mathbf{Types}$$

[SUPER] $$\frac{C,S \vdash M\colon B}{C,S \vdash \mathbf{super}^A(M)\colon A} \quad C \vdash B \leq A \text{ and } A \in_{C,S} \mathbf{Types}$$

[IN] $$\frac{C,S \vdash M\colon T}{C,S \vdash in^A(M)\colon A} \quad C \vdash T \leq S(A) \text{ and } A \in_{C,S} \mathbf{Types}$$

[OUT] $$\frac{C,S \vdash M\colon B}{C,S \vdash out^A(M)\colon S(A)} \quad C \vdash B \leq A \text{ and } A \in_{C,S} \mathbf{Types}$$

[FIX] $$\frac{C,S \vdash M\colon T}{\mu x^T.M\colon T} \quad T \in_{C,S} \mathbf{Types}$$

Appendix C

Specification of the toy language

C.1 Terms

$$
\begin{array}{rcl}
r & ::= & \{\ell_1\texttt{=}exp_1; \ldots ; \ell_n\texttt{=}exp_n\} \\
\\
exp & ::= & x \\
 & | & \texttt{fn(}x_1 : T_1, ..., x_n : T_n\texttt{) => } exp \\
 & | & exp_1\texttt{(}exp_2\texttt{)} \\
 & | & \texttt{(}exp, \ldots , exp\texttt{)} \\
 & | & \texttt{fst(}exp\texttt{)} \mid \texttt{snd(}exp\texttt{)} \\
 & | & \texttt{let } x{:}T \texttt{ = } exp \texttt{ in } exp \\
 & | & \texttt{extend } classname \\
 & & \quad (message \texttt{ = } method\texttt{;})^{+} \\
 & & \quad interface \\
 & & \texttt{in } exp \\
 & | & \texttt{new(}classname\texttt{)} \\
 & | & \texttt{self} \\
 & | & \texttt{(self.}\ell\texttt{)} \\
 & | & \texttt{(update } r\texttt{)} \\
 & | & \texttt{super[}A\texttt{](}exp\texttt{)} \\
 & | & \texttt{coerce[}A\texttt{](}exp\texttt{)} \\
 & | & \texttt{\& fn(}\texttt{x}_1{:}A_1, ..., \texttt{x}_{n_1} : A_{n_1}\texttt{) => } exp_1 \\
 & & \texttt{\& fn(}\texttt{x}_1{:}A_1, ..., \texttt{x}_{n_2} : A_{n_2}\texttt{) => } exp_2 \\
 & & \qquad \vdots \\
 & & \texttt{\& fn(}\texttt{x}_1 : A_1\texttt{, ... ,}x_{n_m} : A_{n_m}\texttt{) => } exp_m \qquad {\scriptstyle (m \geq 1)} \\
 & | & \texttt{[} exp_0 exp \ exp_1 \ldots exp_n \texttt{]} \qquad {\scriptstyle (n \geq 0)}
\end{array}
$$

$$
\begin{array}{rcl}
p & ::= & exp \\
 & | & \texttt{class}\ classname\ [\ \texttt{is}\ classname\ (\ ,\ classname\ )^*] \\
 & & \quad instanceVariables \\
 & & \quad (message\ \texttt{=}\ method\texttt{;}\ )^* \\
 & & \quad interface \\
 & & \texttt{in}\ p \\
 & & \\
method & ::= & exp \\
 & & \\
message & ::= & x \\
 & & \\
interface & ::= & \texttt{[[}message : V ; ... ; message : V\texttt{]]} \\
 & & \\
instanceVariables & ::= & \{\ell_1 : T_1\texttt{=}exp_1; \ldots ; \ell_n : T_n\texttt{=}exp_n\}
\end{array}
$$

C.2 Subtyping

$$C \cup (A_1 \leq A_2) \vdash A_1 \leq A_2$$

$$\frac{C \vdash T_2 \leq T_1 \quad C \vdash U_1 \leq U_2}{C \vdash T_1 \rightarrow U_1 \leq T_2 \rightarrow U_2}$$

$$\frac{C \vdash U_1 \leq T_1\ \ldots C \vdash U_n \leq T_n}{C \vdash\ (U_1 \times \ldots \times U_n) \leq (T_1 \times \ldots \times T_n)}$$

$$\frac{\text{for all } i \in I \text{, there exists } j \in J \text{ such that } C \vdash D''_i \leq D'_j \text{ and } C \vdash U'_j \leq U''_i}{C \vdash \{D'_j \rightarrow U'_j\}_{j \in J} \leq \{D''_i \rightarrow U''_i\}_{i \in I}}$$

$$\frac{C \vdash U_1 \leq T_1\ \ldots C \vdash U_k \leq T_k}{C \vdash \langle\!\langle \ell_1 \colon U_1; \ldots ; \ell_k \colon U_k; \ldots ; \ell_{k+j} \colon U_{k+j} \rangle\!\rangle \leq \langle\!\langle \ell_1 \colon T_1; \ldots ; \ell_k \colon T_k \rangle\!\rangle}$$

The (pre)order for all types is given by the reflexive and transitive closure of the rules above.

C.2.1 Auxiliary Notation

$$C \vdash \langle\!\langle \ell_1 \colon T_1; \ldots ; \ell_k \colon T_k; \ldots ; \ell_{k+j} \colon T_{k+j} \rangle\!\rangle \leq_{strict} \langle\!\langle \ell_1 \colon T_1; \ldots ; \ell_k \colon T_k \rangle\!\rangle$$

$$\frac{C \vdash U_1 \leq T_1\ \ldots C \vdash U_k \leq T_k}{C \vdash \langle\!\langle \ell_1 \colon U_1; \ldots ; \ell_k \colon U_k \rangle\!\rangle \Subset \langle\!\langle \ell_1 \colon T_1; \ldots ; \ell_k \colon T_k; \ldots ; \ell_{k+j} \colon T_{k+j} \rangle\!\rangle}$$

C.3 Typing Rules

Let

$$\Gamma\colon (Vars \cup \{\texttt{self}\}) \to \mathbf{Types}$$
$$S\colon \mathbf{AtomicTypes} \to \mathbf{RecordTypes}$$

Then we have the following typing rules:

[Taut] $$C;S;\Gamma \vdash x : \Gamma(x) \qquad x \in (Vars \cup \{\texttt{self}\})$$

[Funct] $$\frac{C;S;\Gamma[x \leftarrow T] \vdash exp\colon U}{C;S;\Gamma \vdash \texttt{fn(}x\colon T\texttt{)\ =>}\ exp : T \to U} \qquad \text{if } T \in_C \mathbf{Types}$$

[Appl] $$\frac{C;S;\Gamma \vdash exp_1\colon T \to U \qquad C;S;\Gamma \vdash exp_2\colon W}{C;S;\Gamma \vdash exp_1(exp_2) : U} \qquad \text{if } C \vdash W \leq T$$

[Prod] $$\frac{C;S;\Gamma \vdash exp_1\colon T_1 \qquad \ldots \qquad C;S;\Gamma \vdash exp_n\colon T_n}{C;S;\Gamma \vdash \texttt{(}exp_1\texttt{,}\ldots\texttt{,}exp_n\texttt{)}\colon (T_1 \times \ldots \times T_n)}$$

[Record] $$\frac{C;S;\Gamma \vdash exp_1\colon T_1 \qquad \ldots \qquad C;S;\Gamma \vdash exp_n\colon T_n}{C;S;\Gamma \vdash \{\ell_1 = exp_1\texttt{;}\ldots\texttt{;}\ell_n = exp_n\} : \langle\!\langle \ell_1 : T_1; \ldots; \ell_n : T_n \rangle\!\rangle}$$

[Let] $$\frac{C;S;\Gamma \vdash exp'\colon W \qquad C;S;\Gamma[x \leftarrow T] \vdash exp\colon U}{C;S;\Gamma \vdash \texttt{let}\ x : T\ \texttt{=}\ exp'\ \texttt{in}\ exp : U} \qquad \text{if } C \vdash W \leq T$$

[New] $$C;S;\Gamma \vdash \texttt{new(}A\texttt{)}\colon A \qquad \text{if } A \in dom(S)$$

[Read] $$C;S;\Gamma \vdash \texttt{self.}\ell\colon T \qquad \text{if } S(\Gamma(\texttt{self})) = \langle\!\langle \ldots\ell\colon T\ldots \rangle\!\rangle$$

[Write] $$\frac{C;S;\Gamma \vdash r\colon R}{C;S;\Gamma \vdash \texttt{(update}\ r\texttt{)} : \Gamma(\texttt{self})} \qquad \text{if } C \vdash R \Subset S(\Gamma(\texttt{self}))$$

[OvAbst] $$\frac{C;S;\Gamma \vdash exp_1\colon T_1 \ldots C;S;\Gamma \vdash exp_n\colon T_n}{C;S;\Gamma \vdash\ \texttt{\&}exp_1\texttt{\&}\ldots\texttt{\&}\ exp_n\colon \{T_1\texttt{,}\ \ldots\ \texttt{,}T_n\}} \quad \{T_1\texttt{,}\ \ldots\ \texttt{,}T_n\} \in_C \mathbf{Types}$$

[OvAppl] $$\frac{C;S;\Gamma \vdash exp\colon \{D_i \to T_i\}_{i\in I} \qquad C;S;\Gamma \vdash exp_j\colon A_j \quad (j{=}0..n)}{C;S;\Gamma \vdash \texttt{[}exp_0\ exp\ \ exp_1,\ldots,exp_n\ \texttt{]}\colon T_h}$$
$$\text{if } D_h = \min_{i\in I}\{D_i \mid C \vdash A_0\texttt{x}A_1\texttt{x}\ldots A_n \leq D_i\}.$$

[Coerce] $$\frac{C;S;\Gamma \vdash exp\colon A}{C;S;\Gamma \vdash \texttt{coerce[}A'\texttt{](}exp\texttt{)}\colon A'} \qquad \text{if } C \vdash A \leq A'$$

[Super] $$\frac{C;S;\Gamma \vdash exp\colon A}{C;S;\Gamma \vdash \texttt{super[}A'\texttt{](}exp\texttt{)}\colon A'} \qquad \text{if } C \vdash A \leq A'$$

[Multi] $$\frac{C;S;\Gamma \vdash exp_1 : T_1 \ldots C;S;\Gamma \vdash exp_n : T_n}{C;S;\Gamma \vdash \texttt{\&}exp_1\texttt{\&}\ldots\texttt{\&}\ exp_n : \{T_1\texttt{,}\ \ldots\ \texttt{,}T_n\}} \qquad \{T_1, \ldots ,T_n\} \in_C \mathbf{Types}$$

[Extend] $$\frac{C;S;\Gamma'[\texttt{self} \leftarrow A] \vdash exp_j : V_j \ \ (j{=}1..k) \qquad C;S;\Gamma' \vdash exp : T}{C;S;\Gamma \vdash \texttt{extend}\ A\ \ m_1\texttt{=}exp_1\texttt{;}\ldots\texttt{;}m_k\texttt{=}exp_k\ \ \texttt{[[}m_1 \texttt{:} V_1\texttt{,}\ldots\texttt{,}m_k\texttt{:}V_k\texttt{]]}\ \texttt{in}\ \ exp : T}$$

$A \in dom(S)$ and for $i = 1..k$ $\Gamma(m_i) \cup \{A \rightsquigarrow V_i\} \in_C$ **Types**

[Class] $$\frac{C;S;\Gamma \vdash r : R \qquad C';S';\Gamma'[\texttt{self} \leftarrow A] \vdash exp_j : V_j \ \ (j{=}1..k) \qquad C';S';\Gamma' \vdash p : T}{C;S;\Gamma \vdash \texttt{class}\ A\ \ \texttt{is}\ \ A_1\texttt{,}\ldots\texttt{,}A_n\ r : R\ \ m_1\texttt{=}exp_1\texttt{;}\ldots\texttt{;}m_k\texttt{=}exp_k\ \ I\ \ \texttt{in}\ \ p : T}$$

if $A \notin dom(S)$, for $i = 1..n$ $C \vdash R \leq_{strict} S(A_i)$ and for $i = 1..k$ $\Gamma(m_i) \cup \{A \rightsquigarrow V_i\} \in_{C'}$ **Types**

Where:

- $A \rightsquigarrow V = \begin{cases} \{(A \times D_i) \rightarrow U_i\}_{i\in I} & \text{if } V \equiv \texttt{\#}\{D_i \rightarrow U_i\}_{i\in I} \\ \{A \rightarrow V\} & \text{otherwise} \end{cases}$
- $S' \equiv S[A \leftarrow R]$
- $C' \equiv C \cup (\bigcup_{i=1..n} A \leq A_i)$
- $I \equiv\ \texttt{[[}m_1 : V_1,\ldots,m_m : V_m\ \texttt{]]}$
- $\Gamma' \equiv \Gamma[m_i \leftarrow \Gamma(m_i) \cup \{A \rightsquigarrow V_i\}]_{i=1..m}$

Appendix D

Proof of theorem 5.3.8

We prove the theorem only for the case in which there are no mutaully recursive methods; recursive terms do not pose any problem from the viewpoint of type-checking, but render the proof more unreadeable. The proof goes by induction on p. When p is formed only by an expression then the part 1 of the theorem is trivially proved by $[\text{TAUT}_{\varepsilon}]$. Thus in the rest of the proof we will prove the the part 1 of the theorem only when when p is is of the form `class` A `is` $\ldots$`[[`$\ldots$`]]` .

1. $p \equiv x$ but then $\Im[\![x]\!]_{\Gamma\, I\, \Gamma(self)} = x^{\Gamma(x)} : \Gamma(x)$ thus we have the result.

2. $p \equiv exp_1$`(`exp_2`)` then $C;S;\Gamma \vdash exp_1 : T_1 \to T$ and $C;S;\Gamma \vdash exp_2 : T_2 \leq T_1$. By induction $C;S \vdash \Im[\![exp_1]\!]_{\Gamma\, I\, \Gamma(self)} : T_1 \to T$ and $C;S \vdash \Im[\![exp_2]\!]_{\Gamma\, I\, \Gamma(self)} : T_2$. We obtain the thesis by $[\to\text{ELIM}_{(\leq)}]$.

3. $p \equiv$(`fn` $x{:}T_1$ `=>`exp) then $C;S;\Gamma[x \leftarrow T_1] \vdash exp{:}T_2$ where $T \equiv T_1 \to T_2$. By induction $C;S \vdash \Im[\![exp]\!]_{\Gamma[x\leftarrow T_1]\, I\, \Gamma(self)} : T_2$. Therefore $C;S \vdash \lambda x^{T_1}.\Im[\![exp]\!]_{\Gamma[x\leftarrow T_1]\, I\, \Gamma(self)} : T_1 \to T_2$.

4. $p \equiv$ (`let` $x{:}T_1$ `=` exp `in` exp'); combine the techniques of the previous two cases.

5. $p \equiv$ `snd(`exp`)` a straightforward use of the induction hypothesis.

6. $p \equiv$ `fst(`exp`)` a straightforward use of the induction hypothesis.

7. $p \equiv$ `new(`A`)`. By hypothesis $I(A){:}\,S(A)$ therefore $in^A(I(A))$ is well typed and has type A.

8. $p \equiv$ `[`exp_0 exp $exp_1, \ldots, exp_n$ `]` then $C;S;\Gamma \vdash exp : \{D_i \to T_i\}_{i\in I}$ and $C;S;\Gamma \vdash exp_i : A_i$ with $D_j = \min_{i\in I}\{D_i | C \vdash A_0 \times ... \times A_n \leq D_i\}$ and $T \equiv T_j$. From the induction hypothesis $C;S \vdash \Im[\![exp]\!]_{\Gamma\, I\, \Gamma(self)} : \{D_i \to T_i\}_{i\in I}$ and $C;S \vdash \Im[\![\texttt{(}exp_0, exp_1, ..., exp_n\texttt{)}]\!]_{\Gamma\, I\, \Gamma(self)} : A_0 \times ... \times A_n$. Then the thesis is obtained by $[\{\}\text{ELIM}]$.

9. $p \equiv$ `coerce[`A`](`exp`)` thus $T \equiv A$ and $C;S;\Gamma \vdash exp : T_1 \leq A$. By induction hypothesis $C;S \vdash \Im[\![exp]\!]_{\Gamma\, I\, \Gamma(self)} : T_1$. Thus $\mathbf{coerce}^A(\Im[\![exp]\!]_{\Gamma\, I\, \Gamma(self)})$ is well-typed and has type A.

10. $p \equiv$ `super[` A`](`exp`)`. As the previous case.

11. $p \equiv \texttt{self}$ straightforward

12. $p \equiv (\texttt{self}.\ell)$ Then $S(\Gamma(\texttt{self})) = \langle\!\langle ...\ell{:}\,T...\rangle\!\rangle$.
Since $\mathit{self}^{\Gamma(self)}{:}\,\Gamma(\mathit{self})$ and $\mathit{out}^{\Gamma(self)}{:}\,\Gamma(\mathit{self}) \to S(\Gamma(\mathit{self}))$ then $\mathit{out}^{\Gamma(self)}(\mathit{self}^{\Gamma(self)}) : \langle\!\langle ...\ell{:}\,T...\rangle\!\rangle$. Thus $(\mathit{out}^{\Gamma(self)}(\mathit{self}^{\Gamma(self)})).\ell{:}\,T$.

13. $p \equiv \texttt{update}(r)$ Then $T \equiv \Gamma(\mathit{self})$, $C; S; \Gamma \vdash r : R$ and $C \vdash S(\Gamma(\mathit{self})) \leq R$. If $r \equiv \{\ell_1 = \mathit{exp}_1; ...; \ell_n = \mathit{exp}_n\}$ then by induction hypothesis

$$C; S \vdash \langle \ell_1 = \Im[\![\mathit{exp}_1]\!]_{\Gamma\, I\, \Gamma(self)}; ...; \ell_n = \Im[\![\mathit{exp}_n]\!]_{\Gamma\, I\, \Gamma(self)} \rangle{:}\, R$$

By definition $\mathit{out}^{\Gamma(self)}(\mathit{self}^{\Gamma(self)}){:}\, S(\Gamma(\mathit{self}))$. Since $C \vdash S(\Gamma(\mathit{self})) \leq R$ then $(\langle \mathit{out}^{\Gamma(self)}(\mathit{self}^{\Gamma(self)}) \leftarrow \ell_1 = \Im[\![\mathit{exp}_1]\!]_{\Gamma\, I\, \Gamma(self)} ... \leftarrow \ell_n = \Im[\![\mathit{exp}_n]\!]_{\Gamma\, I\, \Gamma(self)} \rangle)$ is well typed and has type $S(\Gamma(\mathit{self}))$.
Therefore also $\Im[\![p]\!]_{\Gamma\, I\, \Gamma(self)} \equiv \mathit{in}^{\Gamma(self)}(\langle \mathit{out}^{\Gamma(self)}(\mathit{self}^{\Gamma(self)}) \leftarrow ...\rangle)$ is well-typed and has type $\Gamma(\mathit{self})$.

14. We prove w.l.o.g. the case for `extend` with only one multi-method: the case with ordinary methods is a slight modification of this case that can be deduced from the next case; extensions including more than one method can be translated in a suite of extensions with only one method, since, we recall, we do not consider the case of mutually recursive methods.

Let $p \equiv \texttt{extend}\ A\ \texttt{m=\&}\mathit{exp}_1 \ldots \texttt{\&}\mathit{exp}_n\ \texttt{in [[m:}\,V\texttt{]] in}\ \mathit{exp}$
where $V \equiv \#\{D_1 \to T_1, \ldots, D_n \to T_n\}$
and $D_i \equiv A^i_1 \times \ldots \times A^i_{n_i}$
and $\mathit{exp}_i \equiv \texttt{fn(}x^i_1{:}.A^i_1, \ldots, x^i_{n_i}{:}.A^i_{n_i}\texttt{) =>}\ \mathit{exp}'_i$ (for $i = 1..n$)

Let exp^*_1 denote the following expression:

$$\texttt{fn(self:}\,A,\ x^i_1{:}.A^i_1, \ldots, x^i_{n_i}{:}.A^i_{n_i}\texttt{) =>}\ \mathit{exp}'_i$$

Then p is translated into:

$$(\lambda m^{\Gamma(m)} \&^{\Gamma(m)\oplus\{A \rightsquigarrow V\}}.\Im[\![\mathit{exp}]\!]_{\Gamma\, I\, \Gamma(self)})$$
$$(\cdots(m^{\Gamma(m)} \&^{\Gamma(m)\oplus\{A\times D_{\sigma(1)}\to T_{\sigma(1)}\}} \Im[\![\mathit{exp}^*_1]\!]_{\Gamma\, I\, B})$$
$$\vdots$$
$$\&^{(\Gamma(m)\oplus\{A\times D_{\sigma(1)}\to T_{\sigma(1)}\}\oplus ... \oplus\{A\times D_{\sigma(n-1)}\to T_{\sigma(n-1)}\})\oplus\{A\times D_{\sigma(n)}\to T_{\sigma(n)}\}} \Im[\![\mathit{exp}^*_n]\!]_{\Gamma\, I\, B})$$

By hypothesis

$$C; S; \Gamma[\mathit{self} \leftarrow A] \vdash \texttt{\&}\mathit{exp}_1 ... \texttt{\&}\mathit{exp}_n{:}\, V$$

and thus it is clear that

$$C; S; \Gamma[\mathit{self} \leftarrow A] \vdash \texttt{\&}\mathit{exp}_{\sigma(1)} ... \texttt{\&}\mathit{exp}_{\sigma(n)}{:}\, V$$

Therefore

$$C; S; \Gamma \vdash \texttt{\&}\mathit{exp}^*_{\sigma(1)} ... \texttt{\&}\mathit{exp}^*_{\sigma(n)}{:}\, A \rightsquigarrow V \tag{D.1}$$

Also by hypothesis

$$C;S;\Gamma[m \leftarrow \Gamma(m) \cup \{A \rightsquigarrow V\}] \vdash exp: T \tag{D.2}$$

Note now that given an overloaded type V if $V \cup \{S \to T\}$ is a well formed overloaded type then

1. Also $V \oplus \{S \to T\}$ is well formed
2. $V \cup \{S \to T\} = V \oplus \{S \to T\}$

Thus from (D.2) we obtain

$$C;S;\Gamma[m \leftarrow \Gamma(m) \oplus \{A \rightsquigarrow V\}] \vdash exp: T$$

We can now apply the induction hypothesis obtaining:

$$C;S \vdash \lambda m^{\Gamma(m)\oplus\{A\rightsquigarrow V\}}.\Im[\![exp]\!]_{\Gamma\,I\,\Gamma(self)}: (\Gamma(m) \oplus \{A \rightsquigarrow V\}) \to T$$

Thus the thesis holds if we prove that

$$(\cdots(m\&^{\Gamma(m)\oplus\ldots}\ldots\&^{(\Gamma(m)\oplus\{A\times D_{\sigma(1)}\to T_{\sigma(1)}\}\oplus\ldots)\oplus\{A\times D_{\sigma(n)}\to T_{\sigma(n)}\}}\ldots): \Gamma(m) \oplus \{A \rightsquigarrow V\}$$

This can be proved by induction on n: for $n = 1$ the thesis is a strightforward application of the induction hypothesis on exp_1^* for (D.1). Consider now

$$(\cdots(m\&^{\Gamma(m)\oplus\ldots}\ldots\&^{(\Gamma(m)\oplus\{A\times D_{\sigma(1)}\to T_{\sigma(1)}\}\oplus\ldots)\oplus\{A\times D_{\sigma(i)}\to T_{\sigma(i)}\}}\Im[\![exp_i^*]\!]_{\Gamma\,I\,B})$$

Using the induction hypothesis on (D.1) it easy to see that the thesis fails only if $\Gamma(m) \oplus \{A \times D_{\sigma(1)} \to T_{\sigma(1)}\} \oplus \ldots \oplus \{A \times D_{\sigma(i)} \to T_{\sigma(i)}\}$ is not a well formed overloaded type. But since $\Gamma(m) \oplus \{A \rightsquigarrow V\}$ is well-formed, thus the previous type (which is a "subset" of this) surely satisfies the conditions of covariance and input type uniqueness. And thanks to the definition of σ it also satisfies the condition of multiple inheritance: if $A \times D_{\sigma(i)}$ has a strict lower bound in common with any other input type, then all the branches with maximal input types (which must already be in $\Gamma(m) \oplus \{A \rightsquigarrow V\}$) are already in $\Gamma(m) \oplus \{A \times D_{\sigma(1)} \to T_{\sigma(1)}\} \oplus \ldots \oplus \{A \times D_{\sigma(i)} \to T_{\sigma(i)}\}$, for either they are in $\Gamma(m)$ or they are of the form $\{A \times D_{\sigma(j)} \to T_{\sigma(j)}\}$ but then, because of the condition on σ, we have $\sigma(j) < \sigma(i)$.

15. As in the previous case we consider a simpler version where we have only one ancestor and one method in the class declaration: the general case can be obtained by adding some indexing in the right places.

    $p \equiv$`class` A `is` A' $r{:}\,R$ $m = exp$ `[[`$m{:}\,V$`]]` `in` p'. This is the only case where the proof of the first part of the theorem is non-trivial thus:

    1. We have to prove that for all $\overline{m} \in Vars$

    $$C \cup C_p; S \vdash \mathcal{M}[\![p]\!]_{\Gamma\,I\,\Gamma(self)}(\overline{m}): \mathcal{T}[\![p]\!](\overline{m})$$

    If $\overline{m} \not\equiv m$ then the thesis follows from the induction hypothesis. Otherwise let first consider the case when $m = exp$ is not a multi-method; then $\mathcal{T}[\![p]\!](m) =$

$\mathcal{T}[\![p']\!](m) \oplus \{A \to V\}$. Since p is well-typed then it is easy to prove that $\mathcal{T}[\![p]\!](m)$ is a well-formed type; moreover it holds that
$\mathcal{M}[\![p]\!]_{\Gamma\, I\, \Gamma(self)}(m) = (\mathcal{M}[\![p']\!]_{\Gamma'\, I\, \Gamma(self)}(m) \&^{(\mathcal{T}[\![p']\!](m))\oplus\{A\to V\}} \lambda self^A.\Im[\![exp]\!]_{\Gamma\, I[A\leftarrow r]\, A})$.
By induction hypothesis $C \cup C_{p'}; S \cdot S_{p'} \vdash \mathcal{M}[\![p']\!]_{\Gamma'\, I\, \Gamma(self)}(m): \mathcal{T}[\![p']\!](m)$. Furthermore by hypothesis we have that

$$C \cup (A \leq A'); S[A \leftarrow R]; \Gamma[\texttt{self} \leftarrow A] \vdash exp: V$$

By induction hypothesis on the part 2 of the theorem we have

$$C \cup (A \leq A'); S[A \leftarrow R] \vdash \Im[\![exp]\!]_{\Gamma[self\leftarrow A]\, I[A\leftarrow r]\, A}: V$$

(Note that $r: R$ and thus the hypothesis on I and S holds). By construction exp is not affected by the declarations in p' thus one also has

$$C \cup (A \leq A') \cup C_{p'}; S[A \leftarrow R] \cdot S_{p'} \vdash \Im[\![exp]\!]_{\Gamma[self\leftarrow A]\, I[A\leftarrow r]\, A}: V$$

which is equivalent to

$$C \cup C_p; S \cdot S_p \vdash \Im[\![exp]\!]_{\Gamma[self\leftarrow A]\, I[A\leftarrow r]\, A}: V$$

But then

$$C \cup C_p; S \cdot S_p \vdash \lambda self^A.\Im[\![exp]\!]_{\Gamma[self\leftarrow A]\, I[A\leftarrow r]\, A}: A \to V$$

The thesis follows by the rule [{}INTRO].
In the case of a multi-method then exp must be of the form & exp_1 ... & exp_n and $V \equiv \#\{D_1 \to T_1, \ldots, D_n \to T_n\}$. Again since p is well-typed it can be shown that $\mathcal{T}[\![p]\!]$ is a well-formed type. Then define exp^*_i as in the previous case. Thus we have to prove under the assumptions $C \cup C_p$ and $S \cdot S_p$ that

$$\begin{aligned}&(\cdots((\mathcal{M}[\![p']\!]_{\Gamma'\, I\, \Gamma(self)}(m)\\ &\quad \&^{\mathcal{T}[\![p']\!](m)\oplus\{A\times D_{\sigma(1)}\to T_{\sigma(1)}\}} \Im[\![exp^*_{\sigma(1)}]\!]_{\Gamma[self\leftarrow A]\, I[A\leftarrow r]\, A})\\ &\qquad\qquad \vdots\\ &\quad \&^{(\mathcal{T}[\![p']\!](m)\oplus\ldots\oplus\{A\times D_{\sigma(n-1)}\to T_{\sigma(n-1)}\})\oplus\{A\times D_{\sigma(n)}\to T_{\sigma(n)}\}} \Im[\![exp^*_{\sigma(h)}]\!]_{\Gamma[self\leftarrow A]\, I[A\leftarrow r]\, A})\end{aligned}$$

has type $\mathcal{T}[\![p']\!](m) \oplus \{A \rightsquigarrow V\}$ This can be shown by induction on n. For $n = 1$ use the induction hypothesis on p'. For $n > 1$ the proof is exactly the same as the corresponding one of the previous case.

2. We know that $C; S; \Gamma \vdash p: T$ and we have to prove that under the hypothesis C and S the following expression

   let A **hide** R **in**
   let $A \leq A'$ **in**
   $\Im[\![p']\!]_{\Gamma\, I\, A}[m^{(\mathcal{T}[\![p]\!](m))} := \mathcal{M}[\![p]\!]_{\Gamma\, I\, A}(m)]$

   has type T. Thus we prove that

   i. $C \cup (A \leq A'); S[A \leftarrow R] \vdash \Im[\![p']\!]_{\Gamma\, I[A\leftarrow r]\, \Gamma(self)}: T$
   ii. $R \leq S(A')$
   iii. $\mathcal{M}[\![p]\!]_{\Gamma\, I\, \Gamma(self)}(m): \mathcal{T}[\![p]\!](m)$ so that we substitute the variable $m^{\mathcal{T}[\![p]\!](m)}$ by a term of the same type.

   The first two conditions follow from the fact that $C; S; \Gamma \vdash p : T$ and by induction hypothesis on p'.
   Clearly $m^{\mathcal{T}[\![p]\!](m)}$ appears after the declarations given in p' since in λ_object no expressions can precede a **let** ... **in** declaration. Thus the thesis follows if we prove the point (iii) in an environment where also the constraints of p' are considered. Thus what we prove is that:

$$C \cup (A \leq A') \cup C_{p'}; S[A \leftarrow R] \cdot S_{p'} \vdash \mathcal{M}[\![p]\!]_{\Gamma\, I\, \Gamma(self)}(m) \colon \mathcal{T}[\![p]\!](m)$$

But since $(A \leq A') \cup C_{p'} = C_p$ and $[A \leftarrow R] \cdot S_{p'} = S_p$ then it is exactly what we have proved in the proposition 1 of the theorem

Appendix E

Original $\mathbf{F}_{\leq}$ rules

E.1 Subtyping

(refl) $$C \vdash T \leq T$$

(trans) $$\frac{C \vdash T_1 \leq T_2 \qquad C \vdash T_2 \leq T_3}{C \vdash T_1 \leq T_3}$$

(taut) $$C \vdash X \leq C(X)$$

(Top) $$C \vdash T \leq \mathsf{Top}$$

($\rightarrow$) $$\frac{C \vdash T_1 \leq S_1 \qquad C \vdash S_2 \leq T_2}{C \vdash S_1 \rightarrow S_2 \leq T_1 \rightarrow T_2}$$

($\forall$) $$\frac{C \vdash T_1 \leq S_1 \qquad C \cup \{X \leq T_1\} \vdash S_2 \leq T_2}{C \vdash \forall(X \leq S_1)S_2 \leq \forall(X \leq T_1)T_2}$$

E.2 Typing

[Vars] $$C \vdash x^T : T$$

[$\rightarrow$Intro] $$\frac{C \vdash a : T'}{C \vdash (\lambda x^T.a) : T \rightarrow T'}$$

[$\rightarrow$Elim] $$\frac{C \vdash a : S \rightarrow T \qquad C \vdash b : S}{C \vdash a(b) : T}$$

[TOP] $$C \vdash \mathsf{top}\colon \mathsf{Top}$$

[∀INTRO] $$\frac{C \cup \{X{\leq}T\} \vdash a\colon T'}{C \vdash \Lambda X{\leq}T.a\colon \forall(X{\leq}T)T'}$$

[∀ELIM] $$\frac{C \vdash a\colon \forall(X{\leq}S)T \qquad C \vdash S'{\leq}S}{C \vdash a(S')\colon T[X := S']}$$

[SUBSUMPTION] $$\frac{C \vdash a\colon T' \qquad C \vdash T' \leq T}{C \vdash a\colon T}$$

E.3 Typing algorithm

[Vars-Alg] $$C \vdash x^T\colon T$$

[→INTRO-Alg] $$\frac{C \vdash a\colon T'}{C \vdash (\lambda x^T.a)\colon T \to T'}$$

[→ELIM-Alg] $$\frac{C \vdash a\colon T' \qquad C \vdash b\colon S'{\leq}S}{C \vdash a(b)\colon T} \qquad \mathcal{B}(T')_C{=}\ S \to T$$

[TOP-Alg] $$C \vdash \mathsf{top}\colon \mathsf{Top}$$

[∀INTRO-Alg] $$\frac{C \cup \{X{\leq}T\} \vdash a\colon T'}{C \vdash \Lambda X{\leq}T.a\colon \forall(X{\leq}T)T'}$$

[∀ELIM-Alg] $$\frac{C \vdash a\colon T' \qquad C \vdash S'{\leq}S}{C \vdash a(S')\colon T[X := S']} \qquad \mathcal{B}(T')_C{=}\ \forall(X{\leq}S)T$$

Where $\mathcal{B}(T)_C{=}\ T$ if T is not a type variable $\mathcal{B}(T)_C = \mathcal{B}(C(T))_C$ otherwise

Appendix F

Translation of $\mathbf{F}_{\leq}^{\top}$ into explicit coercions

$$
\begin{array}{lcl}
X^* & = & X \\
\mathsf{Top}^* & = & 1 \\
(S \to T)^* & = & S^* \to T^* \\
(\forall (X{\leq}S)T)^* & = & \forall X.((X \multimap S^*) \to T^*) \\
\langle\!\langle \ell_1{:}T_1, \ldots, \ell_n{:}T_n \rangle\!\rangle^* & = & \langle\!\langle \ell_1{:}T_1^*, \ldots, \ell_n{:}T_n^* \rangle\!\rangle \\
([\ell_1{:}T_1, \ldots, \ell_n{:}T_n])^* & = & [\ell_1{:}T_1^*, \ldots, \ell_n{:}T_n^*] \\
(\mu X.T)^* & = & \mu X.T^* \\
 & & \\
\emptyset^* & = & \emptyset \\
(C \cup \{X{\leq}T\})^* & = & C^* \cup X \cup \{x{:}\, X \multimap T^*\}
\end{array}
$$

(refl)* $$C^* \vdash \mathsf{refl}{:}\, T^* \multimap T^*$$

(trans)* $$\frac{C^* \vdash a : T_1^* \multimap T_2^* \qquad C^* \vdash b : T_2^* \multimap T_3^*}{C^* \vdash \mathsf{trans}(a)(b) : T_1^* \multimap T_3^*}$$

(taut)* $$C^* \cup X \cup \{x{:}\, X \multimap C(X)^*\} \vdash x{:}\, X \multimap C(X)^*$$

(Top) $$C \vdash \mathsf{top}[T^*]{:}\, T^* \multimap 1$$

($\to$)* $$\frac{C^* \vdash a{:}\, S_1^* \multimap T_1^* \qquad C^* \vdash b{:}\, T_2^* \multimap S_2^*}{C^* \vdash \mathsf{arrow}(a)(b){:}\, (T_1^* \to T_2^*) \multimap (S_1^* \to S_2^*)}$$

($\forall$)* $$\frac{C^* \vdash a{:}\, S_1^* \multimap T_1^* \qquad C^* \cup X \cup \{x{:}\, X \multimap 1\} \vdash b{:}\, T_2^* \multimap S_2^*}{C^* \vdash \mathsf{forall}(a)(\Lambda X.\lambda x^{X \multimap 1}.b){:}\, \forall X((X \multimap T_1^*) \to T_2^*) \multimap \forall X((X \multimap S_1^*) \to S_2^*)}$$

(recd)* $$\frac{C^* \vdash a_1{:}\, S_1^* \multimap T_1^* \ldots C^* \vdash a_p{:}\, S_p^* \multimap T_p^*}{C^* \vdash \mathsf{recd}(a_1) \cdots (a_p){:}\, \langle\!\langle \ell_1{:}\, S_1^*, \ldots, \ell_p{:}\, S_p^*, \ldots, \ell_q{:}\, S_q^* \rangle\!\rangle \multimap \langle\!\langle \ell_1{:}\, T_1^*, \ldots, \ell_p{:}\, T_p^* \rangle\!\rangle}$$

(vart)* $$\frac{C^* \vdash a_1\colon S_1^* \hookrightarrow T_1^* \ldots C^* \vdash p\colon S_p^* \hookrightarrow T_p^*}{C^* \vdash \mathsf{vart}(a_1)\cdots(a_p)\colon [\ell_1\colon S_1^*, \ldots, \ell_p\colon S_p^*] \hookrightarrow [\ell_1\colon T_1^*, \ldots, \ell_p\colon T_p^*, \ldots, \ell_q\colon T_q^*]}$$

[Vars]* $$C^* \cup (x\colon T^*) \vdash x\colon T^*$$ translation of x^T

[→Intro]* $$\frac{C^* \cup \{x\colon T^*\} \vdash a\colon T'^*}{C^* \vdash (\lambda x^{T^*}.a)\colon T^* \to T'^*}$$

[→Elim]* $$\frac{C^* \vdash a\colon S^* \to T^* \qquad C^* \vdash b\colon S^*}{C^* \vdash a(b)\colon T^*}$$

[Top]* $$C^* \vdash \langle\rangle\colon 1$$

[∀Intro]* $$\frac{C^* \cup X \cup \{x\colon X \hookrightarrow T^*\} \vdash a\colon T'^*}{C^* \vdash \Lambda X \lambda x^{X \hookrightarrow T^*}.a\colon \forall X(X \hookrightarrow T^*) \to T'^*}$$

[∀Elim]* $$\frac{C^* \vdash a\colon \forall X(X \hookrightarrow S^*) \to T^* C^* \vdash b\colon S'^* \hookrightarrow S^*}{C^* \vdash a(S'^*)(b)\colon T^*[X := S'^*]}$$

[⟪⟫Intro]* $$\frac{C^* \vdash a_1\colon T_1^* \ldots C^* \vdash a_p\colon T_p^*}{C^* \vdash \langle \ell_1 = a_1, \ldots, \ell_p = a_p\rangle\colon \langle\!\langle \ell_1\colon T_1^*, \ldots, \ell_p\colon T_p^*\rangle\!\rangle}$$

[⟪⟫Elim]* $$\frac{C^* \vdash a\colon \langle\!\langle \ell_1\colon T_1^*, \ldots, \ell_p\colon T_p^*\rangle\!\rangle}{C^* \vdash a.\ell_i\colon T_i^*}$$

[[]Intro]* $$\frac{C^* \vdash a\colon T_i^*}{C^* \vdash [\ell_1\colon T_1^*, \ldots, \ell_i = a_i; \ldots, \ell_p\colon T_p^*]\colon [\ell_1\colon T_1^*, \ldots, \ell_i\colon T_i^*, \ldots, \ell_p\colon T_p^*]}$$

[[]Elim]* $$\frac{C^* \vdash b\colon [\ell_1\colon T_1^*, \ldots, \ldots, \ell_p\colon T_p^*] \qquad C^* \vdash a_1\colon T_1^* \to T^* \ldots C^* \vdash a_p\colon T_p^* \to T^*}{C^* \vdash \textbf{case } b \textbf{ of } \ell_1 \Rightarrow a_1, \ldots, \ell_p \Rightarrow a_p\colon T^*}$$

[μIntro]* $$\frac{C^* \vdash a\colon T^*[X := \mu X.T^*]}{C^* \vdash \textbf{intro}_{\mu X.T^*}(a)\colon \mu X.T^*}$$

[μElim]* $$\frac{C^* \vdash a\colon \mu X.T^*}{C^* \vdash \textbf{elim}(a)\colon [X := \mu X.T^*]}$$

[Subsumption]* $$\frac{C^* \vdash a\colon T'^* \qquad C^* \vdash b\colon T'^* \hookrightarrow T^*}{C^* \vdash \iota(b)(a)\colon T^*}$$

Bibliography

[ABGO93] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In *Proceedings of the 19th VLDB Conference*, Dublin, 1993.

[ABW+92] M. Atkinson, F. Bancilhlon, D. De Witt, K. Dittrich, D. Maier, and S. Zdonik. *The object-oriented database system manifesto*, chapter 1, pages 3–20. Morgan Kaufmann, 1992. In [BDe92].

[AC90] R. Amadio and L. Cardelli. Subtyping recursive types. Technical report, Digital System Research Center, August 1990.

[ACC93] M. Abadi, L. Cardelli, and P.-L. Curien. Formal parametric polymorphism. In Dezani, Ronchi, and Venturini, editors, *Böhm Festschrift.* 1993.

[AKP91] H. Aït-Kaci and A. Podelski. Towards a meaning of LIFE. Technical Report 11, Digital, Paris Research Laboratory, June 1991.

[AL91] A. Asperti and G. Longo. *Categories, Types and Structures: An Introduction to Category Theory for the Working Computer Scientist.* MIT-Press, 1991.

[Ama91] R. Amadio. *Recursion and Subtyping in Lambda Calculi.* PhD thesis, Università degli Studi di Pisa, 1991.

[App92] Apple Computer Inc., Eastern Research and Technology. *Dylan: an object-oriented dynamic language*, April 1992.

[Bar84] H.P. Barendregt. *The Lambda Calculus Its Syntax and Semantics.* North-Holland, 1984. Revised edition.

[BDe92] F. Bancilhon, C. Delobel, and P. Kanellakis (eds.). *Implementing an Object-Oriented database system: The story of O_2.* Morgan Kaufmann, 1992.

[BL90] K.B. Bruce and G. Longo. A modest model of records, inheritance and bounded quantification. *Information and Computation*, 87(1/2):196–240, 1990. A preliminary version can be found in *3rd Ann. Symp. on Logic in Computer Science*, 1988.

[BM92] Kim Bruce and John Mitchell. PER models of subtyping, recursive types and higher-order polymorphism. In *Proceedings of the Nineteenth ACM Symposium on Principles of Programming Languages*, Albequerque, NM, January 1992.

[Bru91] K.B. Bruce. The equivalence of two semantic definitions of inheritance in object-oriented languages. In *Proceedings of the 6th International Conference on Mathematical Foundation of Programming Semantics*, 1991. To appear.

[Bru92] K.B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. Technical Report CS-92-01, Williams College, Williamstown, MA 01267, January 1992.

[Bru93] K. B. Bruce. Safe type checking in a statically typed object-oriented programming language. In *20th Ann. ACM Symp. on Principles of Programming Languages*. ACM Press, 1993.

[BTCGS91] V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172–221, July 1991.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. A previous version can be found in Semantics of Data Types, LNCS 173, 51-67, Springer-Verlag, 1984.

[Car92] Luca Cardelli. Extensible records in a pure calculus of subtyping. Research report 81, DEC Systems Research Center, January 1992. To appear in [GM93].

[Car93] L. Cardelli. An implementation of $F_{<:}$. Technical Report 97, Digital Equipment Corporation, February 1993.

[Cas90a] G. Castagna. Modélisation et typage de quelques propriétés des langages orientés objets. Dea d'informatique fondamentale, Université Paris 7, 1990. (in english).

[Cas90b] G. Castagna. Teoria dei tipi dei linguaggi orientati ad oggetti. Tesi di laurea in scienze dell'informazione, Università di Pisa, April 1990.

[Cas92] G. Castagna. Strong typing in object-oriented paradigms. Technical Report 92-11, Laboratoire d'Informatique, Ecole Normale Supérieure - Paris, June 1992.

[Cas93a] G. Castagna. $F_{\leq}^{\&}$: integrating parametric and "ad hoc" second order polymorphism. In C. Beeri, A. Ohori, and D. Shasha, editors, *Proc. of the 4th International Workshop on Database Programming Languages*, Workshops in Computing, pages 335–355, New York City, September 1993. Springer-Verlag. DBPL4.

[Cas93b] G. Castagna. A meta-language for typed object-oriented languages. In R.K. Shyamasundar, editor, *13th Conference on the Foundations of Software Technology and Theoretical Computer Science*, number 761 in LNCS, pages 52–71, Bombay, India, December 1993. Springer-Verlag. FST&TCS'93.

[CCH+89] P.S. Canning, W.R. Cook, W.L. Hill, J. Mitchell, and W.G. Olthoff. F-bounded quantification for object-oriented programming. In *ACM Conference on Functional Programming and Computer Architecture*, September 1989.

[CCHO89] P.S. Canning, W.R. Cook, W.L. Hill, and W.G. Orthoff. Interfaces for strongly-typed object-oriented programming. In *OOPSLA '89*, New Orleans, October 1989.

[CDG+89] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G.Nelson. *Modula-3 Report*. Digital SRC, 130 Lytton Avenue, Palo Alto CA, tr 52 edition, November 1989.

[CG92] P. L. Curien and G. Ghelli. Coherence of subsumption, minimum typing and the type checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2(1), 1992.

[CGL92a] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. In *ACM Conference on LISP and Functional Programming*, pages 182–192, San Francisco, July 1992. ACM Press. Extended abstract.

[CGL92b] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping, 1992. To appear in *Information and Computation*. An extended abstract has appeared in the proceedings of the *ACM Conference on LISP and Functional Programming*, pp.182-192; San Francisco, June 1992.

[CGL93] G. Castagna, G. Ghelli, and G. Longo. A semantics for λ&-*early*: a calculus with overloading and early binding. In M. Bezem and J.F. Groote, editors, *International Conference on Typed Lambda Calculi and Applications*, number 664 in LNCS, pages 107–123, Utrecht, The Netherlands, March 1993. Springer-Verlag. TLCA'93.

[CH88] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76, 1988.

[CHC90] W.R. Cook, W.L. Hill, and P.S. Canning. Inheritance is not subtyping. *17th Ann. ACM Symp. on Principles of Programming Languages*, January 1990.

[CL91a] L. Cardelli and G. Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, 1991.

[CL91b] G. Castagna and G. Longo. From inheritance to Quest's type theory. In *Ecole Jeunes Chercheurs du GRECO de Programmation*, Sophia-Antipolis (Nice), April 1991. Talk given at the 5th Jumelage on Typed Lambda Calculus - Paris - January 1990.

[CM91] L. Cardelli and J.C. Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1(1):3–48, 1991.

[CMMS91] L. Cardelli, S. Martini, J.C. Mitchell, and A. Scedrov. An extension of system F with subtyping. In T. Ito and A.R. Meyer, editors, *Theoretical Aspects of Computer Software*, pages 750–771. Springer-Verlag, September 1991. LNCS 526 (preliminary version). To appear in Information and Computation.

[CP89] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *OOPSLA '89, New Orleans*, October 1989.

[CP94] G. Castagna and B.C. Pierce. Decidable bounded quantification. In *21st Annual Symposium on Principles Of Programming Languages*, pages 151–162, Portland, Oregon, January 1994. ACM Press. POPL'94.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[dB72] Nicolas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[DG87] L.G. DeMichiel and R.P. Gabriel. Common lisp object system overview. In Bézivin, Hullot, Cointe, and Lieberman, editors, *Proc. of ECOOP '87 European Conference on Object-Oriented Programming*, number 276 in LNCS, pages 151–170, Paris, France, June 1987. Springer-Verlag.

[ES90] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[Ghe90] G. Ghelli. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism*. PhD thesis, Dipartimento di Informatica, Università di Pisa, March 1990. Tech. Rep. TD-6/90.

[Ghe91] G. Ghelli. A static type system for message passing. In *Proc. of OOPSLA '91*, 1991.

[Ghe93a] G. Ghelli. Divergence of $F_{\leq}$ type-checking. Technical Report 5/93, Dipartimento d'Informatica, Università degli Studi di Pisa, 1993.

[Ghe93b] G. Ghelli. Recursive types are not conservative over $F_{\leq}$. In M. Bezem and J.F. Groote, editors, *International Conference on Typed Lambda calculi and Applications*, number 664 in LNCS, pages 146–162, Utrecht, The Netherlands, March 1993. Springer-Verlag. TLCA'93.

[Ghe93c] G. Ghelli. S-All-Loc is not transitive. Mail to the TYPES mailing list, February 1993.

[Gir72] J-Y. Girard. Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur. Thèse de doctorat d'état, 1972. Université Paris VII.

[GM85] Joseph Goguen and José Meseguer. EQLOG: equality, types and generic modules for logic programming. In deGroot and Lindstrom, editors, *Functional and Logic Programming*. Prentice-Hall, 1985.

[GM89] J.A. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Technical Report SRI-CSL-89-10, Computer Science Laboratory, SRI International, July 1989.

[GM93] Carl A. Gunter and John C. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. The MIT Press, 1993. To appear.

[GP93] G. Ghelli and B. Pierce. Bounded existentials and minimal typing. Draft report, Dipartimento d'Informatica Università di Pisa, 1993. Unpublished.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its implementation*. Addison-Wesley, 1983.

[Hin64] R. Hindley. The Church-Rosser property and a result of combinatory logic. Dissertation, 1964. University of Newcastle-upon-Tyne.

[Hin69] R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. A.M.S.*, 149:22–60, January 1969.

[How80] W.A. Howard. The formulae-as-types notion of construction. In J.R. Hindley and J.P. Seldin, editors, *To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus and formalism*. Academic Press, 1980.

[KB70] Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[Kee89] S.K. Keene. *Object-Oriented Programming in* COMMON LISP*: A Programming Guide to CLOS*. Addison-Wesley, 1989.

[KLR92] P. Kanellakis, C. Lécluse, and P. Richard. *Introduction to the* O_2 *data model*, chapter 3, pages 61–76. Morgan Kaufmann, 1992. In [BDe92].

[KS92] D. Katiyar and S. Sankar. Completely bounded quantification is decidable. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, pages 68–77, San Francisco, June 1992.

[LM91] G. Longo and E. Moggi. Constructive natural deduction and its ω-set interpretation. *Mathematical Structures in Computer Science*, 1(2):215–253, 1991.

[LMS93] G Longo, K. Milsted, and S. Soloviev. The genericity theorem and parametricity in functional languages. *Theoretical Computer Science*, 1993. Special issue in honour of Corrado Böhm, to appear. An extended abstract has been presented at the 8th Annual IEEE Symposium on Logic in Computer Science, Montreal, June 1993.

[Lon83] G. Longo. Set-theoretical models of lambda-calculus: Theories, expansions, isomorphisms. *Annals of Pure and Applied Logic*, 24:153–188, 1983.

[Lon93] Giuseppe Longo. Types as parameters. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT'93: Theory and Practice of Software Development*, number 668 in LNCS, pages 658–670, Orsay, France, April 1993. Springer-Verlag.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall International Series, 1988.

[MH88] J.C. Mitchell and R. Harper. The essence of ML. *15th Ann. ACM Symp. on Principles of Programming Languages*, January 1988.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mit86] J. C. Mitchell. A type inference approach to reduction properties and semantics of polymorphic expressions. In *ACM Conference on LISP and Functional Programming*, pages 308–319, 1986.

[Mit90a] J.C. Mitchell. Toward a typed foundation for method specialization and inheritance. *17th Ann. ACM Symp. on Principles of Programming Languages*, January 1990.

[Mit90b] John C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 109–124, January 1990. To appear in [GM93].

[MOM90] N. Martí-Oliet and J. Meseguer. Inclusions and subtypes. Technical report, SRI International, Computer Science Laboratory, December 1990.

[MP85] J.C. Mitchell and G. Plotkin. Abstract types have existential type. *12th Ann. ACM Symp. on Principles of Programming Languages*, pages 37–51, January 1985.

[MR91] Q.Y. Ma and J. Reynolds. Types, abstractions and parametric polymorphism, part II. In *MFCS*. LNCS, Springer-Verlag, 1991.

[New42] M.H.A. Newman. On theories with a combinatorial definition of “equivalence”. *Annals of Math.*, 43(2):223–243, 1942.

[NeX91] NeXT Computer Inc. *NeXTstep-concepts. Chapter 3: Object-Oriented Programming and Objective-C*, 2.0 edition, 1991.

[Pie93] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 1993. To appear; also to appear in [GM93]. Preliminary version in proceedings of POPL '92.

[Plo75] G.D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1, 1975.

[PT93] B.C. Pierce and D.N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 1993. To appear; a preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title “Object-Oriented Programming Without Recursive Types”.

[PW92] L.J. Pinson and R.S. Wiener. *Objective-C: Object-Oriented Programming Techniques*. Addison-Wesley, 1992.

[“Q65] Joaquin Salvador Lavado “Quino”. *Mafalda*, volume 1. Ediciones de la Flor, Buenos Aires, Argentina, December 1965.

[Rém89] D. Rémy. Typechecking records and variants in a natural extension of ML. In *16th Ann. ACM Symp. on Principles of Programming Languages*, 1989.

[Rém90] Didier Rémy. *Algèbres Touffues. Application au Typage Polymorphe des Objects Enregistrements dans les Langages Fonctionnels*. Thèse de doctorat, Université de Paris 7, 1990.

[Rey74] J.C. Reynolds. Towards a theory of type structures. *LNCS*, 19:408–425, 1974.

[Rey83] J.C. Reynolds. Types, abstractions and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing ’83*, pages 513–523. North-Holland, 1983.

[Rey84] J.C. Reynolds. Polymorphism is not set-theoretic. *LNCS*, 173, 1984.

[Ros73] B. K. Rosen. Tree manipulation systems and Church-Rosser theorems. *Journal of ACM*, 20:160–187, 1973.

[Rou90] F. Rouaix. *ALCOOL-90, Typage de la surcharge dans un langage fonctionnel*. PhD thesis, Université PARIS VII, December 1990.

[Sco76] D. Scott. Data-types as lattices. *S. I. A. M. J. Comp.*, 5:522–587, 1976.

[Str67] C. Strachey. Fundamental concepts in programming languages. Lecture notes for International Summer School in Computer Programming, Copenhagen, August 1967.

[Str84] Bjarne Stroustrup. Data abstraction in C. *AT&T Bell Laboratories Technical Journal*, 63(8):1701–1732, October 1984.

[Tsu92] Hideki Tsuiki. *A record calculus with a merge operator*. PhD thesis, Faculty of Environmental Information, Keio University, November 1992.

[Tsu94] Hideki Tsuiki. A normalizing calculus with overloading and subtyping. In *TACS*, LNCS. Springer-Verlag, 1994.

[Tur37] A.M. Turing. The $\mathfrak{p}$-functions in λ-K-conversion. *Journal of Symbolic Logic*, 2, 1937.

[Wan87] Mitchell Wand. Complete type inference for simple objects. In *2nd Ann. Symp. on Logic in Computer Science*, 1987.

[Wan88] Mitchell Wand. Corrigendum: Complete type inference for simple objects. In *3rd Ann. Symp. on Logic in Computer Science*, 1988.

[Wan91] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93(1):1–15, 1991.

[WB89] Philip Wadler and Stephen Blott. How to make "ad-hoc" polymorphism less "ad-hoc". In *16th Ann. ACM Symp. on Principles of Programming Languages*, pages 60–76, 1989.

Index

$\rhd_R$, 59
$\rhd_R^=$, 59
$\rhd_R^+$, 59
$\rhd_R^*$, 59
&, 81
•, 81
$\leftarrow$, 99
$\cap$-closure, *see* meet closure
Ω_T, 112
Θ_T, 113
$\mathbf{Y}_T$, 112
$\Downarrow$, 83
$\leadsto$, 148, 232, 260
$\oplus$, 124
$\sim$, 85, 172
$\setminus$, 99, 241
ζ, 133

β, 90
$\beta_{\&}$
    for second order, 240
    for simple typing, 90
    unconditional, 118
$\beta_{\forall}$, 240
$\beta_{\&}^+$, 125

Church-Rosser, *see* theorem
    property, 60
class, 66
    abstract, 107
    extension, 72, 267
    generic, 271
    partially abstract, 109
closure
    compatible, 59
    reflexive, 59
    transitive, 59
code reuse, 52
coerce, 73
coercion combinators, 215
coercion expressions, 231
coercions, 128–131
    semantic, 179
completion, 173
confluence, 60
contractum, 60
covariance, 78, 265
    rule in $\lambda\&$, 83
    vs. contravariance, 107, 276
Curry-Horward isomorphism, 231

early binding, 52, 65, 175–176
encoding
    of simple records in $\lambda\&$, 98
    of surjective pairs in $\lambda\&$, 97
    of updatable records in $\lambda\&$, 99

fixed point combinators, 112
$F_\leq$, 198
$F_\leq^{\&}$, 223
$F_\leq^{\&\top}$, 259
$F_\leq^{\top}$, 201
$F_\leq^{\top\mu}$, 218

generator, 103

heterarchy, 71

inheritance
    by generator extension, 103
    in $\lambda\&$, 105
instance variables, 63
interface, 67

$\lambda\&$, 81–120

$\lambda\&^{-}_{\leq}$, 114
$\lambda\&^{+}$, 121
$\lambda\&^{-}_{\mathrm{T}}$, 114
λ_object , 150
$\lambda^{\{\}}$, 132
$\lambda\&$+coerce, 128
language
  target
    for $F^{\top}_{\leq}$, 214
    for λ_object , 167
late binding, 52, 65, 82, 240
lemma
  Hindley-Rosen, 95, 255
  substitution
    for $\lambda\&$+coerce, 129
    for $\lambda\&^{+}$, 126
    in $\lambda\&$, 92
loss of information, 196, 224

meet closure, 226
message, 64
messages, 75
method, 63
  binary, 104, 105
  deferred, 107
  multi-method, 74
  virtual, 107
multiple dispatch, 74, 269
multiple inheritance, 71, 79, 122
`Mytype`, 104, 264

normal form, 60
notion of reduction, 59

objects, 63, 151
  as records analogy, 54, 102, 196
overloaded functions
  semantic of, 184
overloaded types
  completion of, 172–175
  semantics of, 180–184
overloading, 64, 75
  coherent, 176fn, 191
overriding, 70

partial equivalence relations, *see* PER
PER, 177–179
polymorphism
  ad hoc, 51
  explicit, 197
  F-bounded, 104
  implicit, 196
  parametric, 51
$\mathcal{P}\omega$, 177
pretypes, 84
product
  indexed, 181

receiver, 64
records
  generator, 103
  simple, 98
  updatable, 99, 131, 241
redex, 60
reduction
  in $F^{\&}_{\leq}$, 240
  in $\lambda\&$, 90
relation, 59
  compatible, 59
  equivalence, 59
  reducttion, 59
  reflexive, 59
  transitive, 59
residual, 82

`self`, 73
semantics
  of $\rightarrow$, 178
  of $F^{\top}_{\leq}$, 212
  of $\lambda\&$, 171
  operational, 152
state coherence, 78
stratified system, 113
subclass, 68
subject reduction, *see* theorem
substitution
  in $\lambda\&$, 90
  lemma, *see* lemma
subsumption
  elimination, 88
  rule, 88

subtyping, 69, 78
  algorithm for $F_{\leq}^{\&\top}$, 259
  algorithm for $F_{\leq}^{\top}$, 207
  algorithm of $F_{\leq}^{\&}$, 236
  in $F_{\leq}^{\&}$, 227
  in $\lambda\&$, 84
  in the toy language, 143
  semantics of, 179
`super`, 73
superclass, 68
surjective pairings, 98

tag, 150
target language, *see* language
termination, 260
terms
  in $F_{\leq}^{\&}$, 238
  in $\lambda\&$, 86
  in λ_object , 152
theorem
  Church-Rosser
    for $\lambda\&$+coerce, 129
    for $\lambda^{\{\}}$, 134
    for $F_{\leq}^{\&}$, 255
    for $\lambda\&$, 95
    for $\lambda\&^{+}$, 127
  coherence for $F_{\leq}^{\top}$, 217
  coherence of $F_{\leq}^{\&}$, 236
  completeness
    of the typing algorithm for $F_{\leq}^{\top}$, 207
  conservativity of $\lambda\&^{+}$, 126
  conservativity of recursive types, 218
  soundness
    of semantics w.r.t. reductions, 188
    of semantics w.r.t. types, 185
    of the typing algorithm for $F_{\leq}^{\top}$, 207
  strong normalization, 114
  subject reduction
    for $\lambda\&$+coerce, 129
    for $\lambda^{\{\}}$, 133
    for $F_{\leq}^{\&}$, 243
    for $\lambda\&$, 92
    for $\lambda\&^{+}$, 126
  subsumption elimination, 88
  termination
    for $F_{\leq}^{\&\top}$, 260
    for $F_{\leq}^{\top}$, 209
  transitivity elimination, *see* transitivity elimination
theory, 60
  conservative extension, 60
  extension, 60
transitivity elimination
  in $F_{\leq}^{\top}$, 208, 230
  in $\lambda\&$, 85
type
  compile-time, 82
  errors, 153
  representation, 100
  run-time, 82
type constraint system, 204
typed induction, 115
types
  in the toy language, 144
  dynamic, 160
  general overloaded, 172
  in $F_{\leq}^{\&}$, 227
  in $\lambda\&$, 85
  in λ_object , 154
  isolated, 97
  recursive, 213, 218
  variant, 213
typing rules
  in the toy language, 145
  in $F_{\leq}^{\&}$, 238
  in $\lambda\&$, 87

value, 83, 152
  in $\lambda\&$, 89

wrapping, 103

Look, that's the world, you see?
Do you know why this world is nice? Ehee?
'cause it's a model ... The original is a disaster!