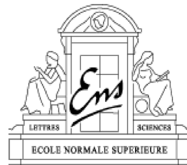
Université **PARIS 7**



TYPES ET MODÈLES :
DES LANGAGES OBJETS SÉQUENTIELS À LA MOBILITÉ ET À LA SÉCURITÉ

Habilitation à diriger des recherches *janvier 2002*

Directeur :
Giuseppe Longo

Rapporteurs :
Martín Abadi
Gérard Boudol
Luca Cardelli
Pierre-Louis Curien

Autres Examinateurs :
Mariangiola Dezani

Giuseppe Castagna

Département d'Informatique
de l'École Normale Supérieure

À Ilaria,
notre chatoune aimée.

Table des matières

1 Préface **5**
1.1 Ce mémoire en 100 lignes (... voire un peu plus) . . . 5
1.2 Une remarque importante . . . 7
1.3 Publications couvertes par ce mémoire . . . 8
1.4 Autres publications non couvertes par ce mémoire . . . 9
1.5 Remerciements . . . 10

2 Notions préliminaires **11**
2.1 Surcharge et liaison tardive . . . 11
2.2 Paradigme à objets . . . 13
2.2.1 Le λ&-calcul . . . 13
2.2.2 λ& et programmation orientée objet . . . 18

I Surcharge et langages à objets séquentiels **23**

1 Typage des langages objets avec classes: l'utilisation de la covariance **25**

2 Applications aux langages Java et O_2 **29**
2.1 O_2 . . . 29
2.2 Java . . . 32
2.3 Conclusion . . . 35

II Généralisation de la surcharge: modules et démonstration automatique **39**

1 Systèmes de modules **41**
1.1 Introduction . . . 41
1.2 Le langage de modules . . . 42
1.2.1 Sous-typage . . . 43
1.2.2 Modules manifestes . . . 44
1.3 Modules avec surcharge et liaison tardive . . . 45
1.3.1 Surcharge et programmation modulaire . . . 45
1.3.2 Surcharge et partage de types . . . 46
1.4 Un calcul de modules avec surcharge . . . 46
1.4.1 Syntaxe . . . 46
1.4.2 Sémantique statique . . . 48
1.4.3 Sémantique dynamique . . . 50

1.5 Questions pendantes . . . 52
1.6 Conclusion . . . 52

2 Démonstration automatique **55**
2.1 Motivations . . . 56
2.2 Aperçu . . . 57
2.2.1 Une brève introduction à la théorie des types dépendants . . . 57
2.2.2 Le système $\lambda\Pi$ . . . 58
2.2.3 Sous-typage pour $\lambda\Pi$: le système $\lambda\Pi_{\leq}$ . . . 58
2.2.4 Surcharge pour $\lambda\Pi_{\leq}$: le système $\lambda\Pi^{\&}$ . . . 60
2.3 Conclusion . . . 66

III Distribution et mobilité **67**

1 Modélisation d'objets mobiles **69**
1.1 Description du modèle générique . . . 70
1.1.1 Syntaxe . . . 70
1.1.2 Congruence structurale . . . 71
1.1.3 Envoi de messages . . . 72
1.1.4 Réplication . . . 74
1.2 MA^{++} . . . 74
1.2.1 Ambients Mobiles . . . 74
1.2.2 Syntaxe de MA^{++} . . . 75
1.2.3 Réduction . . . 75
1.2.4 Exemples . . . 76
1.2.5 Système de types pour MA^{++} . . . 77

2 Analyses de Sécurité **83**
2.1 Aperçu . . . 85
2.2 Syntaxe . . . 85
2.2.1 Types . . . 86
2.2.2 Environnements et typage . . . 86
2.2.3 Exemples . . . 87
2.2.4 Sûreté . . . 88
2.3 Politiques de sécurité . . . 89
2.4 Version distribuée . . . 90
2.5 Conclusion . . . 92

3 Un modèle orienté sécurité **93**
3.1 Abstractions et fonctionnalités . . . 94
3.2 Syntaxe et Sémantique . . . 98
3.2.1 Synchronisation . . . 98
3.2.2 Sémantique de réduction . . . 99
3.2.3 Discussion . . . 101
3.3 Équivalences . . . 102
3.4 Système de types . . . 103
3.4.1 Interfaces . . . 103
3.4.2 Syntaxe des types . . . 104
3.4.3 Règles de typage . . . 104

3.4.4 Propriétés 108
3.5 Conclusion 108

4 Boxed Ambients 109
4.1 Ambients mobiles et sécurité à plusieurs niveaux 109
4.1.1 Ambients mobiles 110
4.1.2 Un problème simple d'accès aux ressources 110
4.1.3 Solutions possibles 110
4.1.4 Enseignements 112
4.2 Boxed Ambients 113
4.3 Canaux 115
4.4 Typage 115
4.5 Comparaison entre les boxed ambients et les ambients mobiles 118
4.6 Moded Types 119
4.7 Communications Asynchrones 120
4.8 Sécurité 121
4.9 Exemples 122
4.9.1 Wrappers 122
4.9.2 Firewalls 122
4.9.3 Chevaux de Troie 123
4.9.4 Sécurité du langage distribué 123
4.10 Conclusion et travaux connexes 126

IV Conclusion 127

Conclusion 129

Chapitre 1

Préface

1.1 Ce mémoire en 100 lignes (. . . voire un peu plus)

Ce mémoire commence là où ma thèse de doctorat s'achève. Dans ma thèse [Cas94] je décris un modèle de la programmation orientée à objets basé sur la surcharge à liaison tardive. L'intérêt principal de la surcharge à liaison tardive est qu'elle permet une programmation de type incrémentale et la réutilisation du code, d'où sa liaison avec les paradigmes à objets. Les deux premières parties de ce mémoire relatent des travaux qui sont la continuation naturelle de ma thèse. C'est pourquoi avant de commencer l'exposé propre à cette habilitation je rappelle dans le chapitre 2 qui suit les idées principales et quelques notions de ma thèse. Ce mémoire d'habilitation se compose de trois parties qui traitent respectivement de langages à objets séquentiels, de modules et déduction automatique, et de distribution et mobilité.

Première partie Dans son application aux objets l'un des avantages du modèle de ma thèse est qu'il clarifie l'utilisation de la covariance et de la contravariance dans la pratique de la programmation à objets. Dans la première partie de ce mémoire j'approfondis cet aspect du modèle.

En particulier dans le chapitre I.1 j'explore les différentes possibilités de typage des paramètres des méthodes et développe une technique générale pour permettre la spécialisation covariante de méthodes dans des classes qui sont en relation de sous-typage sans que cela affecte la correction par rapport aux types.

Dans le chapitre I.2 j'applique et adapte la technique du chapitre précédent à deux langages commerciaux: O_2 [BDK92] et Java [GJS96]. Je choisis ces langages parce qu'ils présentent chacun un obstacle majeur à l'utilisation de la technique du chapitre I.1: l'utilisation de l'héritage multiple dans le cas de O_2, et la présence de méthodes surchargées dans Java. Les solutions proposées pour les deux langages sont complètement opposées. Dans le cas de O_2 je propose une modification sémantique qui n'affecte pas la syntaxe du langage, ce qui permet son application à toutes les bases existantes d'objets O_2 persistants. Dans le cas de Java je propose une extension syntaxique conservative du langage dont la compilation produit du bytecode standard, ce qui permet son exécution sur tout le parc de Java Virtual Machines installées.

Cette partie de ma recherche inclut un troisième volet que je n'ai pas jugé opportun d'inclure dans ce mémoire. Je voudrais quand même en dire quelques mots ici. Il s'agit de l'intégration de la technique du chapitre I.1 aux langages objets à prototypes en général et au $\lambda_o^{\prec}$-calcul de [FHM94, BL95] en particulier. Le problème principal est que pour pouvoir utiliser en pratique la surcharge, il faut que le typage soit décidable. Donc le premier pas consiste à définir un calcul aussi expressif que $\lambda_o^{\prec}$ mais dont le typage soit décidable, ce que j'ai fait en collaboration avec L. Liquori dans [LC96].

Il n'est alors pas trop difficile d'intégrer la technique du premier chapitre à ce travail, ce que nous montrons dans un manuscrit que, pour des raisons de disponibilité, nous n'avons jamais achevé.

Deuxième partie Si l'avantage de la surcharge à liaison tardive est de permettre une programmation de type incrémentale et la réutilisation de code, ces avantages ne doivent pas être confinés aux paradigmes à objets. C'est pourquoi j'ai étudié son intégration dans d'autres paradigmes tels que les systèmes de modules et les langages logiques pour la démonstration automatique.

En particulier dans le chapitre II.1 je considère les systèmes de modules à la SML [MQ85, MTH90] qui, quoique permettant un découpage modulaire des programmes et la définition de transformations (appelées *foncteurs*) sur les modules mêmes, n'autorisent qu'une forme très limitée de réutilisation de code et de programmation incrémentale car ne possèdent pas les caractéristiques d'héritage et de spécialisation de code qui ont fait le succès des langages à objets. Pour pallier ce problème je montre dans ce chapitre comment définir des foncteurs surchargés à liaison tardive et les utiliser en pratique.

Dans le chapitre II.2 qui suit je montre comment fusionner dans un seul formalisme type dépendants, sous-typage et surcharge à liaison tardive. L'intérêt de combiner sous-typage et les types dépendants est amplement discuté dans de nombreux articles de la littérature. Ces mêmes articles montrent qu'une certaine quantité de surcharge est aussi souhaitable, mais pour combler cette carence ils utilisent les types intersection lesquels ne fournissent qu'une forme très limitée de surcharge. Dans le cadres des types dépendants ni l'ajout du sous-typage ni celui de la surcharge à liaison tardive sont aisés, et dans le chapitre j'essaie de donner l'intuition des difficultés techniques que cela comporte. La récompense d'un tel effort est une système très puissant qui, grâce justement à la liaison tardive, permet le même type de programmation modulaire et incrémentale qui a été rendu populaire par les langages orientés objets.

Troisième partie Le point de départ des parties précédentes ce sont les systèmes de types pour les langages orientés objets. Si la programmation orientée objets s'est établie comme un standard "de facto" pour le développement de systèmes logiciels complexes, les avancées dans les communications et dans les matériaux aujourd'hui poussent le développement de la programmation distribuée à grande échelle. Ceci explique pourquoi les calculs d'agents mobiles reçoivent une attention grandissante dans la communauté des langages de programmation.

C'est donc tout à fait naturellement que dès l'étude de langages à objets séquentiels j'ai réorienté mes recherches vers le domaine de la mobilité. Dans les chapitres de la troisième partie je vais donc exposer des études similaires, dans l'esprit et dans la méthodologie suivie, à ceux décrits dans les deux premières parties, mais dont l'objet ce sont les paradigmes avec mobilité forte pour des systèmes distribués. Il s'agit donc d'un changement de thème et, en même temps, d'une suite somme toute naturelle de mes travaux antérieurs, mais avec un gros distinguo: tandis que dans les cadres précédents nous avions des bases et des références bien établies —les paradigmes à objets et leurs problèmes d'expressivité et de typage dans la première partie, les systèmes de modules et de déduction automatique dans la deuxième—-, dans le contexte que nous allons étudier il n'y a ni de paradigmes consolidés ni de problèmes bien définis. C'est pourquoi en affrontant cette partie j'ai effectué une recherche à spectre plus large.

Un premier pas logique de cette démarche, surtout à la lumière de mes travaux antérieurs, est l'étude des objets en présence de mobilité, ce que je présente dans le chapitre III.1. L'approche suivie consiste à prendre un modèle de mobilité et lui greffer les méthodes et les envois de message. En fait, je pousse l'approche un peu plus loin, car au lieu de choisir parmi les modèles de mobilité définis dans la littérature un modèle particulier, je définis mon extension à objets de manière assez générique de façon que des choix différentes pour les actions de mobilité produisent des instances distinctes de ce modèle.

Un deuxième pas tout aussi naturel de cette démarche est de vérifier si les techniques de typage et, plus généralement, d'analyse statique peuvent être utilisées pour la détection et l'élimination précoces de nouveaux problèmes de sécurité que la récente explosion d'Internet et l'apparition conjointe de la mobilité à la fois software et hardware a rendu si pressants. J'étudie cela dans le chapitre III.2 en développant un système dont les types décrivent des approximations du comportement des agents qu'ils typent. Ce système, développé pour le modèle de mobilité des Ambients Mobiles [CG98], permet de définir de manière précise des politiques de sécurité et de détecter statiquement des attaques comme celles de type Cheval de Troie. J'étudie aussi le cas distribué où les agents peuvent évoluer dans des environnements non typés et, donc, potentiellement hostiles. Ce dernier aspect présente d'intéressantes analogies avec l'architecture de sécurité de la Java Virtual Machine.

L'étude du chapitre III.2 montre qu'il est possible d'effectuer des contrôles statiques de sécurité pour le calcul des Ambients. Toutefois ce calcul *ne supporte pas* des mécanismes et des politiques de sécurité car il est dépourvu de primitives permettant une modélisation complète et/ou naturelle des accès aux ressources. En d'autres termes, le modèle des Ambients n'est pas complètement adapté aux problématiques liées à la sécurité. Je me suis donc orienté vers la définition d'autres modèles de mobilité dont le *design* prenne en compte *dès le départ* certains aspects de sécurité. Pour cela j'ai procédé en deux directions parallèle qui sont l'objet des chapitres III.3 et III.4.

La première direction, qui est aussi la plus ancienne, a été de définir un tout nouveau modèle de mobilité en partant du π-calcul et en l'enrichissant par des primitives suggérées principalement par des considérations pratiques en vue d'une implantation. Ce travail, qui est décrit dans le chapitre III.3, a donné lieu à la définition du Seal Calcul, une extension du π-calcul où (i) toute interaction —soit-elle de la mobilité ou de la communication— a lieu via une synchronisation sur des canaux, où (ii) les ressources et leur accès sont aisément identifiables et où (iii) certaines propriétés de sécurité sont assurées par la définition même du calcul, telle par exemple l'absence de *covert channels*. Il est très intéressant de noter que les récents essais parus dans la littérature de modifier le calcul des Ambients afin de le doter d'une théorie équationnelle intéressante [LS00, MH02], le rapprochent de plus en plus de la définition du Seal Calcul, bien que ce dernier ait été défini surtout à partir de considérations pratiques.

La deuxième direction est celle de modifier le calcul des Ambients. En partant de l'analyse des difficultés que l'on rencontre à définir des politiques de contrôle d'accès aux ressources dans les Ambients et en mettant à profit les enseignements du Seal Calcul je définis dans le chapitre III.4 les *Boxed Ambients*. Les Boxed Ambients héritent des Ambients Mobiles l'anonymat des communications (on ne spécifie aucun canal) et une partie des primitives de mobilité (`in` et `out` mais pas `open`), tandis que de Seal ils héritent du modèle de synchronisation des communications à canaux localisés. Je montre que les nouvelles primitives fournissent au calcul des moyens de protection des ressources et de contrôle des accès plus efficaces, et qu'en même temps elles n'affectent pas l'expressivité et l'esprit computationnel des Ambients Mobiles, ainsi que leur élégance. En outre, le typage du nouveau calcul permet, par rapport aux Ambients Mobiles, une plus grande flexibilité des communications, une mobilité moins contraignante et il jette un éclairage nouveau sur la relation entre synchronie et asynchronie.

Ces deux dernières directions de recherche apparaissent au moment actuel très prometteuses. C'est surtout sur elles, ainsi que sur un troisième axe de recherche, les transformations de documents XML (dont l'état de mes études n'est pas encore assez avancé pour être inclus dans ce rapport), que j'ai l'intention de concentrer mes recherches futures.

1.2 Une remarque importante

Dans ce mémoire d'habilitation j'ai taché de donner l'intuition du travail en évitant autant que possible les détails techniques. Ces détails en fait peuvent être trouvés dans les publications citées

au début de chaque chapitre. Le lecteur intéressé ou insatisfait est donc invité à se rapporter à ces indications ou à la section qui suit pour approfondir l'argument de son choix.

1.3 Publications couvertes par ce mémoire

Tout le matériel couvert dans ce mémoire a été publié dans des conférences et des revues (à l'exception de quelques parties du chapitre III.3 fruits de travaux encore en cours). Le lecteur intéressé trouvera tous les détails dans les références suivantes.

Chapitre I.1

[Cas95a] G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.

[BCC+96] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

Chapitre I.2

[BC96] J. Boyland and G. Castagna. Type-safe compilation of covariant specialization: a practical case. In Pierre Cointe, editor, *ECOOP '96, 10th European Conference on Object-Oriented Programming*, number 1098 in Lecture Notes in Computer Science, pages 3–25. Springer, 1996.

[BC97] J. Boyland and G. Castagna. Parasitic methods: Implementation of multi-methods for Java. In *OOPSLA '97, 12th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications. SIGPLAN Notices*, 32(10):66–76, 1997.

Chapitre II.1

[AC96] M.-V. Aponte and G. Castagna. Programmation modulaire avec surcharge et liaison tardive. In *Journées Francophones des Langages Applicatifs*, Val Morin, Québec, Canada, 1996.

Chapitre II.2

[CC01] G. Castagna and G. Chen. Dependent types with subtyping and late-bound overloading. *Information and Computation*, 168(1):1–67, 2001.

Chapitre III.1

[BCC00] M. Bugliesi, G. Castagna, and S. Crafa. Typed mobile objects. In *Proceedings of CONCUR 2000 (11th. International Conference on Concurrency Theory)*, number 1877 in Lecture Notes in Computer Science, pages 504–520. Springer, 2000.

[BCC01d] M. Bugliesi, G. Castagna, and S. Crafa. Subtyping and Matching for Mobile Objects. In *ICTCS 2001 (7th. Italian Conference on Theoretical Computer Science)*, number 2202, Lecture Notes in Computer Science, pages 235–255, Torino, Italy, 2001. Springer.

[BC00] M. Bugliesi and G. Castagna. Mobile objects. In *7th Workshop on Foundation of Object-Oriented Languages*, Boston, 2000. (Informal electronic proceedings available on the web).

Chapitre III.2

[BC02] M. Bugliesi and G. Castagna. Behavioral typing for Safe Ambients. *Computer Languages*, 2002. A paraître.

[BC01] M. Bugliesi and G. Castagna. Secure Safe Ambients. In *Proc. of the 28th ACM Symposium on Principles of Programming Languages*, London, 2001. ACM Press.

Chapitre III.3

[VC99b] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In H. Bal, B. Belkhouche, and L. Cardelli, editors, *Internet Programming Languages*, number 1686 in Lecture Notes in Computer Science, pages 47–77. Springer, 1999.

[CGZ01] G. Castagna, G. Ghelli, and F. Zappa. Typing mobility in the Seal Calculus. In *CONCUR 2001 (12th. International Conference on Concurrency Theory)*, number 2154 in Lecture Notes in Computer Science, pages 82–101, Aahrus, Danemark, 2001. Springer.

Chapitre III.4

[BCC01a] M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *TACS 2001 (4th. International Symposium on Theoretical Aspects of Comuter Science)*, number 2215 in Lecture Notes in Computer Science, pages 38–63, Sendai, Japan, 2001. Springer.

[BCC01c] M. Bugliesi, G. Castagna, and S. Crafa. Reasoning about security in mobile ambients. In *CONCUR 2001 (12th. International Conference on Concurrency Theory)*, number 2154 in Lecture Notes in Computer Science, pages 102–120, Aahrus, Danemark, 2001. Springer.

1.4 Autres publications non couvertes par ce mémoire

Outre les publications suivantes [Cas96a, CGL93, Cas93b, Cas93a, CP96, Cas96b, Cas97a] qui ont été couvertes dans ma thèse de doctorat, il y a un certain nombre de publications qui sont parues simultanément à celles traitées dans cette thèse mais que, pour des raisons d'espace et pour garder une unité conceptuelle j'ai préferé ne pas inclure dans cette dissertation. Il s'agit de:

[LC96] L. Liquori and G. Castagna. A typed calculus of objects. In J. Jaffar and R. Yap, editors, *Proc. of 1996 Asian Computing Conference*, number 1179 in Lecture Notes in Computer Science, pages 129–141. Springer, 1996.

[Cas97b] G. Castagna. Unifying overloading and λ-abstractions: $\lambda^{\{\}}$. *Theoretical Computer Science*, 176(1-2):337–345, April 1997. Note.

[CRR98] G. Castagna, R. Rousseau, and J.-C. Royer. Fiabilité des langages à objets: le typage est-il suffisant? *l'Objet*, 4(1), 1998.

[VC99a] J. Vitek and G. Castagna. Mobile computations and hostile hosts. In *Journées Francophones des Langages Applicatifs*, 1999.

1.5 Remerciements

Comme le lecteur peut le constater la presque totalité de mes travaux sont le produit de collaborations. Ceci explique pourquoi dans le reste de ce mémoire j'abandonnerai le "je" à la faveur d'un "nous" qui correspond mieux à la réalité. En fait une partie importante du mérite de ce travail, y en eût-il un, va à mes coauteurs: Maria-Virginia Aponte, John Boyland, Kim Bruce, Michele Bugliesi, Luca Cardelli, Gang Chen, Silvia Crafa, Giorgio Ghelli, Gary Leavens, Giuseppe Longo, Luigi Liquori, Benjamin Pierce, Scott Smith, Jan Vitek, Francesco Zappa Nardelli.

Ensuite je voudrais remercier Giuseppe Longo, non seulement pour avoir accepté d'être le directeur de cette habilitation, mais surtout pour le guide et le conseil qu'il m'a toujours promptement prodigués: c'est avec grand plaisir que je lui renouvelle mes sentiments d'estime et d'amitié.

J'ai été très flatté par le fait que Martín Abadi, Gérard Boudol, Luca Cardelli et Pierre-Louis Curien aient acceptés d'être mes rapporteurs, et je leur suis sincèrement très reconnaissant pour l'honneur qui m'ont fait.

Encore un très grand merci à Mariangiola Dezani qui montre par sa présence dans ce jury une appréciation de mon travail qui me touche profondément.

Ce mémoire n'en serait pas là sans les encouragements, le soutien, les conseils et l'aide constants de Véronique, à qui je dois aussi la plus belle réussite de ma vie: notre fille Ilaria.

Chapitre 2

Notions préliminaires

Dans ce chapitre nous présentons quelques notions définies et étudiés dans [Cas97a] qui sont nécessaires à la compréhension des chapitres formant les parties I et II de ce rapport.

2.1 Surcharge et liaison tardive

Depuis plus de deux décennies, on distingue en théorie des types deux formes différentes de polymorphisme : le polymorphisme *paramétrique* et le polymorphisme *ad-hoc* [Str67]. Le polymorphisme *paramétrique* autorise l'écriture de fonctions dont le code peut être utilisé avec une infinité de types, alors que par le polymorphisme *ad-hoc* (ou *surcharge*), il est possible d'écrire des fonctions qui exécutent un code différent pour chaque type.

Les langages de programmation traditionnels offrent une forme très limitée de surcharge. Notamment, la sémantique d'un opérateur surchargé est toujours déterminée au moment de la compilation, en accord avec sa définition et avec les types de ses arguments.

Cette situation change dans les langages pour lesquels une relation de *sous-typage* a été définie. Nous rappelons qu'une relation de sous-typage est un préordre défini sur les types qui est utilisé par le contrôle des types pour typer un plus grand nombre de programmes. Son utilisation intuitive est que si un type A est en relation de sous-typage avec un type B (on dira que "A est un *sous-type* de B) alors il est possible d'utiliser toute expression de type A où une expression de type B est attendue. En particulier on pourra appliquer une fonction définie pour un paramètre de type B à un argument de type A.

Dans les langages avec sous-typage, le type dynamique d'une expression (c'est-à-dire le type d'un résidu d'une expression après un certain nombre de pas de réduction) n'est pas toujours identique à son type statique (c'est-à-dire le type de l'expression à la compilation). [1] Dans ce cas, le choix du code associé à une fonction surchargée peut varier selon le moment auquel le type de l'argument d'une fonction surchargée est examiné. Ainsi, on peut distinguer au moins deux disciplines pour déterminer le code associé aux appels de fonctions surchargées :

- La sélection s'opère en fonction du typage statique des arguments : l'information la moins précise sur le type de l'argument est utilisée. Nous appelons cette discipline *liaison précoce*.
- La sélection s'opère en fonction du typage dynamique des arguments et l'information de type utilisée est la plus précise concernant l'argument. Nous appelons cette discipline *liaison tardive*.

1. Comme exemple on peut considérer la fonction identité pour le type B: le type statique du résultat de cette fonction est toujours B, tandis que si cette fonction est appliquée à un argument de type A, sous-type de B, alors lors de l'exécution le résultat de l'application aura le type A.

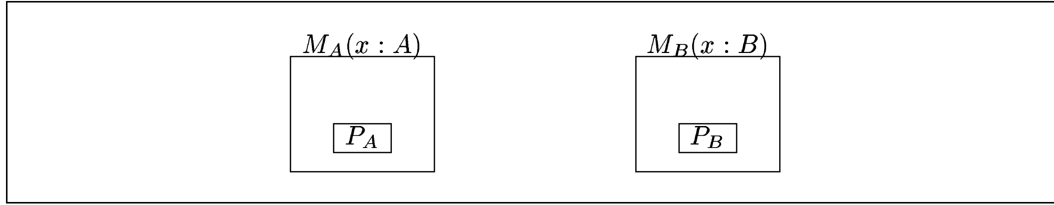


FIG. 2.1: *Deux programmes avec une partie commune*

L'ajout de la surcharge avec liaison précoce ne modifie pas substantiellement le pouvoir expressif du langage sous-jacent. *En revanche, la possibilité de surcharger des opérateurs combinée avec la liaison tardive enrichit le pouvoir expressif d'un langage, notamment en facilitant la réutilisation de code, et donnant lieu ainsi à un style de programmation incrémentale*. Nous illustrons ce dernier point par un exemple.

Nous partons d'un langage de programmation avec surcharge et sous-typage. Nous considérons deux programmes M_A et M_B, chacun avec un argument x, de type A pour M_A et de type B pour M_B, où B est un sous-type de A (noté $B \leq A$). Les deux programmes sont identiques sauf une partie variante qui traite l'argument x de manière différente. Nous notons P_A la partie variante de M_A et P_B celle de M_B. Cette situation est illustrée en figure 2.1. Avec la surcharge on peut réécrire ces deux programmes dans un unique programme M qui réutilise la partie commune de M_A et M_B et qui appelle un sous-programme surchargé P. Ce sous-programme exécute P_A ou P_B selon le type de son argument. Sans perte de généralité, nous supposons que P utilise le paramètre de M. Par ailleurs, le fait que $B \leq A$ implique que $M(x : A)$ accepte un argument soit de type A soit de type B. Cette situation est illustrée en figure 2.2. Dans le but de préciser la corrélation entre les

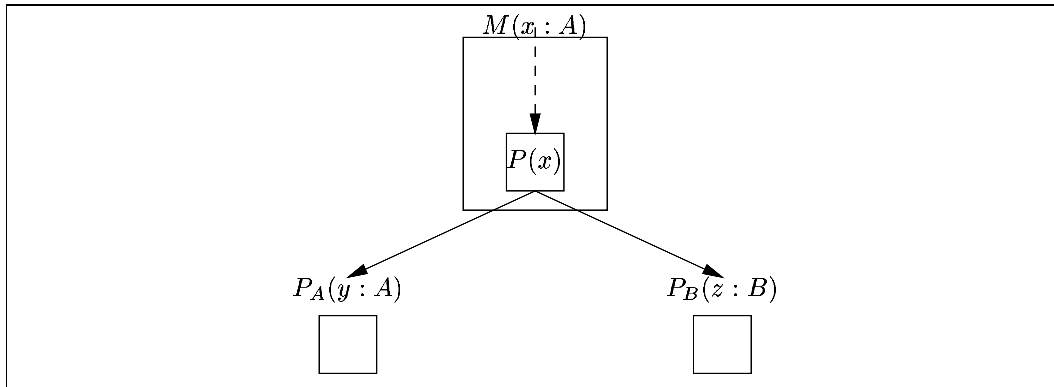


FIG. 2.2: *Surcharge et réutilisation*

programmes de la figures 2.1 et celui de la figure 2.2, nous utilisons les contextes du λ-calcul. Nous utilisons le contexte $\mathscr{C}[\,]$ pour dénoter la partie commune entre M_A et M_B. Ainsi, les programmes M_A et M_B de la figure 2.1 sont dénotés par :

$$\begin{array}{lcl} M_A & = & \lambda x\colon A.\mathscr{C}[P_A(x)] \\ M_B & = & \lambda x\colon B.\mathscr{C}[P_B(x)] \end{array} \tag{2.1}$$

tandis que le programme de la figure 2.2 est équivalent à :

$$M = \lambda x\colon A.\mathscr{C}[P(x)] \tag{2.2}$$

Dans cette dernière expression, le contexte $\mathscr{C}[\ ]$ correspond toujours au code partagé par M_A et M_B, et P est une fonction surchargée avec deux *branches* [2] P_A et P_B.

Il existe néanmoins un problème avec cette transformation : l'expression qui en résulte (et le programme de la figure 2.2) ne reproduit le comportement des expressions en (2.1) (ou des programmes en figure 2.1) que si le choix du code pour la fonction surchargé P se fait avec la discipline de liaison tardive.

En effet, considérons la définition de M donnée par (2.2). Si la liaison précoce est employée, la contrainte $x\colon A$ sur l'argument de M implique qu'un appel à cette fonction exécute toujours la branche P_A (A est le type statique de x). Autrement dit, avec la liaison précoce la définition de M dans (2.2) est équivalente à

$$\lambda x\colon A.\mathscr{C}[P_A(x)]$$

(c'est-à-dire, à M_A) même lorsque le paramètre actuel de M est typé par (un sous-type de) B. Ainsi, avec la liaison précoce, le schéma de surcharge de la figure 2.2 ne correspond pas au comportement du programme de départ. En réalité, avec la liaison précoce, la seule manière d'exécuter les différentes branches de P dans M est de définir M comme une fonction surchargée à deux branches M_A et M_B, ce qui entraîne le retour au schéma de programme de départ (voir (2.1)) et, par conséquent, la duplication du code correspondant à $\mathscr{C}[\ ]$.

Au contraire, si la liaison tardive est employée, le code à exécuter pour P est choisi seulement après substitution du paramètre formel par le paramètre actuel. Ainsi, avec la liaison tardive la définition de M dans (2.2) est équivalente à M_A lorsque M est appelée avec des arguments de type A, et elle est équivalente à M_B lorsque ses arguments sont typés par (un sous-type de) B. Avec la liaison tardive la fonction M dans (2.2) est *implicitement* surchargée avec deux branches ; grâce à la liaison tardive le code $\mathscr{C}[\ ]$ est partagé par ces deux branches.

En outre nous pouvons imaginer que le programme dans (2.2) ait été écrit en deux phases. D'abord les seules données traitées par le programme étaient de type A et P n'était formé que par le code P_A. Dans un deuxième temps on a spécialisé les données par l'introduction d'un sous-type B et pour cela on a aussi spécialisé le code en ajoutant à la définition de P le code P_B traitant les nouvelles données. Nous sommes donc en présence d'une *programmation incr émentale* (d'abord nous avons défini le code pour certaines données et après nous l'avons enrichi pour traiter des données nouvelles) et de *réutilisation du code* (car le code $\mathscr{C}[]$ défini à l'origine pour traiter des données A est réutilisé pour le traitement de données B).

2.2 Paradigme à objets

La réutilisation du code et un style de programmation incrémentale caractérisent de manière assez précise les paradigmes à objets. Vu l'étroite connexion avec la surcharge à liaison tardive il est possible de développer un modèle de ces paradigmes en utilisant cette notion seule. La première étape d'une telle étude consiste dans la définition d'un calcul formel qui capture la surcharge à liaison tardive, ce que nous faisons en section 2.2.1. Ensuite nous présentons le modèle des langages à objets induit par un tel calcul (section 2.2.2).

2.2.1 Le $\lambda\&$-calcul

Avant de définir le calcul sur lequel nous développons notre modèle, nous rappelons brièvement les principales définitions du λ-calcul simplement typé.

2. Nous considérons qu'une fonction surchargée est formée par l'ensemble de ses différents codes associés, et nous appelons *branche* chaque unité de code qui se trouve dans cet ensemble.

Le λ-calcul simplement typé est un formalisme logique pour exprimer des fonctions. Il constitue la base de toute étude formelle sur le typage des langages de programmation. Ce calcul est formé par trois catégories de termes : variables, abstractions et applications. Pour des raisons de simplicité, nous indiquons en exposant le type des variables ; ainsi x^T indique que la variable x est de type T. L'application est dénotée par un point ; ainsi, si M et N sont deux termes, l'application de M à N est notée $M \cdot N$ (ou parfois $M(N)$). L'abstraction permet la construction des fonctions ; ainsi, si M est un terme, $\lambda x^T.M$ est la fonction qui a comme paramètre formel x de type T, et comme corps M. La définition des termes est exprimée de manière plus compacte en utilisant la notation suivante :

$$M ::= x^T \mid \lambda x^T.M \mid M \cdot M$$

qui exprime que tout terme M est soit une variable, soit l'abstraction d'un autre terme soit l'application de deux termes.

Par exemple la fonction identité pour les réels, `fun(x:real) = x` est exprimée par le terme $\lambda x^{real}.x^{real}$. Le type d'une fonction qui associe à des arguments de type A des valeurs de type B est dénoté $A \to B$ (type *flèche*). Ainsi la fonction identité ci-dessus est de type *real*$\to$*real*. On peut exprimer des fonctions plus complexes. Ainsi, la fonction de composition $\circ$ entre deux fonctions, l'une de A vers B (type $A \to B$) et l'autre de B vers C (type $B \to C$) est définie par le terme $\lambda f^{A\to B}.\lambda g^{B\to C}.\lambda x^A.(g \cdot (f \cdot x))$.[3] L'application de ce terme à deux fonctions et un argument (de types appropriés) retourne la composition de ces fonctions appliquée à l'argument. Ce comportement est établi de façon formelle par la définition d'une relation binaire de réduction, $\rightarrow$:

$$(\lambda x^T.M) \cdot N \quad \rightarrow \quad M[x^T := N]$$

Le symbole $\rightarrow$ se lit "se réduit à" et la notation $M[x^T := N]$ dénote le terme M dans lequel toute occurrence (libre) de la variable x^T a été remplacée par N. La relation $\rightarrow$ décrit la sémantique opérationnelle du calcul. En particulier, la règle ci-dessus, dite de β-réduction, exprime le fait que lorsqu'on applique une fonction de paramètre x et de corps M à un argument N, il faut remplacer dans le corps de la fonction le paramètre formel par l'argument. Cette réduction peut avoir lieu dans tout contexte, c'est-à-dire même si l'expression à réduire (celle à gauche de $\rightarrow$) n'est qu'un sous-terme (ceci prend le nom de *fermeture compatible*: voir [Bar84] pour plus de détails). Ainsi, par exemple, $(\lambda x^T.x) \cdot y \rightarrow y$ (β-réduction) implique $\lambda y^T.((\lambda x^T.x) \cdot y) \rightarrow \lambda y^T.y$ (fermeture compatible).

Il ne reste plus qu'à montrer comment il est possible d'assigner à un terme M un type T (noté $M : T$). Ceci est décrit par un axiome et deux règles :

(ax) Pour tout T, $x^T : T$

(r1) Si $M : S \to T$ et $N : S$ alors $(M \cdot N) : T$

(r2) Si $M : T$ alors $(\lambda x^S.M) : S \to T$

En théorie des types, on préfère utiliser une autre notation pour définir les axiomes et les règles, celle de la *déduction naturelle*[4]. Dans cette notation, l'implication logique est dénotée par une barre horizontale qui sépare les prémisses (en haut) de la conclusion (en bas). Ainsi les trois règles ci-dessus sont habituellement écrites de la manière suivante.

[TAUT] $$\frac{}{x^T : T}$$

[$\to$ELIM] $$\frac{M : S \to T \qquad N : S}{M{\cdot}N : T}$$

[$\to$INTRO] $$\frac{M : T}{\lambda x^S.M : S \to T}$$

3. Nous avons omis les exposants dans le corps de la fonction.

4. Strictement parlant il ne s'agit pas de notations mais de *systèmes de déduction*.

Dans la suite nous allons omettre la barre horizontale dans les axiomes.

Pour plus de détails sur le lambda-calcul typé et sur d'autres notions que nous utiliserons dans la suite nous renvoyons le lecteur à l'excellente introduction de Cardelli et Wegner [CW85].

Nous proposons d'étendre le λ-calcul simplement typé afin de pouvoir exprimer les fonctions surchargées avec la liaison tardive.

Une fonction surchargée est formée d'un ensemble de fonctions ordinaires (c'est-à-dire des λ-abstractions), chacune constituant une branche différente. Pour relier ces fonctions nous avons choisi le symbole & (d'où le nom de λ&-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 : à la première est associée une substitution, à la deuxième une sélection (qui produit une application ordinaire). 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 ajoutant de nouvelles branches par &. Donc dans le terme précédent, $(M\&N)$, le sous-terme M est une fonction surchargée tandis que N est une fonction ordinaire (une branche). Ainsi, le terme

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

dénote une fonction surchargée avec n branches $M_1, M_2, \ldots, M_n$ (en l'absence de parenthèses on considérera que & associe à gauche).

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 types flèche. 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

$$(\varepsilon\&M_1\&\ldots\&M_n)\bullet N \rightarrow^* M_j\cdot N \tag{2.3}$$

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

Nous introduisons sur les types une relation de sous-typage. Intuitivement $U \leq V$ si tout terme de type U peut être utilisé de manière "sûre" par rapport aux types (c'est-à-dire sans que des erreurs de type puissent survenir pendant l'exécution) 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, autrement dit, tant que l'exécution d'un terme d'un certain type produira un résultat ayant comme type un sous-type du type originel.

La relation de sous-typage pour les types flèche est bien connue : covariance à droite et contravariance à gauche (voir les règles de sous-typage plus loin dans cette section) ; 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 (2.3) 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 $U_j = \min_{i\in I}\{U_i | U \leq U_i\}$.[5]

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 tout i, j dans I, il satisfait les conditions suivantes

$$U_i \leq U_j \quad \Rightarrow \quad V_i \leq V_j \tag{2.4}$$

$$U \text{ est maximal dans } LB(U_i, U_j) \quad \Rightarrow \quad \exists !\, h \in I \text{ tel que } U_h = U \tag{2.5}$$

où $LB(S, T)$ est l'ensemble des minorants communs de S et T.

La condition (2.4) 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 (c'est-à-dire le résultat du calcul 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 $V_2 \leq V_1$ doit être vérifiée.

La condition (2.5) 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 $U_j = \min_{i\in I}\{U_i | U \leq U_i\}$. La condition (2.5) est une condition nécessaire et 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 (2.3) ne soit effectuée que si N est *clos* (sans variables libres, c'est-à-dire des variables non abstraites par un λ) et en *forme normale* (il ne peut plus se réduire).

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

Pré-types

Nous définissons les pré-types, parmi lesquels nous irons dans la suite sélectionner les types

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

Dans la définition ci-dessus, A représente les *types atomiques*, ce qui correspond dans un langage de programmation aux types prédéfinis (tels que *real*, *bool*, ...) et, dans plusieurs langages à objets, aux noms des classes.

Sous-typage

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

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

La règle de gauche établit que deux types flèche (les types des fonctions) sont l'un plus petit que l'autre si leurs co-domaines sont dans la même relation (covariance) et leur domaines sont dans la relation inverse (contravariance) ; cette règle est due à [Car88]. La règle de droite traduit d'une façon formelle l'intuition qu'un premier type surchargé est plus petit qu'un second type surchargé si le premier possède, pour toute branche du second, une branche plus petite

5. Soit E un ensemble muni d'un pré-ordre $\leq$. Nous utilisons $\min E$ pour dénoter le plus petit élément de l'ensemble E par rapport à $\leq$, s'il existe, et $\inf E$ pour dénoter la borne inférieure de E, c'est-à-dire le plus grand des minorants de E par rapport à $\leq$, s'il existe. Nous utilisons aussi à la place de $\min\{U_i | i \in I, \ldots\}$ la notation plus compacte $\min_{i\in I}\{U_i | \ldots\}$.

Types
Les types du calcul sont les pré-types qui satisfont les conditions décrites auparavant :

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) $(U$ maximal dans $LB(U_i, U_j) \quad \Rightarrow \quad \exists!\ h \in I$ tel que $U_h = U)$
   alors $\{U_i \to V_i\}_{i \in I} \in$ **Types**

Termes
Les termes sont obtenus en ajoutant aux termes du λ-calcul les trois termes pour les fonctions surchargées

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

Notons que dans la définition formelle du calcul nous avons annoté & par un type V. Ce type sert à mémoriser le type de la fonction surchargée et a été introduit principalement pour des raisons techniques (pour la confluence du calcul, voir [CGL92]). N'ayant pas une importance fondamentale dans le contexte de ce rapport, il sera souvent omis dans la suite.

Typage
La relation de typage est définie par les règles suivantes, où tout pré-type apparaissant dans les règles est un type bien formé (c'est-à-dire appartenant à **Types**) :

[TAUT] $\quad x^T : T$

[TAUT$_\varepsilon$] $\quad \varepsilon : \{\}$

[$\to$INTRO]

$$\frac{M : T}{\lambda x^U.M : U \to T}$$

[$\{\}$INTRO]

$$\frac{M : W_1 \leq \{U_i \to T_i\}_{i \leq (n-1)} \quad N : W_2 \leq U_n \to T_n}{(M \&^{\{U_i \to T_i\}_{i \leq n}} N) : \{U_i \to T_i\}_{i \leq n}}$$

[$\to$ELIM$_\leq$]

$$\frac{M : U \to T \quad N : W \leq U}{M{\cdot}N : T}$$

[$\{\}$ELIM]

$$\frac{M : \{U_i \to T_i\}_{i \in I} \quad N : U \quad U_j = \min_{i \in I}\{U_i | U \leq U_i\}}{M \bullet N : T_j}$$

Les règles [TAUT] et [$\to$INTRO] sont les mêmes que celles du λ-calcul simplement typé. La règle [$\to$ELIM] a été modifiée de façon à permettre l'application d'une fonction à un argument dont le type est un sous-type du domaine de la fonction : grâce au sous-typage on sait qu'utiliser un argument de type W là où un argument de type U plus grand que W était attendu ne posera pas de problèmes. La règle [TAUT$_\varepsilon$] établit que la fonction surchargée vide a le type surchargé vide. Une fonction surchargée est bien typée si ses composants sont typés par des sous-types des types en exposant du & (règle [$\{\}$INTRO]). Lorsqu'une fonction surchargée est appliquée à un argument (règle [$\{\}$ELIM]), le type de l'application est le co-domaine de la branche qui serait sélectionnée à cet instant (même si l'argument n'est pas en forme normale close).

Réduction
La réduction $\twoheadrightarrow$ est la fermeture compatible de la réduction suivante :

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

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

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

Nous avons ajouté à la règle qui décrit l'application d'une fonction (la β-réduction) celle qui décrit l'application d'une fonction surchargée : si la branche sélectionnée est la plus à droite nous passons l'argument à cette branche ; sinon nous continuons la sélection sur les branches restantes. Il faut noter que la sélection utilise le type dynamique de l'argument et le type, on peut dire, statique de la fonction mémorisé à l'index de &.

Un certain nombre de propriétés théoriques ont été prouvées pour ce calcul (voir [CGL92]).

Théorèmes Principaux

1. *Élimination de la subsumption* : le calcul admet aussi une présentation équivalente avec la règle de subsumption ($M : S$ et $S \leq T$ impliquent $M : T$). [6]
2. *Admissibilité de la transitivité et de la réfléxivité* : l'ajout de la règle de transitivité (c'est-à-dire $T_1 \leq T_2$ et $T_2 \leq T_3$ impliquent $T_1 \leq T_3$) ou de reflexivité ($T_1 \leq T_2$ implique $T_2 \leq T_1$) ne modifie pas la relation de sous-typage.
3. *Unicité du type* : chaque terme bien typé possède (avec le système actuel) un seul type.
4. *"Subject Reduction"* : soit $M : S$; si $M \twoheadrightarrow^* N$ alors $N : T$ et $T \leq S$.
5. *Confluence* : des termes égaux se réduisent à un terme commun.

Plutôt que de décrire en détail la signification de ces propriétés, il est intéressant, pour le lecteur non spécialiste, de connaître les conséquences pratiques de ces résultats. Les deux premiers résultats assurent qu'il est possible de programmer un algorithme de contrôle de type pour ce système, ce qui montre la faisabilité pratique du système. La troisième propriété assure que la sélection d'une branche pendant l'application d'une fonction surchargée est faite de manière déterministe. La quatrième propriété est sûrement la plus importante d'un point de vue pratique car elle prouve que toutes les erreurs de type sont récupérées par le typage statique. Plus précisément, elle prouve que tout terme bien typé ne peut se réduire que dans un autre terme bien typé, ce qui implique qu'un terme statiquement bien typé ne peut pas causer d'erreurs de type pendant son exécution. La dernière propriété assure que la sémantique définie par $\twoheadrightarrow$ est déterministe. Pour terminer avec les propriétés théoriques, il est intéressant de noter que le calcul n'est pas fortement normalisant (c'est-à-dire qu'il peut y avoir des réductions infinies). En fait il est possible de coder en λ& un opérateur de point fixe de type $(T \to T) \to T$ pour tout type T, ce qui, en terme de langages de programmation, signifie la possibilité de définir des fonctions récursives.

2.2.2 $\lambda\&$ et programmation orientée objet

Nous supposons que le lecteur est à peu près familier avec les concepts de la programmation orientée objet, tels que objet, méthode et message. Intuitivement, un objet est une unité de programmation qui associe des données avec les opérations qui peuvent utiliser ou modifier ces données. Ces opérations sont appelées *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*. Un message est le nom d'une méthode définie pour l'objet en question.

L'envoi 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 intéragissent en s'échangeant des messages. Chaque langage possède sa propre syntaxe pour l'envoi de message. Dans ce chapitre nous utilisons la notation suivante :

[*destinataire message*]

qui est celle de Objective-C (ce qui correspond à la notation *destinataire.message*`()` de Java ou *destinataire*`->`*message* de C++). Le *destinataire* est un objet ou, plus généralement, une expression dont

6. Nous n'avons pas utilisé la règle de *subsumption* [Car88], mais la présentation algorithmique car cela nous paraissait plus simple.

le résultat est un objet ; lors de l'envoi d'un message, le système sélectionne parmi les méthodes définies pour l'objet en question, celle dont le nom correspond au message ; il est (très) souhaitable que l'existence de cette méthode puisse être vérifiée statiquement (typiquement, lors de la compilation) et de manière certaine par un programme de vérification des types. Une telle vérification permet non seulement d'éviter le contrôle de types pendant l'exécution (ce qui implique des économies en temps et en mémoire) mais aussi constitue un premier outil de contrôle (partiel) de la correction d'un programme.

La majeure partie de la recherche sur l'étude et la vérification des types dans la programmation orientée objet utilise comme idée de départ le modèle à enregistrements de [Car88]. L'idée est de considérer un objet comme un enregistrement dont chaque champ contient une des méthodes définies pour l'objet ; chaque champ est étiqueté par le message correspondant à la méthode contenue. Ainsi l'envoi de message est considéré comme la sélection d'un champ d'un enregistrement. Une plus ample présentation de ce modèle est donnée dans le livre édité par Gunter et Mitchell [GM94].

Dans le reste de ce chapitre nous décrivons de manière informelle un modèle alternatif à celui des enregistrements et basé sur la surcharge.

L'idée principale de ce modèle est de considérer l'envoi 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], Cecil [CL95, Cha92] et Dylan [Dyl92]). Toutefois les fonctions ordinaires ne suffisent pas à formaliser cette approche. Le fait qu'une méthode *appartienne* à un objet spécifique implique que la sémantique de l'envoi de message est tout à fait différente de celle de l'application fonctionnelle 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. Sous l'hypothèse que le type d'un objet est sa classe, cela revient à dire que *les messages dénotent des fonctions surchargées*, car le code à exécuter est choisi sur la base du type (la classe) de l'argument (le destinataire). Chaque méthode associée à un message m constitue une *branche* (c'est-à-dire le code défini pour un certain type) de la fonction surchargée dénotée par m.
2. *Liaison tardive* : la deuxième différence entre l'application d'une fonction et l'envoi 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 (ou *dynamique*), est un des traits saillants de la programmation orientée objet. Dans notre approche elle naît de la combinaison de la surcharge et du sous-typage. Par exemple, 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, l'envoi du message suivant
$$\texttt{fun}(x\colon \mathit{Figure}) = (\ldots\ \texttt{[}x\ \mathit{dessine}\texttt{]}\ \ldots)$$
est toujours effectué en utilisant la méthode définie pour les figures. En revanche, avec la liaison tardive, la méthode est choisie *après* que la fonction ait été appliquée, selon que x est lié à un cercle, à un carré ou à une figure.

Donc nous retrouvons là notre surcharge à liaison tardive. Nous pouvons donc utiliser $\lambda\&$ pour modéliser les langages à objets. Nous ne donnons pas ici la traduction détaillée d'un langage à objets dans $\lambda\&$: celle-ci peut être trouvée dans [Cas95b].

Tout d'abord il faut noter que dans $\lambda\&$ il est possible de coder les produits cartésiens, les enregistrements simples (ceux de [Car88]), et les enregistrements extensibles (c'est-à-dire des enregistrements auxquels de nouveaux champs peuvent être ajoutés ; voir [Wan87, Rém89, CM91]).

Le nom d'une classe est un type atomique, que l'on utilise pour typer les instances de la classe même. En outre, chacun de ces types atomiques est associé à un *type-représentation* qui, comme nous le montrerons plus loin, implante les variables d'instance.

Les conditions (2.4) et (2.5) ont dans les langages à 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* (en anglais *overrides*) celle de C_2. La condition (2.4) 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 comparables mais qu'il existe une classe C_3 qui est sous-classe des deux ($C_3 \leq C_1$ et $C_3 \leq C_2$) et si en plus il n'y a aucune superclasse de C_3 qui soit une sous-classe de C_1 et C_2 (C_3 est maximale) alors C_3 a été définie par *héritage multiple* de C_1 et C_2. La condition (2.5) impose qu'une branche soit définie dans *mesg* pour C_3, c'est-à-dire qu'en cas de conflit d'héritage multiple [DH89, DHH+95] la méthode doit être redéfinie [7].

Voyons ceci sur un exemple. Considérons la classe `2DPoint` avec deux variables d'instance à valeurs 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
  state
    x:Int;
    y:Int;
  methods

     ⋮
```

```
class 3DPoint is 2DPoint
  state
    x:Int;
    y:Int;
    z:Int
  methods
     ⋮
```

où à la place des pointillés se trouvent les définitions des méthodes (la déclaration de `x` et `y` dans `3DPoint` pouvait être omise car ces variables sont héritées de `2DPoint`). 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 $\langle\!\langle x\colon \text{Int}; y\colon \text{Int}; z\colon \text{Int}\rangle\!\rangle$ et $\langle\!\langle x\colon \text{Int}; y\colon \text{Int}\rangle\!\rangle$. [8] Il faut noter que *3DPoint*$\leq$*2DPoint* est compatible avec le sous-typage des types-représentation correspondants. Ces deux types atomiques sont utilisés pour typer les instances respectives des deux classes et intuitivement peuvent être considérés comme des noms pour leurs types-représentation (pour plus de détails voir [CGL92]).

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

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

masquée 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

$$norme \equiv (\ \lambda self^{2DPoint}.\sqrt{self.x^2 + self.y^2} \\ \&\ \lambda self^{3DPoint}.\sqrt{self.x^2 + self.y^2 + self.z^2}\ )$$

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

7. Ceci est l'une de deux manières utilisées pour gérer l'héritage multiple, et correspond à la solution adoptée, par exemple, par les langages Eiffel [Mey91] et O_2 [BDK92]. L'autre manière est celle utilisée par le langage CLOS et qui se base sur l'utilisation de listes de priorités (*class precedence lists*), ce qui correspond à établir des ordres sur les classes —locaux à une classe dans le cas de CLOS, ou globaux si l'on veut prendre en compte le typage— qui seront utilisés pour résoudre la sélection [DH89, DHH+95].

8. Les parenthèses $\langle\ \rangle$ et $\langle\!\langle\ \rangle\!\rangle$ sont une notation standard pour les expressions et les types enregistrements, respectivement.

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 :

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

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* :

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

Une telle définition n'est pas bien typée, car $\{\mathit{2DPoint} \rightarrow \mathit{2DPoint}\ , \mathit{Color} \rightarrow \mathit{Color}\}$ ne satisfait pas la condition (2.5) ; en réalité en appliquant *efface* à un objet de classe *ColorPoint* on ne saurait pas quelle méthode choisir. Par conséquent la condition (2.5) impose l'ajout d'une méthode pour *ColorPoint* et donc

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

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 envoie un message de type $\{C_i \rightarrow T_i\}_{i \in I}$ 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 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 (la condition (2.5) exige l'existence de ce minimum).

Sélection multiple

L'un des avantages de modéliser les messages par des fonctions surchargées est que, ces dernières étant des valeurs de première classe (c'est-à-dire qu'elles peuvent être utilisées dans n'importe quel contexte et donc, en particulier, peuvent être passées comme arguments à des fonctions ou en être le résultat), les messages sont aussi de première classe. Il devient donc possible, par exemple, d'écrire des fonctions (même surchargées) qui prennent comme argument un message ou le rendent comme résultat.

Mais l'avantage le plus intéressant que revêt cette forme de programmation objet est la possibilité de pouvoir utiliser la sélection multiple [9] : 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 (pour plus de détails sur les méthodes binaires voir [BCC+96]). 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 :

$$\begin{array}{l} \mathit{EqPoint} \equiv \langle\!\langle x\colon \text{Int};\ y\colon \text{Int};\ equal\colon \mathit{EqPoint} \rightarrow \text{Bool}\rangle\!\rangle \\ \mathit{ColEqPoint} \equiv \langle\!\langle x\colon \text{Int};\ y\colon \text{Int};\ c\colon \text{String};\ equal\colon \mathit{ColEqPoint} \rightarrow \text{Bool}\rangle\!\rangle \end{array}$$

Nous avons ici utilisé les noms *EqPoint* et *ColEqPoint* pour bien les différencier de *2DPoint* et *ColorPoint* du modèle à surcharge: dans le premier cas (objets-comme-enregistrements) il s'agit d'enregistrements récursifs qui représentent les méthodes présentes dans une classe, tandis que dans

9. En anglais *multiple dispatch*, c'est-à-dire la possibilité de sélectionner une méthode en tenant compte non seulement de la classe du destinataire du message mais aussi de celle d'autres arguments, comme le font les fonction génériques de CLOS.

le deuxième cas (message-comme-fonctions-surchargées) il s'agit d'enregistrements non-récursifs qui ne représentent que l'état d'une classe, et dont la définition ne dépend donc pas des méthodes présentes.

Un type enregistrement est un sous-type d'un autre type enregistrement s'il possède *au moins* tous les champs du second et si chacun de ces champs est un sous-type du champ correspondant dans le second. Ainsi *ColEqPoint* est un sous-type de *EqPoint* si et seulement si le type du champ *equal* dans le premier est un sous-type du type du champ *equal* dans le second. Mais à cause de la contravariance de la flèche, le type d'*equal* dans *ColEqPoint* n'est pas inférieur à celui d'*equal* dans *EqPoint* et donc $\mathit{ColEqPoint} \not\leq \mathit{EqPoint}$. [10]

Considérons maintenant le même exemple dans $\lambda\&$. Nous avons déjà rencontré les types atomiques *2DPoint* et *ColorPoint*. On peut continuer à les utiliser parce que, contrairement à ce qui se passe avec les enregistrements, l'ajout d'une méthode à une classe ne change pas le type des instances. En $\lambda\&$, une définition telle que

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

ne possède pas un type bien formé : $\mathit{ColorPoint} \leq \mathit{2DPoint}$ donc la condition (2.4) requiert que $\mathit{ColorPoint} \to \text{Bool} \leq \mathit{2DPoint} \to \text{Bool}$ ce qui n'est pas vrai à cause de la contravariance de la flèche (pour laquelle il faudrait $\mathit{2DPoint} \leq \mathit{ColorPoint}$). 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 lorsque 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 $\lambda\&$, 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 :

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

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. Dans ce cas *equal* est appelée une *multi-m éthode*.

Une autre caractéristique intéressante de ce modèle est que, contrairement au modèle à enregistrements, il permet d'ajouter une méthode à une classe C déjà existante sans affecter le type de ses objets (comme cela est possible par exemple en Dylan ou CLOS). Ceci est possible grâce au fait que, dans ce modèle, le typage des objets d'une classe ne dépend pas des méthodes de la classe (ce qui n'est pas vrai dans le modèle à enregistrements, où les types des objets ne sont pas atomiques). En fait, si la méthode doit être associée 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* que n'ait été définie la branche de m pour C. Cet aspect est très important quand on a à gérer des données persistantes, car on peut ainsi modifier le schéma logique des données (en ajoutant des fonctionnalités) sans devoir modifier les applications déjà écrites.

10. Pour vérifier que la contravariance est bien nécessaire le lecteur peut considérer le terme suivant : $(\lambda x^{EqPoint}.\lambda y^{EqPoint}.(x.\mathit{equal}(y)))\cdot M\cdot N$, où M : *ColEqPoint* et N : *EqPoint*. Si $\mathit{ColEqPoint} \leq \mathit{EqPoint}$ alors ce terme est statiquement bien typé mais il cause une erreur de type à l'exécution.

Première partie

Surcharge et langages à objets séquentiels

L'un des avantages du modèle d'objets basé sur la surcharge présenté dans le chapitre précédent est celui de clarifier l'utilisation de la covariance et de la contravariance dans la pratique de la programmation à objets. Dans cette partie nous allons étudier cet aspect du modèle en profondeur (chapitre I.1) et voir ses applications pratiques sur des langages commerciaux (chapitre I.2).

Chapitre 1

Typage des langages objets avec classes: l'utilisation de la covariance

Articles de référence: [Cas95a, BCC$^+$96]

L'un des apports du modèle présenté dans le chapitre précédent est celui de clarifier les rôles que jouent la covariance et la contravariance dans le sous-typage (*equal* en constitue un bel exemple) : comme il est expliqué en détail dans [Cas95a], la contravariance 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 dernier cas, la nouvelle branche *ne se substitue pas* à l'ancienne branche mais plutôt elle la *masque* aux objets de certaines classes. En fait, notre formalisation montre très clairement que la question "contravariance 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 de type T. Cette information est utilisée par l'application : soit f une fonction de type $T \rightarrow U$, 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 relation 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 $T_1 \rightarrow T_2$ 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 $S \leq T_1 \rightarrow T_2$ alors S doit être de la forme $S_1 \rightarrow S_2$), 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). Évidemment, 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 [BDK92] fonctionne même sans la contravariance).

Le masquage correspond à un tout autre phénomène : on a un identificateur m (en l'occurrence un *message*) qui identifie par exemple deux fonctions $f\colon A \rightarrow C$ et $g\colon B \rightarrow D$ (A et B incomparables) ; cet identificateur peut être appliqué à une expression e ; cette application est résolue par l'envoi 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 contrôleur de types 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 ; dans ce cas D doit être un type qui puisse se substituer à C (dans le sens de la substitutivité définie plus haut), c'est-à-dire $D \leq C$. 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 ; en fait 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'un point de vue plus pratique : une méthode a des paramètres ; la classe de chaque paramètre peut être ou ne pas être prise en compte pour la sélection de la méthode. Lorsqu'on masque cette méthode, les paramètres dont le type est pris en compte pour la sélection doivent être masqués de manière covariante (chaque paramètre dans la nouvelle méthode doit posséder un type plus petit que celui du paramètre correspondant de l'ancienne), les autres de manière contravariante (dans la nouvelle méthode les paramètres correspondants doivent posséder un type plus grand).

Ceci se traduit en termes de typage des langages à objets de la manière suivante. Considérons un message m qui est appliqué (c'est-à-dire envoyé) à n objets $e_1 \ldots e_n$, où e_i est une instance de la classe C_i. Supposons que pour la sélection de la méthode on ne veuille prendre en compte que les classes des premiers k objets. Ce schéma de sélection peut être exprimé par la syntaxe suivante:

$$m(e_1, \ldots, e_k | e_{k+1}, \ldots, e_n).$$

Si le type de m est $\{S_i \to T_i\}_{i \in I}$, alors l'expression ci-dessus signifie que nous voulons sélectionner la méthode dont le domaine est $\min_{i \in I}\{S_i \mid (C_1 \times \ldots \times C_k) \leq S_i\}$ pour ensuite lui passer l'ensemble des n arguments. Le type $S_j \to T_j$, de la branche sélectionnée doit alors avoir la forme suivante :

$$\underbrace{(A_1 \times \ldots \times A_k)}_{S_j} \to \underbrace{(A_{k+1} \times \ldots \times A_n) \to U}_{T_j}$$

où $C_i \leq A_i$ pour $1 \leq i \leq k$, et $A_i \leq C_i$ pour $k < i \leq n$.[1] Si nous voulons spécialiser la branche sélectionnée par une branche plus précise nous devons alors, comme expliqué ci-dessus, masquer $A_1 \ldots A_k$ de manière covariante (pour spécialiser la branche) et $A_{k+1} \ldots A_n$ de manière contravariante (pour que l'exécution soit sûre du point de vue des types).

Cette analyse nous a permis d'introduire la covariance dans les modèles basés sur les enregistrements (voir [Cas95a]). L'idée est simple et consiste à utiliser à l'intérieur des champs des enregistrements des fonctions surchargées. Une des conséquences de cette introduction est qu'il est possible de donner une solution satisfaisante au problème des méthodes binaires dans le modèle à enregistrements (voir [BCC+96]). En fait, l'analyse ci dessus nous indique très clairement que dans le cas de méthodes binaires —c'est-à-dire les méthodes qui ont un paramètre de la même classe que celle du destinataire du message— il n'est pas possible de sélectionner la bonne méthode en ne connaissant que le type d'un seul argument. Autrement dit, dans le cas de la méthode *equal* (voir page 21) il faut connaître le type des *deux* arguments pour pouvoir choisir la définition de *equal* appropriée. Or le problème dans les langages avec sélection simple ("single dispatching") est justement que la sélection se base sur le type d'un seul argument: le destinataire du message. C'est pourquoi dans le modèle à enregistrements *ColEqPoint* $\not\leq$ *EqPoint*. Toutefois cette même analyse nous suggère qu'il suffit d'utiliser une fonction surchargée pour *equal* pour pourvoir typer les méthodes binaires dans le modèle à enregistrements. Et en fait si nous modifions les définitions de *EqPoint* et *ColEqPoint* de page 21 de la manière suivante :

1. En fait, par la condition de covariance (2.4) page 16, toute méthode dont le domaine est compatible avec le type des arguments doit avoir cette forme.

$$
\begin{aligned}
\textit{EqPoint} \equiv\ & \langle\!\langle x\colon \text{Int};\\
& y\colon \text{Int};\\
& \mathit{equal}\colon \{\textit{EqPoint} \to \text{Bool}, \textit{ColEqPoint} \to \text{Bool}\}\ \rangle\!\rangle\\
\textit{ColEqPoint} \equiv\ & \langle\!\langle x\colon \text{Int};\\
& y\colon \text{Int};\\
& c\colon \text{String};\\
& \mathit{equal}\colon \{\textit{EqPoint} \to \text{Bool}, \textit{ColEqPoint} \to \text{Bool}\}\ \rangle\!\rangle
\end{aligned}
$$

nous obtenons *ColEqPoint*$\leq$ *EqPoint*. Toutefois une telle definition violerait l'un de principes de modularité de la programmation à objets car le code de la classe *EqPoint* dépendrait de sa sousclasse *ColEqPoint*. Mais encore une fois la relation de soutypage pour les fonctions surchargées nous vient en aide car *ColEqPoint*$\leq$ *EqPoint* est valable aussi pour les définitions suivantes:

$$
\begin{aligned}
\textit{EqPoint} \equiv\ & \langle\!\langle x\colon \text{Int};\\
& y\colon \text{Int};\\
& \mathit{equal}\colon \{\textit{EqPoint} \to \text{Bool}\}\ \rangle\!\rangle\\
\textit{ColEqPoint} \equiv\ & \langle\!\langle x\colon \text{Int};\\
& y\colon \text{Int};\\
& c\colon \text{String};\\
& \mathit{equal}\colon \{\textit{EqPoint} \to \text{Bool}, \textit{ColEqPoint} \to \text{Bool}\}\ \rangle\!\rangle
\end{aligned}
$$

Par cette transformation nous obtenons que la méthode *equal* pour *ColEqPoint* est associée à deux codes différents, l'un qui est exécuté avec les arguments de type *EqPoint*, l'autre avec les arguments de type plus petit ou égal à *ColEqPoint*. En pratique quand on veut spécialiser une méthode de manière covariante il ne suffit pas de donner la nouvelle définition de la méthode mais puisque cette nouvelle méthode peut être utilisée où l'ancienne est attendue il faut définir un deuxième code qui s'occupera des arguments prévus pour l'ancienne méthode et que le nouveau code ne sait pas utiliser. Autrement dit une telle transformation requiert que, lorsque nous masquons une méthode binaire, nous définissions la méthode masquante non seulement pour des arguments ayant comme type la nouvelle classe (dans l'exemple *ColEqPoint*) mais aussi pour des arguments ayant comme type l'ancienne classe (dans l'exemple *EqPoint*), comme pour les multi-méthodes. Nous sommes donc en présence de multi-méthodes qui au lieu de constituer des définitions globales, sont *encapsul ées* dans les objets (ou dans leurs classes), d'où leur nom de *encapsulated multi-methods* que nous avons introduit dans [BCC+96].

Dans le langage à objets jouet que nous avons utilisé pour les exemples, cela revient à admettre les déclarations suivantes (où une fonction est considérée comme une fonction surchargée d'une seule branche) :

```
class 2DPoint
  state
    x:real=0,
    y:real=0
  methods
    xVal = self.x;
    yVal = self.y;
    equal(p:2DPoint) = return( (self.x == [p xVal])
                            and (self.y == [p yVal]) );
  interface
    x:real;
    y:real;
```

```
      equal:2DPoint ->Bool
endclass

class ColorPoint is 2DPoint
   state
      c:string = "red"
   methods
     cVal = self.c;
     equal(p:2DPoint) = return( (self.x == [p xVal])
                             and (self.y == [p yVal]))
     equal(p:ColorPoint) = return( (self.x == [p xVal])
                                and (self.y == [p yVal])
                                and (self.c == [p cVal]));
   interface
     cVal:string;
     equal:{2DPoint->Bool,ColorPoint ->Bool}
endclass
```

Avec ces déclarations, quand le message *equal* est envoyé à un objet de classe *ColorPoint*, on sélectionnera dynamiquement la méthode à exécuter sur la base du type de l'argument de *equal*. Si l'argument est de type *2DPoint*, la première définition de *equal* est utilisée ; s'il est de type inférieur ou égal à *ColorPoint* la deuxième définition est utilisée.

Il faut remarquer que, grâce à la règle de sous-typage pour les fonctions surchargées, deux définitions sont toujours suffisantes pour la spécialisation des méthodes binaires. Si par exemple nous avions une suite de sous-classes $2DPoint \leq 3DPoint \leq 4DPoint$, etc., il suffirait pour le sous-typage de définir dans la classe *nDPoint* deux méthodes *equal* l'une pour des arguments de type *nDPoint* (la méthode binaire) et l'autre pour des arguments de type *2DPoint* (qui gérerait tous les autres cas).

Il faut aussi remarquer que rien n'empêche d'ajouter à un langage avec des multi-méthodes encapsulées (comme l'exemple ci-dessus) les multi-méthodes "pures" —fonction génériques— comme on les trouve en CLOS. Dans ce cas nous obtenons un paradigme qui combine les caractéristiques des langages à dispatch simple avec celles des langages à dispatch multiple: en n'écrivant que des classes sans méthodes on obtiendrait un style de programmation à la CLOS; en n'utilisant aucune fonction générique nous aurions un style à la Java. Il est clair qu'en modulant l'utilisation des fonctions génériques et des (multi-)méthodes encapsulées nous pouvons aussi moduler les avantages de chaque style, en primant la flexibilité et la rapidité de prototypage dans un cas, ou la modularité et l'encapsulation de l'état dans l'autre, et ceci en fonction du degrée d'utilisation des méthodes encapsulées ou des fonction génériques.

Enfin cette analyse de la covariance et de la contravariance peut être utilisée pour définir des versions de compilateurs de langages qui utilisent une règle de sous-typage covariante (qui n'est pas sûre par rapport au typage), afin d'ajouter des "patches" aux programmes de ces langages pour éviter les erreurs de type, comme nous le montrons dans le prochain chapitre.

Chapitre 2

Applications aux langages Java et O_2

Articles de référence: [BC96, BC97]

Dans le chapitre précédent nous avons introduit une technique générale pour permettre la spécialisation covariante des paramètres des méthodes tout en conservant une relation de sous-typage. Dans ce chapitre nous allons appliquer et adapter cette technique au cas de deux langages commerciaux: O_2 [BDK92] et Java [GJS96].

Nous avons choisi ces langages parce qu'ils présentent chacun une difficulté majeure à l'utilisation de la technique du chapitre I.1 celle-ci étant l'utilisation de l'héritage multiple dans le cas de O_2, et la présence des méthodes surchargées dans Java.

2.1 O_2

O_2 est un langage de programmation pour bases de données fortement typé. L'aspect le plus important du système de types de O_2 est la présence d'une relation de sous-typage. Toutefois la discipline de types de O_2 n'est pas sûre, puisque des erreur de types peuvent apparaître pendant l'exécution d'un programme même si celui-ci a passé le contrôle de types statique. Le problème est du au fait que O_2 admet la spécialisation covariante des paramètres des méthodes. Mais les auteurs de O_2 ont préféré renoncer à la sûreté statique plutôt qu'à la possibilité de pouvoir (re)définir des méthodes binaires dans des sous-types.

La solution pour réconcilier sous-typage et méthodes binaires —nous le savons depuis le chapitre précédent— est celle d'introduire les multi-méthodes encapsulées. Toutefois la particularité de O_2 fait qu'une modification de la syntaxe serait inacceptable. En fait, O_2 est utilisé pour programmer des bases de données. Nous avons donc à traiter des objets (états et méthodes) persistantes. Ainsi une modification de la syntaxe ne pourrait s'appliquer qu'aux bases de données nouvelles mais le problème persisterait pour tous les objets (persistants) déjà existant.

Il faut donc chercher une solution qui puisse être appliquée aussi aux données existantes afin de pouvoir "patcher" ces dernières. La solution nous est suggérée par l'observation du programme de la page 27. Si l'on regarde la première définition de `equal` dans `ColorPoint`, c'est-à-dire celle dont le rôle est de gérer les arguments statiquement destinés à la super-classe, on notera qu'elle utilise le même code que la méthode en `2DPoint`. Ainsi à la place de répéter dans `ColorPoint` l'ancien code, nous pourrions le remplacer par un pointeur à l'ancienne méthode, ce qui communément correspond à un appel à "super". Par exemple, en empruntant la notation de O_2 pour les appels à "super" `equal` dans `ColorPoint` serait ainsi défini

```
equal(p:2DPoint) = return([self 2DPoint@equal]);
```

```
equal(p:ColorPoint) = return( (self.x == [p xVal])
                           and (self.y == [p yVal])
                           and (self.c == [p cVal]));
```

ou C@ devant un message m est une directive qui indique que la méthode associée à m est à rechercher à partir de la classe C. La signification d'un tel procédé est assez simple. Chaque fois qu'une méthode spécialisée de manière covariante est appelé on contrôle le type de son argument et on décide soit d'exécuter le code (la deuxième branche est sélectionnée), soit de continuer la recherche d'une méthode plus appropriée dans une super-classe (la première branche est sélectionnée).

Or il est clair que la méthode qui fait appel à la super-classe peut être insérée automatiquement par le compilateur sans demander aux programmeur de le faire. Il est aussi clair que cette insertion peut ne pas être effectuée au niveau du code source mais qu'elle peut être prise en compte directement dans le code objet: il suffit de marquer toute méthode spécialisée de manière covariante, et de faire en sorte qu'elle déclenche une recherche dans sa super-classe chaque fois qu'elle est appelée avec un argument de type incompatible. Une telle discipline permet de surmonter le problème avec le code déjà existant: une simple recompilation de la bases de donnée permettra de reparer le code. En plus on peut prouver que ces techniques et la discipline de typage qui vient avec sont *type-safe*.

Toutefois il reste encore un problème supplémentaire à résoudre dans O_2, celui de la gestion de l'héritage multiple. En fait la technique ci-dessus n'est définie qu'en cas d'héritage simple car dans le cas d'héritage multiple il n'existe pas une unique super-classe. Autrement dit, il n'existe plus un endroit standard d'où commencer la recherche d'une méthode pour une application mal-typée.

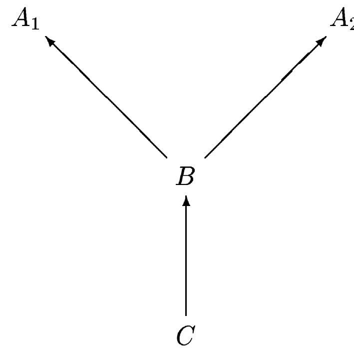
Un deuxième problème est du à l'application des multi-méthodes: dans les chapitres précédents nous avons vu que si une multi-méthode de type $\{S_1 \to T_1, \ldots, S_n \to T_n\}$ est appliquée à un argument de type U la méthode sélectionnée est celle définie pour le type $S_j = \min_{i=1..n}\{S_i \,|\, U \leq S_i\}$. Avec la présence de l'héritage multiple des conditions sont requise pour assurer que l'ensemble $\{S_i \,|\, U \leq S_i\}$ possède un plus petit élément.

Voici quelques exemples où une extension naïve de la solution pour l'héritage simple échoue. Considérons quatre classes A_1, A_2, B et C où C est définie par héritage de B qui a son tour est définie par héritage multiple de A_1 et A_2. Considérons maintenant le programme suivant où $T \leq T_1, T_2$:

```
class E
   method m(x:A1):T1;
   method n(x:B):T1
end;

class F
   method m(x:A2):T2;
   method n(x:B):T2
end;

class G inherit E, F
   method m(x:C):T;
   method n(x:C):T
end;
```



La classe G hérite de E et F, et puisque le message m est défini dans les deux super classes G évite le conflit d'héritage multiple en définissant une méthode pour m. Toutefois cette méthode est une spécialisation covariante. Ainsi pour éviter une erreur de type le compilateur pourrait ajouter des nouvelles branches qui exécutent la méthode de E pour des arguments de type A_1 et la méthode dans F pour ceux de type A_2. Mais quelle méthode devrait-il ajouter pour des arguments de type B? Il n'y a que deux choix possibles et les deux sont également bonnes. Ou, plutôt, également mauvaises car les deux peuvent engendrer une erreur de type à l'exécution: si par exemple la branche dans E

est choisie alors on pourrait avoir une méthode dont le résultat a le type statique T_2 (car appliquée à un argument de type A_2) mais qui dynamiquement a le type, incomparable, T_1 (car dynamiquement l'argument est de type B). Ainsi un choix arbitraire non seulement casserait la compréhensibilité de la sélection mais pourrait même résulter type-unsafe.

Tous les cas "pathologiques" peuvent être ramenés à ce cas particulier: une classe C a deux super-classes incomparables qui définissent une méthode pour un même message et les domaines de ces méthodes partagent un sous-type qui n'est pas géré par la redéfinition dans la classe C. Nous avons vu qu'un choix arbitraire non seulement casserait la prédictibilité et la compréhensibilité —on peut aussi dire la "naturalité"— de la sélection mais pourrait même résulter type-unsafe.

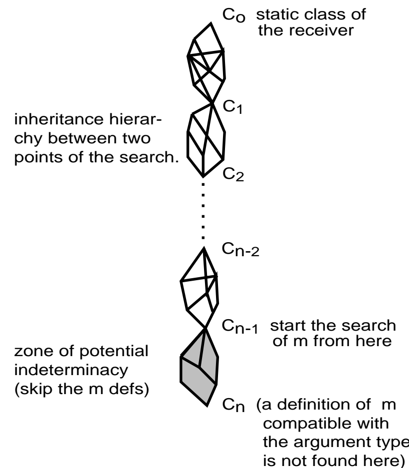
Pour éviter ces problèmes nous proposons une solution que s'inspire directement de celle pour l'héritage simple. Cette dernière fonctionne essentiellement pour deux raisons: (i) il n'y a pas des cas pathologiques où un choix arbitraire s'impose; (ii) la recherche aboutit car, à la limite, elle s'arrêtera dans la classe statique du receveur, qui doit forcément posséder une méthode capable de gérer les arguments. Notre solution pour l'héritage multiple est basée sur ces deux idées, notamment:

1. La recherche de la méthode est restreinte à la portion de la hiérarchie d'héritage inclue entre le type statique et le type dynamique du receveur
2. Si cette portion de la hiérarchie d'héritage inclut une zone où un cas pathologique apparaît, la zone en question est sautée par la recherche.

Cette solution présente deux caractéristiques extrêmement intéressantes: (i) elle peut être obtenue d'une manière totalement statique par l'ajout de la part du compilateur d'un nombre opportun de branches; (ii) dans le cas d'héritage simple on obtient la même solutions qu'auparavant.

Le détail technique et l'algorithme que le compilateur doit effectuer pour l'ajout des branches sont décrits dans [BC96]. Ici nous nous limitons a donner une intuition graphique de l'algorithme de lookup résultant.

Imaginons que le message m ait été envoyé à un objet dont le type statique est C_o mais dont le type dynamique est C_n. Le système cherche une méthode pour m dans C_n mais à cause de la spécialisation covariante la méthode pour m ne peut pas gérer l'argument actuel. Alors les méthodes ajoutées par le compilateur font continuer la recherche dans C_{n-1} qui est la *super-classe directe de jointure pour* C_o de C_n (voir figure a coté). Intuitivement la super-classe directe de jointure pour C de A est la plus petite super-classe de A et sous-classe de C qui est comparable avec toute autre classe comprise entre A et C. Et la recherche est répétée pour toute classe de jointure ($C_{n-1}, C_{n-2}, \ldots$ dans la figure) tant que la bonne méthode n'est pas trouvée.



Il faut noter que dans la zone grise entre C_n et C_{n-1} il peut y avoir une méthode capable de gérer l'argument actuel. Mais c'est précisément pour cela qu'il ne faut pas la sélectionner car cette partie de la hiérarchie est une zone de risque potentiel, où un choix arbitraire pourrait être requis. Ainsi la recherche saute cette zone (ainsi que toute autre zone dangereuse) se restreignant aux seules classes de jointure comprises entre le type statique et le type dynamique du receveur.

Cette solution maintient un certain dégré de naturalité: elle peut ne pas exécuter la méthode la plus spécifique, mais elle n'effectue jamais de choix arbitraire, ce qui la rend prédictible. Elle est aussi compréhensible, car comme dans le cas de l'héritage simple une spécialisation covariante

n'est qu'un overriding partiel de la méthode de la super-classe, ici dans le cas de l'héritage multiple il s'agit du même overriding partiel mais de la méthode qui cette fois-ci se trouve dans la super-classe jointure. En plus notre solution est fondée sur des bases théoriques solides, c'est pourquoi la correction de notre solution peut être prouvée formellement et la type-safety garantie statiquement.

2.2 Java

Dans les cas de Java les contraintes pour l'introduction de la spécialisation covariante des méthodes sont totalement opposées à celles d'O_2. En fait dans le cas de O_2 nous avons cherché une solution qui change la sémantique des programmes sans modifier leur syntaxe. Au contraire pour Java nous cherchons une extension conservative du langage dont la compilation ne demande aucune modification du bytecode. Comme dans le cas d'O_2 où nous devions prendre en compte les bases d'objets persistants existantes, ici il est impératif de considérer que l'un des points forts de Java est la grande diffusion de la JVM.

Des difficultés supplémentaires pour l'implantation de multi-méthodes dans Java sont données par la nécessité de supporter une programmation modulaire et une compilation séparée et par la présence dans le langage de mécanismes de surcharge statique. Tous ces problèmes sont pris en compte par la solution que nous proposons, qui est basée sur la définition des méthodes que nous appelons *parasites*. Les méthodes parasites sont une adaptation des multi-méthodes encapsulées (proposées au chapitre I.1), une forme restreinte de multi-méthodes qui permet une programmation modulaire et la compilation séparée.

Une méthode parasite est une méthode Java comme toutes les autres qui *en plus* étend la fonctionnalité d'autres méthodes (les méthodes *hôtes*) pour une certaine combinaison des arguments. Si une méthode hôte est appelée avec des arguments compatibles avec les types des paramètres d'un parasite, le corps du parasite est appelé à la place. Une méthode parasite en général spécialise de manière *covariante* son hôte, c'est-à-dire qu'elle gère un sous-ensemble des arguments de son hôte. Comme dans le cas de O_2 cette solution peut être considérée comme un cas de overriding partiel, c'est-à-dire que la méthode parasite overrides son hôte seulement pour les arguments qu'elle peut gérer. Pour les mêmes raisons que celles pour O_2 elle est *type-safe*. Mais une méthode parasite peut aussi spécialiser un hôte de manière *contravariante* (notez que cela n'est pas possible en Java), c'est-à-dire gérer tout les arguments qui sont passés à l'hôte. Autrement dit, l'hôte est purement et simplement overridden et son corps ne sera jamais exécuté pour un receveur instance de la nouvelle classe. Ainsi les méthodes parasites réconcilient dans un cadre commun la spécialisation covariante et celle contravariante (tandis que Java ne permet que la spécialisation invariante), sans sacrifier la type safety ou la compilation séparée.

Comme dans les cas de O_2 nous nous limitons à présenter les idées principales par des exemples. Le lecteur est invité à consulter [BC97] pour plus de détails.

Une méthode est déclarée *parasite* en utilisant le mot-cléf `parasitic`. Par exemple considérons un classe `IntList` avec une méthode *union* qui effectue l'union ensembliste entre `this` (qui en Java dénote le receveur du message) et une autre `IntList`. Plus précisément *union* produit une autre instance de `IntList` qui contient tout les éléments de `this` plus les éléments de l'argument qui ne sont pas déjà présents dans `this`. Cette méthode utilisera un algorithme $O(mn)$ (ou m et n sont les longueurs des deux listes) ou un algorithme plus sophistiqué qui d'abord ordonne les listes et ensuite effectue un merge. Plus tard nous définissons la sous-classe `IntSortedList` qui redéfini *union* pour effectuer un simple $O(m+n)$ merge. Dans ce cas il est opportun définir cette deuxième méthode comme parasite (figure 2.1) Sans cela Java utiliserait la surcharge statique. Ainsi la méthode la plus efficace ne serait utilisée que si les deux arguments de `union` n'avaient *statiquement* le type `IntSortList`. Par contre en définissant `parasitic` la méthode la plus efficace on utilise le type dynamique des arguments pour la sélection. En fait une méthode parasite s'attache à toute méthode

```
class IntList {
  public IntList union(IntList l)
          {  body for unioning two lists  }
}

class IntSortedList extends IntList {
  public parasitic IntSortedList union(IntSortedList l)
          {  body for merging two ordered lists  }
}
```

FIG. 2.1: *Parasite of an inherited host method*

moins spécifique [1] de la même classe. Dans l'exemple en figure 2.1 la nouvelle définition de `union` s'attache à la méthode `union` héritée de `IntList` et la remplace pour tout argument qu'elle peut gérer. Le parasite donc ne redéfini pas son hôte mais il lui s'attache et lui vole tout argument que le parasite peut utiliser.

Dans l'exemple ci dessus nous avons que le parasite s'attache à une méthode héritée. Toutefois il peut aussi bien s'attacher à une méthode définie dans la classe même, ce qui peut par exemple être utilisé pour obtenir une implantation encore plus efficace d'union comme montré en figure 2.2. Dans

```
class IntList {
  public IntList  union(IntList l)
            { body for unioning two lists }
}

class IntSortedList extends IntList {
  public IntSortedList  union(IntList l)
            { body for efficiently unioning two
              lists when the first is ordered }

  public parasitic IntSortedList  union(IntSortedList l)
            { body for merging two ordered lists }
}
```

FIG. 2.2: *Parasite of a local host method*

ce cas le parasite s'attache aux deux autre méthodes pour `union` car les deux sont moins spécifiques.

Enfin il est possible d'utiliser les parasites pour une spécialisation contravariante car un parasite s'attache non seulement aux méthodes moins spécifiques définies (c'est-à-dire déclarées ou héritées) dans sa classe mais aussi à toute méthode héritée plus spécifique. Dans ce cas l'hôte cède tout le contrôle à son parasite: l'hôte est "overridden".

Par exemple si nous avons décidé d'implanter dans `IntSortedList` la méthode `union` pour tout objet `IntBag` comme dans la figure 2.3 alors en déclarant la méthode comme parasite nous masquons complètement la définition de `union` dans `IntList`. Quand une instance de `IntSortedList` reçoit le message `union` elle exécute la deuxième méthode déclarée dans sa classe si l'argument est une `IntSortList`, la première sinon.

Il est aussi possible d'attacher des parasites à des méthodes statiques, pourvu que les parasites aussi soient statiques.

1. Une méthode est plus spécifique qu'une autre si elles ont le même nom et le même nombre de paramètres et le type de chaque paramètre de la première méthode est un sous-type du type du paramètre correspondant de la seconde.

```
interface IntBag {
      ⋮        // various method signatures
}

class IntList implements IntBag {
  public IntList union(IntList l) { ...  }
      ⋮        // implementation of interface's signatures
}

class IntSortedList extends IntList {
  public parasitic IntSortedList union(IntBag b)
        { super general version }

  public parasitic IntSortedList union(IntSortedList l)
        { efficient specific version }
}
```

FIG. 2.3: *Contravariant Parasite*

Pour ce qui concerne l'héritage la relation hôte-parasite est aussi héritée. Comme pour toute autre méthode, une méthode parasite est overridden quand une méthode avec le même nom et les mêmes paramètres est définie dans une sous-classe. Sinon elle est héritée. Quand une méthode parasite est overridden par une méthode non parasite, la nouvelle méthode est implicitement parasite avec les mêmes hôtes (et parasites) qu'il y avait avant, sauf que si une de celles-ci est overridden par une méthode déclarée parasite, la connexion est cassée.

En résumé une méthode déclarée parasite s'attache à toute méthode moins spécifique disponible et à toute méthode plus spécifique héritée. En plus une méthode non déclarée parasite qui overrides une méthode parasite est néanmoins parasite.

La définition de l'héritage pour les méthodes parasites est peut être l'un des choix de *design* les plus délicats de ce travail. Elle peut paraître très compliquée mais en fait elle n'obéit qu'à quelques principes assez simples. Le programmeur doit considérer que toutes les méthodes parasites du même nom et nombre de paramètres forment ensemble une unique multi-méthode. Ces méthodes inter-agissent de façon que quand l'une d'entre elles est appelée celle qui est choisie est toujours la plus spécifique pour les arguments donnés. Si maintenant nous avons une classe avec une multi-méthode et nous voulons définir une sous-classe nous pouvons vouloir hériter la multi-méthode telle quelle, ou bien la modifier. Dans ce dernier cas on peut envisager trois modifications possibles: (1) Remplacer une ou plusieurs méthodes de la multi-méthode; (2) remplacer totalement la multi-méthode par une nouvelle; (3) ajouter une nouvelle méthode à la multi-méthode. Notre solution permet cela d'une manière très simple. Ainsi pour (1) il suffit d'utiliser Java et faire un overriding par une méthode qui n'est pas déclarée parasite; pour (2) et (3) il suffit de définir des nouvelles méthodes parasites.

En conclusion le programmeur doit se rappeler que le simple overriding cause le remplacement pur et simple d'une méthode dans une multi-méthode, tandis qu'un overriding parasite peut causer une forme d'overriding partiel et donc la restructuration de la relation de parasitage.

Une fois que nous avons associé à chaque méthode l'ensemble de ses hôtes il est alors très facile de définir la discipline de typage de méthodes parasites. Elle se résume dans la règle suivante.

> *Le type du résultat d'un parasite doit être compatible ("assignment compatible") avec le type du résultat de ses hôtes.* [2]

2. Il faut ajouter à cette règle des restrictions assez naturelles sur la compatibilité des `throws` et sur les méthode statiques.

Enfin, pour terminer, lorsque plusieurs parasites sont attachés à un même hôte il faut choisir l'ordre dans lequel on va tester leur compatibilité. Nous avons choisi d'utiliser l'ordre textuel: la dernière méthode parasite qui s'applique est choisie. Il faut toutefois noter que puisque les parasites peuvent avoir à leur tour des parasites, le parasite le plus spécifique (s'il existe) est toujours choisi: si l'ordre textuel choisit un parasite moins spécifique, il sera à son tour l'hôte d'autres parasites plus spécifiques qui auront ainsi une deuxième chance.

Dans [BC97] le lecteur pourra trouver toutes les définitions formelles d'héritage, de sélection et de typage. Il trouvera aussi une ample discussion sur les différents problèmes rencontrés et sur les choix de *design* effectués ainsi que plusieurs considérations sur l'implantation des méthodes parasites. Quoique pour des raisons d'efficacité nous ayons implanté les méthodes parasites en modifiant (plus précisément, en étendant) le compilateur de Sun, il est intéressant noter que les méthodes parasites peuvent être considérées comme du sucre syntaxique car elle peuvent être traduites en Java standard par un pré-processeur. Comme exemple nous donnons dans le figures 2.4, 2.5 et 2.6 les traductions des exemples données auparavant. Cette traduction peut être effectuée de manière automatique comme décrit dans [BC97], ce qui montre que les méthodes parasites constituent une extension conservative de Java.

2.3 Conclusion

Nous avons vu deux manières complètement opposées pour ajouter les multi-méthodes dans un langage de programmation à la Simula (c'est-à-dire avec du dispatch simple) de manière type safe et en maintenant la modularité et la compilation séparée. La première est basée sur une modification sémantique du langage, la deuxième sur une extension syntaxique. Comme il était prévisible les contraintes de la deuxième solutions sont bien plus ardues à satisfaire, c'est pourquoi la solution trouvée est somme tout moins satisfaisante que la solution proposée pour O_2. Dans le cas particulier de Java la plupart des compromis que nous avons du accepter dérivent du fait que nous ne voulions pas changer la sémantique de la surcharge statique de Java. En fait une solution bien plus satisfaisante —mais non-conservative— serait de considérer toutes le méthodes de Java comme implicitement parasites, ce qui revient à obtenir la solution proposée pour O_2. Dans la pratique cette solution consisterait précisément à avoir un résolution dynamique (plutôt que statique) des méthodes surchargés en Java. De toute manière nous sommes certains que cela constituerait la meilleure solution pour un nouveau langage.

```
class IntList { ...  } // unchanged

class IntSortedList extends IntList {
  public IntList union(IntList l) {
    if (l == null ||
        l instanceof IntSortedList) {
      return IntSortedList$union((IntSortedList)l);
    }
    else return super.union(l);
  }

  public IntSortedList union(IntSortedList l) {
    return IntSortedList$union(l);
  }

  public final IntSortedList
      IntSortedList$union(IntSortedList l)
    { body for merging two ordered lists }
}
```

FIG. 2.4: *Translation of Figure 2.1 (the host is inherited).*

```
class IntList { ...  } // unchanged

class IntSortedList extends IntList {
  public IntSortedList union(IntList l) {
    if (l == null ||
        l instanceof IntSortedList) {
      return IntSortedList$union((IntSortedList)l);
    }
    else { body for efficiently unioning two
           lists when the first is ordered }
  }

  public IntSortedList union(IntSortedList l) {
    return IntSortedList$union(l);
  }

  public final IntSortedList
      IntSortedList$union(IntSortedList l)
  { body for merging two ordered lists }
}
```

FIG. 2.5: *Translation of Figure 2.2 (the host is local).*

```
interface IntBag { ... } // unchanged

class IntList implements IntBag { ... } // unchanged

class IntSortedList extends IntList {
  public IntList union(IntList l) {
    if (l == null ||
        l instanceof IntSortedList) {
      return IntSortedList$union((IntSortedList)l);
    } else {
      return IntSortedList$union((IntBag)l);
    }
  }

  public IntSortedList union(IntBag b)
  { return IntSortedList$union(b); }
  public final IntSortedList
      IntSortedList$union(IntBag b)
  { super general version }

  public IntSortedList union(IntSortedList l)
  { return IntSortedList$union(l); }
  public final IntSortedList
      IntSortedList$union(IntSortedList l)
  { efficient specific version }
}
```

FIG. 2.6: *Translation of Figure 2.3 (contravariant parasite).*

Deuxième partie

Généralisation de la surcharge: modules et démonstration automatique

Nous avons souligné que l'avantage de la surcharge avec liaison tardive est qu'elle permet une programmation de type incrémental et la réutilisation de code, d'où sa liaison avec les paradigmes à objets. Toutefois ces avantages ne doivent pas être cantonnés aux paradigmes à objets, c'est pourquoi nous étudions leur intégration dans d'autres paradigmes tels que les systèmes de modules (chapitre II.1) et la démonstration automatique (chapitre II.2).

Chapitre 1

Systèmes de modules

Articles de référence: [AC96b]

1.1 Introduction

Les systèmes de modules à la SML [MQ85, MTH90] sont très généraux et puissants. Ils permettent un découpage modulaire des programmes et la définition de transformations (appelées *foncteurs*) sur les modules mêmes (appelés *structures*). Toutefois, ils n'autorisent qu'une forme très limitée de réutilisation de code et de programmation incrémentale. Plus précisément, les modules à la SML ne possèdent pas les caractéristiques d'héritage et de spécialisation de code qui ont fait le succès des langages à objets. Ce manque est pénalisant lorsqu'on veut faire évoluer un programme.

Ainsi, si à un moment de la vie du programme on décide d'ajouter des nouvelles fonctionnalités en déclarant des nouvelles structures qui spécialisent celles déjà existantes, l'utilisation pour ces structures de foncteurs du programme initial (avant spécialisation) reste valide. En revanche il est impossible d'affiner le comportement de ces foncteurs par la prise en compte des fonctionnalités ajoutées. Ainsi, les nouvelles structures, soit utilisent les foncteurs définis pour les anciennes, soit en définissent des nouveaux qui leur sont propres et qui n'auront aucune corrélation avec les anciens. Dans une telle situation la capacité d'évolution de programmes est sérieusement limitée, et leur durée de vie et facilité de maintenance sont diminuées.

Nous nous proposons de pallier ces limites en définissant une extension des systèmes de modules à la SML par l'ajout de la *surcharge* et de la *liaison tardive*. Cette extension sera extrêmement élémentaire d'un point de vue syntaxique. En fait, le programmeur devra apprendre l'utilisation d'une seule nouvelle commande : `extend`. En contrepartie de cet effort, il aura l'avantage de pouvoir suivre, au cours du développement de l'application, une approche semblable à celle permise par certains langages orientés objets, plus précisément par les langages qui font appel aux fonctions génériques, tels que CLOS [DG87].

Dans les chapitres précédents l'intégration de la surcharge et de la liaison tardive dans les langages fonctionnels est étudiée dans le but d'obtenir un modèle typé des langages objets et d'améliorer ceux-ci. Dans ce chapitre, nous montrons comment intégrer ces mécanismes dans un langage de modules à la SML: le langage des *modules manifestes* proposés par Xavier Leroy dans [Ler94]. Par sa syntaxe et son pouvoir expressif, ce langage est très proche des modules de SML. En revanche, il possède une sémantique plus simple que les modules de SML, ce qui en fait un cadre d'études plus agréable et mieux adapté aux extensions que nous proposons dans ce travail.

Le chapitre est organisé de la manière suivante. Dans la section 1.2 nous présentons le système de modules de SML et les modules manifestes. En section 1.3 nous illustrons par des exemples l'utilisation de la surcharge et la liaison tardive dans les systèmes de modules, ainsi que les avantages induits par l'ajout de ces mécanismes dans le langage : une plus grande réutilisation de code et la programmation incrémentale. La définition formelle de notre système est donnée en section 1.4, où nous décrivons syntaxe, sémantique statique et sémantique dynamique. Dans la section 1.5 nous discutons du problème de choix du code pour un foncteur surchargé.

1.2 Le langage de modules

Le découpage modulaire des programmes est fondamental pour le développement de programmes de grande taille.

Une discipline modulaire très générale est définie par le système de modules de SML, où les modules sont des objets d'un sous-langage typé. La construction de base de ce langage est le module (*structure*), formé de séquences de déclarations de types, valeurs et autres modules. Les types des modules (*signatures*), sont des séquences de spécifications de types pour les composantes d'un module. Enfin, il est possible de construire un module à partir d'autres modules par le biais de fonctions de modules vers les modules, appelées *foncteurs*. De cette manière, la combinaison des modules a lieu par l'application de foncteurs.

Dans ce travail nous utilisons les *modules manifestes*, une variante des modules de SML, proposée par Xavier Leroy dans [Ler94] avec une sémantique plus simple et presque le même pouvoir expressif que les modules de SML.

Montrons quelques exemples de structures, signatures et foncteurs dans les modules de SML. Les signatures ci-dessous décrivent des structures d'arbre et de dictionnaire.

```
signature Item =
sig
  type item;
  val isequal: item*item -> bool
end

signature Tree =
sig  structure i: Item;
  type 'a tree;
  val empty: 'a tree;
  val isnull: 'a tree -> bool
  val cons: 'a * 'a tree * 'a tree -> 'a tree
  val root: 'a tree -> 'a
  val left: 'a tree -> 'a tree
  val right: 'a tree -> 'a tree
end


signature Dict =
 sig
  type key
  type 'a dict
  val empty : 'a dict
  val isnull: 'a dict -> bool
  val find: key * 'a dict -> 'a
  val insert : key * 'a * 'a dict -> 'a dict
 end
```

La structure suivante est de signature `Item`:

```
structure IntItem : Item =
 struct
  type item = int
  fun isequal (a,b) = (a = b)
 end
```

On peut construire un dictionnaire à partir de structures d'arbres. Le foncteur `mkDict` suivant construit un dictionnaire.

```
functor mkDict(t: Tree): Dict =
  struct
    type key = t.i.item;
    type 'a dict = (key * 'a) t.tree
    val empty = t.empty;
    val isnull = t.isnull
    fun find (k,d) = if isnull(d) then raise Notfound
                   else let (k',a) = t.root(d) in
                        if t.i.isequal(k,k') then a ...
    fun insert (k,a,d) = ...
 end
```

1.2.1 Sous-typage

Le typage de SML autorise une forme de sous-typage structurel avec la notion de *filtrage de signatures* (en anglais, *signature matching*). Une structure s *est filtrée* par une signature S, si s possède au moins tous les composants spécifiés par S, et si chaque type spécifié dans un composant de S filtre, ou est compatible avec (c'est-à-dire, est moins polymorphe que) le composant de même nom dans s. Ci-dessous nous déclarons la signature `OrdItem` spécifiant les objets avec un ordre arbitraire et la signature `OrdTree` des arbres de recherche. La structure `IntOrd` suivante est filtrée non seulement par la signature `OrdItem` mais aussi par la signature `Item` des objets sans ordre.

```
signature OrdItem =
 sig
  type item
  val isequal: item * item -> bool
  val isless: item * item -> bool
 end

signature OrdTree =
 sig
  structure i: OrdItem
  type 'a tree
  val empty: 'a tree
  val isnull: 'a tree -> bool
  val insert: 'a * 'a tree -> 'a tree
  val remove: 'a * 'a tree -> 'a tree
  val root: 'a tree -> i.item
  val left: 'a tree -> 'a tree
  val right: 'a tree -> 'a tree
  val max: 'a tree -> 'a
  val min: 'a tree -> 'a
 end
```

```
structure IntOrd =
 struct
  type item = int
  val isequal = op =
  val isless = op <
 end
```

La relation de filtrage induit une relation d'ordre sur les structures. Nous dirons qu'une signature S est un *sous-type* de la signature S' si toute structure filtrée par S est aussi filtrée par S'. L'ensemble des signatures qui filtrent une expression possède un plus petit élément (par rapport à l'ordre de sous-typage). Ce plus petit élément est le *type* de l'expression (on dira aussi que chaque expression est *filtrée exactement* par son type, afin de différencier le type d'une structure des autres signatures que la filtrent). Par exemple, toute structure qui est filtrée (exactement) par la signature `OrdTree` est aussi filtrée par la signature `Tree`.

Par ailleurs, la règle de typage pour l'application de foncteurs impose le filtrage de l'argument par la signature du paramètre formel du foncteur. Ainsi, il est possible d'appliquer le foncteur `mkDict` sur une structure qui est filtrée par `OrdTree`, c'est-à-dire dont le type est un sous-type de la signature du paramètre formel de `mkDict`.

1.2.2 Modules manifestes

La sémantique des modules à la SML est très complexe, particulièrement en ce qui concerne les foncteurs d'ordre supérieur et la spécification du partage des types. Dans ce travail, nous utilisons les *modules manifestes* qui ont les mêmes constructions que les modules de SML, mais les contraintes de types dans les signatures, telles que les spécifications de partage, de transparence ou d'abstraction sur les types, sont spécifiées par l'unique moyen d'annotations explicites sur les types : les annotations *manifestes*. Il s'agit d'annotations sur les constructeurs de types de la forme `type t =` σ, qui à la fois, spécifient un nouveau constructeur de type `t`, et imposent le type σ en tant que représentation effective pour `t`. L'absence d'annotations sur un constructeur de type spécifié (notée `type t`) rend ce constructeur abstrait dans la suite du typage. Dans l'exemple suivant la structure

```
structure intOrder =
 struct
  type t = int
  fun cmp i1 i2 = i1 < i2
 end
```

a la signature

```
intOrder : sig
             type t = int
             fun cmp : t -> t -> bool
           end
```

où les annotations manifestes spécifient la compatibilité entre les types `intOrder.t` et `int`. Dans la signature suivante, la spécification du type `t` est abstraite :

```
signature Intlist =
 sig
  type t
  val nil : t
  val cons : int -> t -> t
 end
```

1.3 Modules avec surcharge et liaison tardive

Dans cette partie, nous montrons par quelques exemples les avantages d'utiliser un langage de modules manifestes qui incorpore aux foncteurs un mécanisme de surcharge avec liaison tardive.

1.3.1 Surcharge et programmation modulaire

Le foncteur `Use` ci-dessous utilise un dictionnaire qu'il construit à partir d'une structure d'arbre passée en paramètre. Les éléments du dictionnaire sont extraits d'une file de priorité qui est également un paramètre du foncteur [1].

```
functor Use (p: PQueue, t: Tree with t.i.item =  p.item) =
 struct
   structure dict = mkDict(t)
   ...
   fun solve(q,d) =  val fkey = p.front q
                     val el = dict.find(fkey,d) in ...
   ...
 end
```

Cette première version de `Use` construit le dictionnaire à partir d'arbres sans aucun ordre, ce qui rend inefficace l'opération de recherche d'un élément. Le foncteur `mkODict` construit un dictionnaire à partir d'une structure d'arbre de recherche. L'opération de recherche qui en résulte est optimale dans le cas des arbres qui restent équilibrés.

```
functor mkODict(t:OrdTree): Dict =
 struct
  type key = t.i.item;  type 'a dict = (key * 'a) t.tree
  val empty = t.empty;  val isnull = t.isnull
  fun find (k,d) = if isnull(d) then raise Notfound
     else let (k',a) = t.root(d) in
            if t.i.isequal(k,k') then a
            else if t.i.isless(k,k') then find (k,left(d))
            else find (k,right(d))
  fun insert (k,a,d) = if isnull(d) then t.cons((k,a),empty,empty)
                       else (* Ordered search of a free position *)
 end
```

Tel qu'il est définit, le foncteur `Use`, ne construit que des dictionnaires inefficaces, même si un arbre de recherche lui est passé en argument (ce qui est possible grâce au sous-typage). Si nous voulons que `Use` construise des dictionnaires exploitant la structure d'arbre de recherche, la seule solution est de définir un nouveau foncteur `OUse` identique à `Use`, sauf pour la construction du dictionnaire qui se fera alors par un appel à `mkODict`. Nous observons en particulier, qu'il n'est pas possible d'avoir le foncteur de construction de dictionnaires en argument de `Use` (ce qui permettrait à l'utilisateur de choisir la manière de construire les dictionnaires). En fait, à cause de la contravariance pour le type du paramètre dans la règle de sous-typage des foncteurs (voir par exemple [Ler94]), il est impossible d'obtenir un foncteur `Use` correctement typé et avec le comportement souhaité. Ainsi, la seule solution qui permet d'avoir deux manières de construire les dictionnaires, entraîne d'une part une duplication de code, et d'autre part laisse à la charge du programmeur la tâche de choisir l'utilisation de l'une ou l'autre des deux versions de `Use`.

Une solution à ce problème est de surcharger le foncteur `mkDict` utilisé dans le corps de `Use` de manière à ajouter à sa première définition une définition correspondant à la sémantique de `mkODict`. La construction

```
extend functor mkDict with mkODict
```

1. Nous ne donnons pas la définition de la signature `PQueue`.

ajoute la définition de `mkODict` à l'ensemble des définitions associées au foncteur `mkDict`, dès lors surchargé avec deux branches : l'une étant la première définition donnée pour `mkDict`, l'autre correspondant au code de `mkODict`.[2]

Plus généralement, si `f` est un foncteur (éventuellement surchargé) et `g` un foncteur non surchargé, alors après la directive `extend functor f with g` la variable `f` dénote partout dans le programme le foncteur surchargé formé de toutes les branches qui composaient `f` plus une branche définie par `g`.

Pour la sélection des branches des foncteurs surchargés, nous proposons d'utiliser une discipline de liaison tardive. Ainsi, lorsque `mkDict` (ou bien `Use`) est appliqué à une structure d'arbre sans ordre, la branche qui correspond à la première version de `mkDict` est sélectionnée. En revanche, si `mkDict` (ou `Use`) est appelé avec un arbre de recherche en argument, la branche correspondant à `mkODict` est alors utilisée, et le dictionnaire construit est plus efficace dans la recherche.

Cet exemple reflète fidèlement la situation illustrée par l'exemple décrit dans l'introduction : `Use` correspond à M, `Tree` correspond à A, `OrdTree` est B, `mkDict` correspond à P_A, `mkODict` est P_B et `mkDict` dans la définition de `Use` est P. Toutes les lignes de code qui, dans la définition de `Use`, entourent l'appel à `mkDict` correspondent au contexte $\mathscr{C}[\;]$ qui, grâce à la surcharge, n'a pas besoin d'être dupliqué.

1.3.2 Surcharge et partage de types

Dans les modules manifestes, le partage entre types est spécifié à l'aide d'annotations manifestes sur les types. Par exemple, dans le foncteur

```
functor f(structure s1: sig type t;..end
          structure s2: sig type t=s1.t;.. end)
```

la signature du deuxième argument contient l'annotation manifeste `type t= s1.t` sur le type `t`, qui peut être satisfaite seulement si deux paramètres actuels `a` et `b` de `f` ont les mêmes types `a.t` et `b.t`. Les annotations de types manifestes dans les signatures sont comparées avec les types des arguments par sous-typage. Par ailleurs, le mécanisme de sélection de code parmi les branches d'un foncteur surchargé utilise le sous-typage tel qu'il est défini dans [Ler94]. Ainsi, le partage entre types pour des foncteurs surchargés ne nécessite aucun traitement spécial.

1.4 Un calcul de modules avec surcharge

Nous décrivons dans cette partie un calcul sur les modules manifestes avec foncteurs surchargés. Le calcul de modules manifestes sans surcharge est celui de [Ler94].

1.4.1 Syntaxe

Nous supposons que la définition d'un langage de base, sous-jacent aux modules, ait été préalablement donnée.

Dans la grammaire suivante nous notons τ les expressions de type du langage de base, p les chemins d'accès pour les composants des structures, t les identificateurs de type et x les identificateurs de structures. Les expressions de structure (notés par s) sont soit des chemins d'accès, soit des définitions de structure, soit des applications de foncteur, soit encore des structures restreintes par une

2. Il faut noter que pour obtenir l'effet désiré il faut abandonner le monde purement fonctionnel, car la modification de `mkDict` doit être globale. En fait, même si la déclaration

```
extend functor mkDict with mkODict
```

suit celle de `Use`, elle doit néanmoins modifier la sémantique de `mkDict` utilisé à l'intérieur de `Use`. Dans cette étude théorique nous n'étudions pas cet aspect, même si dans l'implantation réaliste du système il doit être inclus.

signature. Une structure est une collection de déclarations de types et de sous-structures; nous ne considérons pas le cas de valeurs déclarées, car cela concerne davantage la sémantique du langage de base.

Nous limitons ce travail au cas de foncteurs unaires, l'extension aux foncteurs à plusieurs arguments étant triviale [3], et nous laissons l'étude de foncteurs à l'ordre supérieur pour un travail à venir. Les signatures (ou *types de modules*) sont soit des *signatures de structure* (notées S), soit des *signatures de foncteur* (notées F). Une signature de structure est une collection de spécifications de sous-structures et de spécifications de types, ces dernières pouvant être abstraites (de la forme `type t`) ou manifestes (de la forme `type t` $= \tau$). Une signature de foncteur est soit un type dépendant (c'est-à-dire, un type où la variable paramètre du foncteur peut apparaître dans le type du résultat), soit une union de types dépendants (signature de foncteur surchargé).

Les foncteurs surchargés sont construits à partir de foncteurs *simples* (non surchargés) en ajoutant de nouvelles branches à leur définition par l'utilisation de la construction `and`: $f_1 f_2 \ldots /\texttt{and}/ f_n$ est le foncteur surchargé avec les branches $f_1, f_2, \ldots, f_n$. La construction `extend` que nous avons présenté dans l'introduction, est en réalité du sucre syntaxique pour les foncteurs surchargés construits avec `and`. Ainsi, pour cette étude formelle nous nous limitons au cas purement fonctionnel, dans lequel, l'utilisation de `extend functor f` n'a aucun effet sur les occurrences de `f` précédentes.

Chemins d'accès :

$p ::= x$	identificateur de structure
$\mid p.x$	accès à un composant de structure

Expressions de structure :

$s ::= p$	chemin d'accès
$\mid \texttt{struct}\ d\ \texttt{end}$	construction de structure
$\mid f\ (s)$	application de foncteur
$\mid s : S$	restriction par signature

Corps de structure :

$d ::= \epsilon \mid b; d$

Composants de structure :

$b ::= \texttt{type}\ t = \tau$	déclaration de type
$\mid \texttt{structure}\ x = s$	déclaration de structure

Expressions de foncteur :

$f ::= \texttt{functor}\ (x : S) s$	foncteur simple
$\mid f\ \texttt{and}\ (x : S) s$	foncteur surchargé

Expressions de type:

$\tau ::= t$	identificateur de type
$\mid p.t$	accès à un composant de type
$\mid \ldots$	dépendant du langage de base

Signature de structure :

$S ::= \texttt{sig}\ D\ \texttt{end}$

Corps de signature :

$D ::= \epsilon \mid B; D$

Composants de signature :

$B ::= \texttt{type}\ t$	spécification abstraite de type
$\mid \texttt{type}\ t = \tau$	spécification manifeste de type
$\mid \texttt{structure}\ x : S$	spécification de structure

3. Les foncteurs à plusieurs arguments sont obtenus en ajoutant au calcul les produits cartésiens de signatures.

Signature de foncteur :

$F ::= \texttt{functor}\ (x : S)S'$	signature de foncteur dépendant
$\mid F \cup \texttt{functor}\ (x : S)S'$	signature de foncteur surchargé

Une signature de foncteur surchargé est de la forme

$$\texttt{functor}(x_1 \colon S_1)S'_1 \ \cup\ \texttt{functor}(x_2 \colon S_2)S'_2 \ \cup\ \ldots \cup\ \texttt{functor}(x_n \colon S_n)S'_n$$

que nous abrégeons par $\bigcup_{i=1}^{n} \texttt{functor}(x_i \colon S_i)S'_i$. De manière intuitive, cette signature est utilisée pour typer un foncteur surchargé formé par n branches, la i-ème branche étant un foncteur simple de signature $\texttt{functor}(x_i \colon S_i)S'_i$. Lorsqu'un foncteur surchargé avec la signature ci-dessus est appliqué à une structure s, si s est filtrée exactement par S_j, alors s est passée à la j-ème branche du foncteur. Si aucune des signatures de paramètre du foncteur ne filtre exactement l'argument s, alors on choisit parmi les branches dont la signature de paramètre filtre s, la plus petite dans l'ordre du sous-typage (ceci est établi formellement par les règles (1.5) et (1.6) de la sémantique dynamique en section 1.4.3). Ainsi, par exemple, si nous appliquons le foncteur surchargé `mkDict` défini en section 1.3 à un argument qui est filtré par les signatures `Tree` et `OrdTree`, la branche définie pour `OrdTree` est choisie.

L'expression $s \colon S$ contraint la structure s à être exactement de signature S, pourvu que S filtre s; ainsi, l'expression $s \colon S$, joue le même rôle que les *coercions explicites* dans les langages fonctionnels avec sous-typage. Les coercions explicites sont importantes dans les systèmes, comme le nôtre, où l'exécution d'une expression est guidée par les signatures. Dans ce contexte les coercions explicites deviennent un moyen utile de contrôle. En fait, les restrictions de signature peuvent être utilisées pour forcer le choix d'une branche particulière dans une application de foncteur surchargé. Dans l'exemple présenté en section 1.3, si `OT` est une structure filtrée par `OrdTree`, lors de l'appel `mkDict(OT)` on choisira la branche du foncteur `mkDict` définie pour les arbres de recherche. Néanmoins, on peut forcer la sélection de la branche correspondant aux arbres génériques par l'expression `mkDict(OT:Tree)`. De manière similaire l'application `Use(P,(OT:Tree))` contraint `Use` à l'utilisation d'un dictionnaire générique, même si l'arbre passé en argument est un arbre de recherche.

1.4.2 Sémantique statique

Nous utilisons E pour noter les environnements de types, $BV(S)$ pour noter l'ensemble de variables liées par les déclaration dans S, et $S\{x \leftarrow s\}$ (respectivement, $s'\{x \leftarrow s\}$) pour noter le type (respectivement, l'expression) obtenu(e) en remplaçant x par s dans S (respectivement, dans s').

Le jugement $E \vdash S_1 <: S_2$ établit que la signature S_1 est un *sous-type* de la signature S_2 dans l'environnement E. Le jugement $E \vdash \tau \approx \tau'$ établit que le type τ est *équivalent* au type τ' dans l'environnement E.

Nous utilisons la relation de sous-typage définie dans [Ler94], sauf les règles concernant le sous-typage de foncteurs, puisque nous ne considérons pas le cas de foncteurs d'ordre supérieur.

Signatures bien formées

Dans la section précédente nous avons défini la relation de sous-typage pour les signatures. Maintenant nous utilisons cette relation pour sélectionner parmi les signatures de foncteur surchargé, celles qui sont *bien formées*. La définition de bonne formation est la suivante

Soit S la signature de foncteur surchargé $\bigcup_{i=1}^{n} \texttt{functor}(x_i \colon S_i)S'_i$. La signature S est *bien formée* dans l'environnement E (noté $WF_E(S)$) si et seulement si

$$\forall i, j \in [1..n] \qquad (E \vdash S_i <: S_j \quad \Longrightarrow \quad E; \texttt{structure}\ x : S_i \vdash S'_i <: S'_j) \tag{1.1}$$

La condition ci-dessus, appelée *condition de covariance*, n'est rien d'autre que la contrepartie pour les modules de la condition (2.4) introduite à page 16 pour $\lambda\&$ et comme cette dernière elle est nécessaire pour assurer que la signature obtenue après l'exécution d'une expression soit un sous-type de celle déterminée statiquement pour ladite expression.

Typage

Les règles de typage pour les expressions sont montrées dans la figure 1.1 (page suivante). Elles utilisent des jugements de la forme $E \vdash s\colon S$ établissant qu'une structure s *possède le type* (ou *est filtrée exactement par*) S dans l'environnement E.

$$E_1; \texttt{structure}\, x : S; E_2 \vdash x : S$$

$$\frac{E \vdash p : \texttt{sig}\, D_1; \texttt{structure}\, x : S; D_2\, \texttt{end}}{E \vdash p.x : S\{n \leftarrow p.n \mid n \in BV(D_1)\}}$$

$$\frac{E;\ \texttt{structure}\, x : S \vdash s : S'}{E \vdash \texttt{functor}(x : S)s : \texttt{functor}(x : S)S'} \quad x \notin BV(E)$$

$$\frac{E \vdash f : \texttt{functor}(x : S')S \qquad E \vdash s : S'' \qquad E \vdash S'' <: S'}{E \vdash f(s) : S\{x \leftarrow s\}}$$

$$\frac{E \vdash f : \bigcup_{i=1}^{n-1} \texttt{functor}(x : S'_i)S_i \qquad E;\ \texttt{structure}\, x : S'_n \vdash s : S_n}{E \vdash f\ \texttt{and}\ (x\colon S_n)s : \bigcup_{i=1}^{n} \texttt{functor}(x\colon S'_i)S_i} \ (*)$$

$$\frac{E \vdash f : \bigcup_{i=1}^{n} \texttt{functor}(x\colon S_i)S'_i \qquad E \vdash s : S}{E \vdash f(s) : S'_j\{x \leftarrow s\}} \quad (**)$$

$$\frac{E \vdash p : S}{E \vdash p : S/p}$$

$$\frac{E \vdash s : S' \qquad E \vdash S' <: S}{E \vdash (s : S) : S}$$

$$\frac{E \vdash d : D}{E \vdash \texttt{struct}\, d\, \texttt{end} :\ \texttt{sig}\, D\, \texttt{end}}$$

$$\frac{E;\ \texttt{type}\ t = \tau \vdash d : D}{E \vdash (\texttt{type}\ t = \tau; d) : (\texttt{type}\ t = \tau; D)} \quad t \notin BV(E)$$

$$\frac{E \vdash s : S \qquad E;\ \texttt{structure}\, x : S \vdash d : D}{E \vdash (\texttt{structure}\, x = s; d) : (\texttt{structure}\, x : S; D)} \quad x \notin BV(E)$$

(*) $x \notin BV(E)$ et $WF_E(\bigcup_{i=1}^{n} \texttt{functor}(x\colon S'_i)S_i)$

(**) $S_j = \min_{i=1..n}\{S_i \mid E \vdash S <: S_i\}$

FIG. 1.1: *Règles de typage*

La cinquième et la sixième règle méritent une attention particulière car elles traitent des cas de construction et d'application de foncteurs surchargés. Le type de la concaténation d'une nouvelle branche à un foncteur surchargé f est obtenu par l'ajout du type de la nouvelle branche au type de f, après avoir vérifié que ce type étendu est bien formé. Lorsqu'un foncteur surchargé f est appliqué à un argument de signature S, le système de types sélectionne parmi les branches dont la signature du paramètre filtre l'argument (c'est-à-dire, telles que $E \vdash S <: S_i$), la branche dont le type S_j du paramètre est le plus proche du type de l'argument (c'est-à-dire, telle que $S_j = \min_{i \in I}\{S_i \mid E \vdash S <: S_i\}$) ; le type de l'application est le type du résultat de cette branche où le paramètre formel a été remplacé par le paramètre actuel.

Il faut aussi remarquer que, contrairement à [Ler94], nous n'utilisons pas de règle de *subsumption* dans le typage, mais employons plutôt la version algorithmique des règles (où le sous-typage est utilisé directement dans les règle d'élimination). Ceci a comme avantage que toute expression typable a une signature unique, notamment celle qui la filtre exactement. Cette propriété permet l'utilisation de ce système de règles pour la sélection dynamique des branches, tandis qu'avec la subsumption, du moment que chaque argument pourrait posséder plusieurs signatures, cette sélection ne serait pas déterministe.

Les règles de typage utilisent une opération de *renforcement* [Ler94] notée S/p qui enrichit la structure du type S avec de l'information sur le chemin complet p identifiant les composantes abstraites de S. Cette opération est définie de la manière suivante :

$$\begin{array}{rcll}
(\texttt{sig}\ D\ \texttt{end})/p & = & \texttt{sig}\ D/p\ \texttt{end} & \\
(\texttt{type}\ t; D)/p & = & \texttt{type}\ t = p.t; D/p & \\
(\texttt{structure}\ x : S; D)/p & = & \texttt{structure}\ x : S/p; D/p & \\
S/p & = & S & \text{sinon}
\end{array}$$

1.4.3 Sémantique dynamique

Dans cette section nous définissons la manière dont les expressions sont “calculées” au moment de l'exécution. En particulier nous décrivons comment la liaison tardive des foncteurs surchargés est obtenue. À cette fin nous définissons une sémantique opérationnelle avec une stratégie d'appel par valeur, pour les structures closes. Définissons d'abord ce qu'est une *valeur de structure* (ou, plus simplement, une *valeur*):

Valeurs de structure :

$$\begin{array}{rl}
v ::= & \texttt{struct}\ u\ \texttt{end} \\
\mid & (v : S)
\end{array}$$

Composants valeurs:

$$\begin{array}{rl}
u ::= & \varepsilon \\
\mid & \texttt{type}\ t = \tau; u \\
\mid & \texttt{structure}\ x = v; u
\end{array}$$

Une valeur est soit une structure dont les sous-structures sont des valeurs, soit la restriction d'une valeur.

La sémantique opérationnelle est définie par les règles de réécriture suivantes :

[Beta]

$$(\texttt{functor}(x : S)s)(v) \quad \rightarrow \quad s\{x \leftarrow v\} \tag{1.2}$$

[Restriction]

$$(v : S).x \quad \rightarrow \quad v.x \tag{1.3}$$

[Access]

$$(\texttt{struct}\ u_1; \texttt{structure}\ x = v; u_2\ \texttt{end}).x \quad \rightarrow \quad v \tag{1.4}$$

[Overload]

Soit $(f\ \texttt{and}\ (x : S_n)s) : \bigcup_{i=1}^{n} \texttt{functor}(x : S_i)S'_i$ et $v : S$.

- Si $S_n = \min_{i=1..n}\{S_i \mid S <: S_i\}$ alors

$$(f\ \texttt{and}\ (x : S_n)s)(v) \quad \rightarrow \quad s\{x \leftarrow v\} \tag{1.5}$$

- Si $S_n \neq \min_{i=1..n}\{S_i \mid S <: S_i\}$ alors

$$(f\ \texttt{and}\ (x : S_n)s)(v) \quad \rightarrow \quad f(v) \tag{1.6}$$

[Context]

Si $s \rightarrow s'$ alors

$$s.x \quad \rightarrow \quad s'.x \tag{1.7}$$

$$f(s) \quad \rightarrow \quad f(s') \tag{1.8}$$

$$(s : S) \quad \rightarrow \quad (s' : S) \tag{1.9}$$

$$\texttt{struct}\ u; \texttt{structure}\ x = s; d\ \texttt{end} \quad \rightarrow \quad \texttt{struct}\ u; \texttt{structure}\ x = s'; d\ \texttt{end} \tag{1.10}$$

Commentons les règles ci-dessus en détail. D'abord il faut noter qu'elles décrivent une sémantique opérationnelle déterministe. La règle (1.2) est la règle habituelle qui décrit l'appel par valeur pour les foncteurs. La règle (1.3) efface la restriction portant sur une structure lors de l'accès à l'un de ses composants. La règle (1.4) effectue la sélection d'un composant de structure. Les règles (1.7), (1.8), (1.9) et (1.10) décrivent les réductions à l'intérieur d'un contexte.

Les règles importantes sont les règles (1.5) et (1.6) qui effectuent la sélection par liaison tardive lors de l'application de foncteurs surchargés. Lorsqu'un foncteur surchargé est appliqué, la réduction est déclenchée seulement si son argument est une valeur (cette contrainte assure la réalisation de la liaison tardive). La première chose à faire est de considérer les types du foncteur surchargé et de son argument. Nous remarquons que la déduction des types ne nécessite aucun environnement puisque les deux expressions sont closes. Dans l'ensemble des signatures paramètres du foncteur surchargé nous sélectionnons celles qui filtrent l'argument, c'est-à-dire les S_i tels que $S <: S_i$. Si l'ensemble de ces signatures a un plus petit élément, alors on peut déclencher la réduction : si la plus petite signature est égale à la signature de paramètre de la dernière branche, alors cette branche est sélectionnée [règle (1.5)] ; sinon la recherche est poursuivie sur les branches restantes [règle (1.6)].

Un point intéressant de la règle (1.5) est qu'elle rend possible la redéfinition d'une branche d'un foncteur surchargé. En fait, rien n'empêche dans un unique foncteur surchargé d'avoir plusieurs branches définies pour la même signature. Cependant, en vertu de la règle (1.5), la plus à droite parmi elles sera toujours sélectionnée. En pratique, si un foncteur surchargé f possède une branche définie pour la signature S et si nous voulons redéfinir cette branche, il n'est pas nécessaire de réécrire f ; il suffit de concaténer au foncteur la nouvelle définition de la branche. Ceci est d'autant plus intéressant du fait que la définition de la nouvelle branche peut se trouver dans un module différent de celui contenant la définition de f ; dans ce cas ce dernier module ne nécessite pas une recompilation. Avec le sucre syntaxique de la section 1.3 si `g` dénote un foncteur de signature $\texttt{functor}(x : S)S'$ alors

```
extend functor f with g
```

ajoute au foncteur dénoté par `f` une nouvelle branche pour les arguments de signature S ou, si une telle branche existait déjà dans `f`, il la redéfinit. Remarquez qu'en cas de redéfinition le système de types (plus précisément, la condition de covariance) contraint le type du résultat de la nouvelle branche à être le même que celui de l'ancienne.

1.5 Questions pendantes

Bien sûr, ce système hérite des inconvénients des modules manifestes. En particulier, le système de types de [Ler94] (et, par conséquent, notre système) ne satisfait pas la propriété de subject reduction (car des termes intermédiaires de la réduction ne sont pas typables). Mais tandis que pour [Ler94] il est possible d'établir la correction du typage par voie sémantique, il nous parait plus improbable de montrer la correction de notre système par cette voie: les seuls modèles dénotationnels pour la notion de surcharge utilisée dans ce travail ne peuvent gérer que la liaison précoce [Tsu92, CGL93, Tsu95]. L'idée d'utiliser le travail présenté dans ce chapitre pour étendre le système de modules décrit dans [Cou97b, Cou97a] nous parait bien plus prometteuse et facile, ce qui n'a pas encore été fait par manque de temps.

Bien qu'importante, la preuve de correction du système de types n'est pas parmi nos objectifs prioritaires. Une telle preuve manque aussi pour d'autres propositions de systèmes de types pour les modules (par exemple [HL94]). Nous sommes plus intéressés dans l'immédiat par l'élaboration de techniques qui assurent que l'exécution de toute structure close termine par une valeur (c'est-à-dire, soit par une valeur de structure soit par la restriction d'une valeur ; cf. la définition en section 1.4.3). Actuellement cette propriété n'est pas vérifiée à cause de la règle **[Overload]**. En fait, nous remarquons que pour être appliquée cette règle nécessite l'existence d'une branche plus spécialisée parmi celles qui filtrent l'argument. Mais si une telle branche n'existe pas, c'est-à-dire si l'ensemble $\{S_i \mid S <: S_i\}$ ne possède pas de plus petit élément, l'exécution est bloquée et rend comme résultat une application, ce qui n'est pas une valeur. La raison réside dans le fait que tandis que nous avons utilisé la règle de covariance pour la bonne formation des types, nous n'avons pas utilisé pour la même une règle correspondant à la condition (2.5) à page 16. Dans [AC96b] nous expliquons pourquoi une telle condition ne serait pas satisfaisante pour un système de modules, et nous explorons de manière approfondie plusieurs autres solutions. Parmi celles-ci la plus intéressante semble être l'utilisation de techniques d'analyse statique du code (autres que le typage) pour déterminer l'absence d'ambiguïtés dans la sélection. De telles techniques sont déjà utilisées dans la programmation à objets: par exemple, Eiffel [Mey91] les utilise pour vérifier que l'appel de méthodes spécialisées de manière covariante ne puisse pas soulever l'exception “message non compris”. Il est important de remarquer que la critique la plus importante émise à l'encontre de l'utilisation de ces techniques, c'est à dire le manque de modularité, ne s'applique pas à notre approche. En effet, un des problèmes de la solution de Eiffel est qu'elle effectue une analyse globale ; ainsi le fait d'introduire de nouvelles définitions dans un programme peut causer le rejet, de la part du module de contrôle des types, d'anciennes parties acceptées auparavant. Dans notre cas, par contre, cet ajout ne cause pas le rejet de modules existants mais, simplement, peut demander la spécialisation d'anciens foncteurs pour les nouvelles structures introduites, ce qui constitue une exigence habituelle de la spécification des modules. Nous laissons l'étude de ces techniques à des travaux futurs et renvoyons à [AC96b] pour plus de détails.

1.6 Conclusion

Avec ce travail nous avons voulu dépasser deux limitations du système de modules de SML. La première est l'absence de la surcharge qui oblige le programmeur à gérer directement l'emploi d'une opération sur des types de données différents. La deuxième est l'impossibilité d'effectuer une programmation incrémentale et une réutilisation du code comme celles qui caractérisent les langages orientés objets.

Il arrive qu'un programmeur effectue la même opération sur des structures de signatures différentes et, donc, qu'il utilise de noms de foncteurs différents pour la même opération, un foncteur différent pour chaque type de données. Chaque fois que l'emploi de cette opération est nécessaire,

la tâche de choisir le foncteur approprié est laissée à la charge du programmeur. Avec l'introduction de la surcharge ce choix est délégué au système : le programmeur définit un seul foncteur, surchargé, ayant comme nom le nom de l'opération ; à chaque emploi de l'opération il lui suffit d'utiliser ce nom : le système choisit automatiquement le code à utiliser. Ceci évite une inutile et dangereuse duplication de code à tout endroit où seul le nom de l'opération suffit (sans surcharge il faudrait faire autant de copies de ce code que de types de données, en remplaçant dans chaque copie le nom de l'opération par le nom du foncteur approprié, comme nous avons montré par l'exemple avec `Use`). Avec l'introduction de liaison tardive cette réutilisation du code devient encore plus importante, au point que le style même de programmation peut en être affecté. Ainsi avec la liaison tardive on passe d'une simple organisation modulaire du travail à un gestion incrémentale du développement d'un projet. Le succès de la programmation orientée à objets témoigne de l'importance d'une telle possibilité.

L'idée fondamentale qui sous-tend notre approche est donc celle de déléguer au système le plus de travail possible : la surcharge gère le choix des différents codes associés à une même opération et la liaison tardive permet d'affiner ce choix par la prise en compte des contraintes supplémentaires de spécialisation.

Chapitre 2

Démonstration automatique

Articles de référence: [CC01]

Nous avons vu dans le chapitre précédent que l'ajout de la surcharge avec liaison tardive à un système de modules pour SML permet une programmation de type incrémentale et la réutilisation de code. Toutefois dans le système présenté la correction du système de types reste un problème ouvert. En fait la manière standard pour prouver la correction, la démonstration de la propriété de *subject reduction*, ne peut pas être utilisée car le système de Leroy (et par conséquent le nôtre qui en est une extension) ne satisfait pas cette propriété. Mais tandis que Leroy a pu prouver la correction de son système par un biais sémantique, sa technique ne peut pas être utilisée pour les foncteurs surchargés car leurs bases théoriques ne sont pas encore bien établies. Un premier pas vers l'établissement de telles bases, et donc la résolution de ce problème, est constitué par le système logique que nous présentons dans ce chapitre. Ce système intègre trois caractéristiques différentes: le sous-typage, la surcharge avec liaison tardive et les types dépendants. Nous connaissons déjà les deux premières caractéristiques. Voyons brièvement ce que sont les types dépendants.

Les *types dépendants* sont des types qui dépendent de termes. Les tableaux en sont un exemple typique. Considérons par exemple des tableaux de caractères. Dans les langages de programmation on ne rencontre pas un type "`array of char`" mais plutôt une *famille* de types `char[1]`, `char[2]`, ... , où `char[`n`]` dénote le type des tableaux de caractères de longueur n. Considérons la fonction `string_to_array` qui transforme une chaîne de caractères s dans un tableau de type `char[`*length*(s)`]`. Il est possible d'exprimer le type de cette fonction par un type dépendant:

$$\texttt{string_to_array} : \pi s\colon \texttt{string}.\texttt{char}[\mathit{length}(s)]$$

En français le jugement ci-dessus exprime que `string_to_array` est une fonction qui, appliquée à une chaîne de caractères s, rend un résultat de type `char[`*longueur_de*(s)`]` (π est le lieur —*binder*— de la variable de terme s). Ainsi les types dépendants permettent d'exprimer une relation entre l'argument d'une fonction et *le type* de son résultat.

Les types dépendants sont à la base de plusieurs applications informatiques comme la démonstration automatique —e.g., [HHP93, CAB$^+$86]— ou, comme nous avons vu lors de la section précédente les systèmes de modules (un module peut exporter des fonctions dont le type est défini dans le module même, et donc le type résultat d'un foncteur peut dépendre du module auquel le foncteur est appliqué).

D'un point de vue technique, la contribution de ce chapitre est double: (i) la définition d'un système de déduction du sous-typage pour les types dépendants qui possède de très bonnes propriétés méta-théoriques et qui en plus est facilement extensible et (ii) la définition d'une discipline de types pour la surcharge à liaison tardive en présence de types dépendants.

2.1 Motivations

Il y a trois motivations principales à ce travail. L'une, nous l'avons déjà dit, est de donner une base formelle à l'étude du chapitre précédent. Mais, selon nous, le fait que le besoin de sous-typage et de surcharge se fasse ressentir dans le domaine de la démonstration automatique (leur absence rendant les codages logiques bien plus complexes) constitue une motivation bien plus importante. Une motivation supplémentaire est que l'utilisation de la surcharge à liaison tardive, en permettant la réutilisation de code est susceptible d'introduire un style de "programmation" à objet en démonstration automatique. Examinons ces dernières deux motivations plus en détail.

Codages Logiques

La théorie de types dépendants $\lambda\Pi$ [HHP93] (voir section 2.2.2 pour une introduction succincte) est utilisée pour spécifier des systèmes logiques. Pfenning [Pfe93] a démontré que l'absence de sous-typage conduit à des codages très compliqués, en particulier pour les sous-ensembles de formules. Par exemple, considérons les formules de la logique propositionelle:

$$F ::= A \mid \neg F \mid F \wedge F \mid F \vee F \mid F \Rightarrow F$$

où A dénote les formules atomiques. Une telle définition peut être représentée par les déclarations de type suivantes:

$$\begin{array}{ll} F & : \star \\ \neg & : F \to F \\ \wedge & : F \to F \to F \\ \vee & : F \to F \to F \\ \Rightarrow & : F \to F \to F \end{array}$$

Intuitivement $\star$ dénote l'ensemble de tout les types. Ainsi $F : \star$ signifie "F est un type". Si Γ est l'ensemble des déclarations ci-dessus, un exemple de codage formel est:

$$\Gamma, a : F, b : F \vdash \Rightarrow (\wedge ab) : F$$

Considérons maintenant le sous-ensemble de F défini par:

$$F_1 ::= A \mid \neg F_1 \mid F_1 \vee F_1$$

Il existe plusieurs manières de représenter F_1 en $\lambda\Pi$ (voir [Pfe93]) mais elles sont toutes compliquées et inefficaces. C'est pourquoi Pfenning propose d'étendre $\lambda\Pi$ par des types intersection (voir [BCDC83, CD80]) et du *subsorting*. Ce dernier, dénoté par $<:$ n'est qu'une forme restreinte de sous-typage. Il est alors possible d'avoir une représentation élégante de F_1

$$\begin{array}{rll} A & <: F_1 & \qquad A \text{ est une } \textit{subsort} \text{ de } F_1 \\ F_1 & <: F & \qquad F_1 \text{ est une } \textit{subsort} \text{ de } F \\ \neg & : (F \to F) \cap (F_1 \to F_1) & \\ \vee & : (F \to F \to F) \cap (F_1 \to F_1 \to F_1) & \end{array}$$

À la place du subsorting et des types intersection, dans ce chapitre nous proposons $\lambda\Pi^{\&}$ une extension de $\lambda\Pi$ par le sous-typage et les types surchargés, grâce auquel on peut définir le codage suivant:

$$\neg : \{F \to F, F_1 \to F_1\} \qquad\qquad \vee : \{F \times F \to F, F_1 \times F_1 \to F_1\}$$

Le sous-typage et la surcharge sont respectivement plus riches que le subsorting et les types intersection. La différence entre les types intersection et les types surchargés est qu'un terme appartient à l'intersection $A \cap B$ si et seulement si il appartient à A et à B. Tandis que un terme de $\{A, B\}$ est l'union de deux termes distincts, l'un appartenant à A l'autre à B. Les types intersection sont moins contraignants que ceux surchargés car $\{A, B\}$ n'est défini que si A et B sont des types flèche

(tandis que dans $A \cap B$ ils peuvent être quelconques). Mais si on se restreint aux types flèche les types surchargés sont d'une certaine manière plus expressifs que les intersections car il est correct de considérer un terme de type intersection comme le cas dégénéré d'un terme de type surchargé où les sous-termes qui le forment sont tous égaux.

Dans le système de Pfenning la décidabilité est obtenue grâce à l'utilisation de sortes (*sorts*). Les sortes sont une simplification des types. Le prix à payer pour l'utilisation de sortes est assez important car elles ne peuvent pas apparaître dans les λ-abstractions et il n'est donc pas possible d'écrire des fonctions dont le domaine est limité par du *subsorting*. C'est pourquoi Aspinall et Compagnoni ont introduit $\lambda P_{\leq}$ une extension de $\lambda\Pi$ avec du sous-typage [AC96c]. Si leur système ne présente pas le problème pour les fonctions lié aux sortes, il n'inclut pas non plus des types intersection où surchargés permettant le codage des exemples de Pfenning. Le système présenté dans ce chapitre ne présente aucun de ces deux problèmes: puisqu'il est pourvu du sous-typage (mais différent de celui de [AC96c]) il permet une utilisation générale des fonctions, mais étant aussi muni des types surchargés il permet le codage des exemples de Pfenning.

Spécialisation de preuves

Considérons à nouveau les types F et F_1 de la section précédente. Puisque $F_1 \leq F$ alors $F \rightarrow \text{Bool} \leq F_1 \rightarrow \text{Bool}$. Par conséquent une fonction de $F \rightarrow \text{Bool}$ est aussi une fonction de $F_1 \rightarrow \text{Bool}$. En particulier une fonction de décision p pour la logique propositionnelle (type $F \rightarrow \text{Bool}$) est aussi une fonction de décision pour le formules F_1 (type $F_1 \rightarrow \text{Bool}$).

Ainsi le sous-typage est un premier ingrédient pour la réutilisation de code car il permet l'utilisation de p, écrite à l'origine pour des formules F, sur des arguments de type F_1. Ceci toutefois n'est qu'une forme très limitée de réutilisation car elle ne fait que rendre p un peu plus polymorphe. Le vrai gain de la réutilisation de code est obtenu par la spécialisation du code. Considérons à nouveau la fonction de décision p. Elle est NP-hard. Toutefois, de par la structure de F_1 il est possible de construire une fonction de décision polynomiale p_1 pour F_1. Ainsi la manière la plus efficace de construire une fonction de décision *dec* est d'utiliser p_1 pour les formules F_1 et p sinon. Ceci peut être obtenu en définissant *dec* comme la fonction surchargée $p\&p_1$, dont le type est $\{F \rightarrow \text{Bool}, F_1 \rightarrow \text{Bool}\}$. L'utilisation de la liaison tardive assure que la branche la plus efficace sera toujours exécutée. Bien sûr une programmation de type incrémentale est aussi envisageable (voir [CC01]).

2.2 Aperçu

Dans ce bref rapport nous ne pouvons donner qu'une description intuitive de $\lambda\Pi^{\&}$. Nous procéderons par pas. Ainsi nous commençons par introduire les types dépendants, c'est-à-dire $\lambda\Pi$ (§ 2.2.2). Après nous définissons notre sous-typage pour les types dépendants, c'est-à-dire le système $\lambda\Pi_{\leq}$. Quoique $\lambda\Pi_{\leq}$ soit similaire à $\lambda P_{\leq}$ d'Aspinall Compagnoni, il améliore ce dernier en plusieurs aspects. Nous n'avons pas ici l'espace nécessaire à une comparaison détaillée des deux systèmes, comparaison que le lecteur pourra trouver dans la section 3.3.2 de [CC01]. Nous nous limiterons à remarquer que nous n'aurions pas pu utiliser $\lambda P_{\leq}$ comme base pour $\lambda\Pi^{\&}$ car $\lambda P_{\leq}$ est très difficilement extensible. C'est pourquoi nous avons défini $\lambda\Pi_{\leq}$ qui est très modulaire et facilement extensible, par des types surchargés en particulier.

2.2.1 Une brève introduction à la théorie des types dépendants

Les types sont utilisés pour classer les termes, mais en présence de types dépendants nous avons vu que les types et le termes ne sont pas complètement disjoints. Ceci est par exemple montré

par le type $\pi s\colon \texttt{string}.A(s)$ de la fonction string_to_array où la variable de terme s apparaît. Nous avons aussi rencontré des *familles de types* telles que la famille des tableaux de caractères $\{\texttt{char[1]},\ \texttt{char[2]},\dots\}$. Les familles de types sont des transformations de termes dans des types. La famille ci-dessus correspond à la transformation $n \mapsto \texttt{char}[n]$ qui possède le "kind" $\Pi n\colon \texttt{nat}.\star$. Pour les familles de types on utilise la Λ-notion. Ainsi

$$\Lambda n\colon \texttt{nat}.\texttt{char}[n] \ : \ \Pi n\colon \texttt{nat}.\star$$

dénote la famille de tableaux ci-dessus. Par β-réduction $(\Lambda n\colon \texttt{nat}.\texttt{char}[n])(3)$ est le type des tableaux de caractères de taille 3, c'est-à-dire char[3].

Plus généralement, nous considérons un système de types où les termes sont classés par les types et les types (plus précisément, les familles de types) par des "kinds". Les types peuvent être des types atomiques (e.g. int), des applications de familles de types (s'ils ont le kind $\star$) ou des types de la forme $\pi x\colon A.B$ utilisés pour typer les λ-abstractions (le type $A \to B$ du lambda-calcul simplement typé est le cas particulier de $\pi x\colon A.B$ où x n'apparaît pas libre dans B).

2.2.2 Le système $\lambda\Pi$

Formellement nous avons le système $\lambda\Pi$ tel qu'il est présenté dans [HHP93]:

Terms	M	$::=$	$x \mid \lambda x\colon A.M \mid MM$
Types	A	$::=$	$\alpha \mid \pi x\colon A.A \mid \Lambda x\colon A.A \mid AM$
Kinds	K	$::=$	$\star \mid \Pi x\colon A.K$
Contexts	Γ	$::=$	$<> \mid \Gamma, x : A \mid \Gamma, \alpha : K$

1. Un terme (dénoté par $M, N, \dots$) est soit une variable, soit une abstraction, soit une application.
2. Un type (dénoté par $A, B, C, \dots$) est soit un type atomique α, soit un π-type, soit une famille, soit l'application d'une famille.
3. Un kind est de la forme $\Pi x_1\colon A_1..x_n\colon A_n.\star$ avec $n \geq 0$
4. Un contexte est une liste éventuellement vide d'hypothèses de type $x\colon A$ ou d'hypothèses de kind $\alpha\colon K$. Si $x\colon A$ (respectivement, $\alpha\colon K$) apparaît dans Γ alors nous écrivons $x \in \mathit{Dom}(\Gamma)$ (respectivement, $\alpha \in \mathit{Dom}(\Gamma)$) et utilisons $\Gamma(x)$ (respectivement, $\mathit{Kind}_\Gamma(\alpha)$) pour dénoter A (respectivement, K).

La β-réduction, dénotée par $\to_\beta$, est la *clôture compatible* (ou *par contexte* voir [Bar84]) de l'union des deux notions de réduction suivantes:

$$(\lambda x\colon A.M)N \to_{\beta_1} M[x\colon = N]$$
$$(\Lambda x\colon A.B)N \to_{\beta_2} B[x\colon = N]$$

la β-conversion, dénoté par $=_\beta$, est la relation d'équivalence générée par la β-réduction. Le système de types est défini en figure 2.1.

2.2.3 Sous-typage pour $\lambda\Pi$: le système $\lambda\Pi_\leq$

Une première contribution de ce chapitre est la définition d'une relation de sous-typage pour $\lambda\Pi$ (nous invitons le lecteur à consulter [CC01] pour des explications plus détaillées). D'habitude l'ajout du sous-typage à un système de types qui en est dépourvu est fait en deux pas. D'abord la relation $\leq$ est définie sur les types bien formés; ensuite la règle de subsumption est ajoutée aux règles de typage. Cette règle est d'habitude de la forme suivante.

Subsumption $$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A \leq B}{\Gamma \vdash M : B}$$

Context Formation

F-EMPTY $\dfrac{}{<> \vdash \star}$

F-TERM $\dfrac{\Gamma \vdash A : \star \quad x \notin Dom(\Gamma)}{\Gamma, x : A \vdash \star}$

F-TYPE $\dfrac{\Gamma \vdash K \quad \alpha \notin Dom(\Gamma)}{\Gamma, \alpha : K \vdash \star}$

F-Π $\dfrac{\Gamma, x : A \vdash K}{\Gamma \vdash \Pi x \colon A.K}$

Typing

T-VAR $\dfrac{\Gamma \vdash \star \quad x \in Dom(\Gamma)}{\Gamma \vdash x : \Gamma(x)}$

T-λ $\dfrac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x \colon A.M : \pi x \colon A.B}$

T-APP $\dfrac{\Gamma \vdash M : \pi x \colon A.B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x : = N]}$

T-CONV $\dfrac{\Gamma \vdash M : A \quad \Gamma \vdash A =_\beta B}{\Gamma \vdash M : B}$

Kinding

K-VAR $\dfrac{\Gamma \vdash \star \quad \alpha \in Dom(\Gamma)}{\Gamma \vdash \alpha : Kind_\Gamma(\alpha)}$

K-π $\dfrac{\Gamma, x : A \vdash B : \star}{\Gamma \vdash \pi x \colon A.B : \star}$

K-Λ $\dfrac{\Gamma, x : A \vdash B : K}{\Gamma \vdash \Lambda x \colon A.B : \Pi x \colon A.K}$

K-APP $\dfrac{\Gamma \vdash A : \Pi x \colon B.K \quad \Gamma \vdash M : B}{\Gamma \vdash AM : K[x \colon = M]}$

K-CONV $\dfrac{\Gamma \vdash A : K \quad \Gamma \vdash K' \quad K =_\beta K'}{\Gamma \vdash A : K'}$

FIG. 2.1: *The* λΠ *type system*

Nous utilisons ici une approche différente, car nous voulons qu'une éventuelle extension du système soit très facilement définissable et soit conservative. Pour cela un des ingrédients principaux est d'avoir une définition de sous-typage qui soit indépendante de la bonne formation des types. C'est pourquoi nous définissons notre relation de sous-typage sur les pré-types (c'est-à-dire des types qui peuvent ne pas être bien formés) et nous ajoutons aux règles de la figure 2.1 une règle de subsumption modifiée de manière à contrôler la bonne formation des types concernés:

T-SUB $$\dfrac{\Gamma \vdash M : A \qquad \Gamma \vdash A \leq B \qquad \Gamma \vdash A, B : \star}{\Gamma \vdash M : B}$$

Le deuxième ingrédient pour qu'une éventuelle extension soit conservative est de ne pas utiliser dans le sous-typage une règle de transitivité. Il faut donc définir un système où cette règle soit *admissible* (c'est-à-dire , que pour toute instance de la règle il existe une déduction qui prouve la conséquence de la règle à partir de ses prémisses). Ceci peut être obtenu par les règles en figure 2.2

Donc $\lambda\Pi_\leq$ est défini par les règles en figure 2.2 plus celles en figure 2.1 où l'on aura ajouté la règle (T-SUB) ci dessus. Commentons chaque règle de sous-typage en commençant par les plus faciles:

- La règle S-π est la généralisation du sous-typage pour les flèches. Elle est contravariante sur les domaines et covariante sur les codomaines. Mais puisque la variable x peut apparaître libre dans les codomaines ces derniers sont contrôlés sous l'hypothèse que x appartient au domaine commun aux deux types, c'est-à-dire le plus petit.

S-ApT	$\dfrac{\Gamma \vdash \Gamma(\alpha)M_1..M_n \leq A}{\Gamma \vdash \alpha M_1..M_n \leq A}$
S-ApR	$\dfrac{M_1 =_\beta M'_1 \ \cdots \ M_n =_\beta M'_n}{\Gamma \vdash \alpha M_1..M_n \leq \alpha M'_1..M'_n}$
S-π	$\dfrac{\Gamma \vdash A' \leq A \qquad \Gamma, x : A' \vdash B \leq B'}{\Gamma \vdash \pi x\colon A.B \leq \pi x\colon A'.B'}$
S-Λ	$\dfrac{A' =_\beta A \qquad \Gamma, x : A' \vdash B \leq B'}{\Gamma \vdash \Lambda x\colon A.B \leq \Lambda x\colon A'.B'}$
S-ApSL	$\dfrac{\Gamma \vdash B[x\colon = M_1]M_2..M_n \leq C}{\Gamma \vdash (\Lambda x\colon A.B)M_1..M_n \leq C}$
S-ApSR	$\dfrac{\Gamma \vdash C \leq B[x\colon = M_1]M_2..M_n}{\Gamma \vdash C \leq (\Lambda x\colon A.B)M_1..M_n}$

FIG. 2.2: $\lambda\Pi_{\leq}$ *subtyping rules*

- La règle S-Λ établit que les familles de types qui, rappelons-le, sont des fonctions, sont comparées par points.
- S-ApSL et S-ApSR établissent que le sous-typage est invariant par β_2-réductions.
- S-ApR établit que la relation $\leq$ est réflexive sur les types atomiques, et qu'elle étend la réflexivité par points à toutes les applications possibles des types atomiques.
- La règle S-ApT (en combinaisons avec la réflexivité) produit la fermeture transitive des déclarations de sous-typage. Intuitivement, pour prouver que $\Gamma \vdash \alpha \leq \Gamma(\Gamma(\Gamma(\alpha)))$ on utilise trois applications consécutives de cette règle précédées par une application de S-ApR. Comme pour cette dernière, S-ApT est étendue par points à toutes les applications possibles.

2.2.4 Surcharge pour $\lambda\Pi_{\leq}$: le système $\lambda\Pi^{\&}$

Nous allons procéder exactement comme pour l'ajout de la surcharge au λ-calcul simplement typé (section I.2.2.1). Ainsi nous aurons des termes de la forme

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

Toutefois dans ce système les branches d'une fonction surchargée auront des π-types. Ainsi si M_i : $\pi x\colon A_i.B_i$ alors la fonction surchargée ci-dessus aura le type

$$\{\pi x\colon A_1.B_1\ ,\ \pi x\colon A_2.B_2, \ldots, \pi x\colon A_n.B_n\}$$

Comme d'habitude si nous appliquons cette fonction à un argument N de type A_j la branche sélectionnée est M_j, c'est-à-dire :

$$(\varepsilon\&M_1\&\ldots\&M_n)\bullet N \quad \twoheadrightarrow \quad M_j{\cdot}N \tag{2.1}$$

Sous-typage

Si l'on était en train d'étendre $\lambda\Pi$ on pourrait (presque) s'arrêter la. Toutefois nous sommes en train d'étendre $\lambda\Pi_{\leq}$, et donc nous devons prendre en compte le sous-typage. Une première conséquence de ce fait est qu'il faut définir la relation de sous-typage pour les nouveaux types surchargés. Ceci n'est pas très difficile car il s'agit de la simple généralisation de la règle pour $\lambda\&$:

$$\text{S-OVER} \qquad \frac{\forall j \in J\ \exists i \in I \qquad \Gamma \vdash \pi x{:}\ A_i.B_i \leq \pi y{:}\ C_j.D_j}{\Gamma \vdash \{\pi x{:}\ A_i.B_i\}_{i \in I} \leq \{\pi y{:}\ C_j.D_j\}_{j \in J}}$$

ce qui revient à demander que pour prouver que $\{\pi x{:}\ A_i.B_i\}_{i\in I}$ est un sous-type de $\{\pi y{:}\ C_j.D_j\}_{j\in J}$ il faut exhiber une fonction totale ϕ de J à I telle que pour tout $j \in J$ on peut prouver que $\pi x{:}\ A_{\phi(j)}.B_{\phi(j)} \leq \pi y{:}\ C_j.D_j$.

La deuxième conséquence de l'utilisation du sous-typage est que dans une réduction telle que (2.1) le type de N peut ne pas être l'un des A_i, mais plutôt les sous-type d'un ou plusieurs d'entre eux. Comme toujours nous sélectionnons la branche qui approche au mieux le type de N. Ainsi si $N : A$ nous sélectionnons la branche M_j telle que $A_j = \min_{i=1..n}\{A_i \mid A \leq A_i\}$. L'existence de ce minimum est assurée par des restrictions de bonne formation pour les types.

Kinding

Comme dans le cas de $\lambda\&$ pour obtenir la propriété de subject reduction il faut imposer des conditions de bonne formation sur les types surchargés. Dans le cas de types dépendants les restrictions à utiliser sont la généralisation des restrictions pour $\lambda\&$ plus une condition supplémentaire sur la forme des π-types qui apparaissent dans un type surchargé. Plus précisément, pour un environnement Γ donné un type $\{\pi x:\ A_i.B_i\}_{i\in I}$ est bien formé s'il est composé de types bien formés et s'il satisfait les trois conditions suivantes :

(Normal types) Pour tout $i \in I$ le type $\pi x:\ A_i.B_i$ est clos (il ne contient pas de variables de *terme* libres) et en forme normale.

(Covariant types) Pour tout $i, j \in I$ si $\Gamma \vdash A_i \leq A_j$ alors $\Gamma, x:\ A_i \vdash B_i \leq B_j$.

(Unique selection) Pour tout type A dont les variables libres sont déclarées dans *Dom*(Γ) l'ensemble $\{A_i \mid \Gamma \vdash A \leq A_i, i \in I\}$ est soit vide, soit il possède un plus petit élément.

Par rapport à $\lambda\&$ la définition de bonne formation utilise un environnement de types Γ. La différence est minime: en $\lambda\&$ on supposait travailler avec une relation de sous-typage prédéfinie sur les types atomiques; dans $\lambda\Pi^{\&}$ le sous-typage pour les types atomiques est défini par Γ. Il faut aussi noter que toutes ces conditions utilisent le sous-typage. Le fait d'avoir défini le sous-typage de manière indépendante de la bonne formation nous évite une circularité qui empêcherait cette extension. Examinons chaque condition plus en détail:

– Imaginons que la condition [normal types] ne soit pas satisfaite et que donc des types surchargés ouverts soient admis dans le calcul. Nous pouvons donc écrire le terme suivant:

$$M_1\&^{\{\pi x:Ay.B,\pi x:AN.B\}}M_2$$

où A est une famille de types avec kind $\Pi x{:}\ A'.\star$ et N un terme de type A'.
Supposons que $y \notin Fv(M_1) \cup Fv(M_2) \cup Fv(N)$. Si nous plaçons ce terme dans un contexte tel que

$$(\lambda y{:}\ S.(M_1\&^{\{\pi x:Ay.B,\pi x:AN.B\}}M_2))N$$

alors par β-réduction nous obtiendrons $M_1[y{:} = N]\&^{\{\pi x:Ay.B,\pi x:AN.B\}[y:=N]}M_2[y{:} = N]$ c'est-à-dire

$$M_1\&^{\{\pi x:AN.B,\pi x:AN.B\}}M_2$$

qui clairement ne peut pas être typable car la sélection de la branche est ambiguë. Un problème similaire apparaît lorsque les π-types ne sont pas en forme normale.

- La condition [covariant types] généralise la condition correspondante de $\lambda\&$. Ainsi, si nous avons une fonction surchargée M de type $\{\pi x\colon A_1.B_1, \pi x\colon A_2.B_2\}$ où $A_2 < A_1$ et nous l'appliquons à un terme N dont le type statique est A_1, alors à la compilation l'application $M \bullet N$ aura le type B_1 (plus précisément, le type $B_1[x\colon = N]$). Mais si le type de la forme normale de N est A_2 alors le type dynamique de l'application sera B_2 (plus précisément, le type $B_2[x\colon = N]$). Ainsi il faut que $B_2 \leq B_1$ soit vérifié (plus précisément, sous l'hypothèse que $x : A_2$)
- La condition [unique sélection] assure l'absence d'ambiguïtés lors de la sélection d'une branche. Autrement dit, elle assure que lors qu'une fonction de type (bien formé) $\{\pi x\colon A_i.B_i\}_{i \in I}$ est appliquée à un terme de type A, alors il existe toujours une et une seule branche de type $\pi x\colon A_j.B_j$ telle que $A_j = \min_{i \in I}\{A_i | A \leq A_i\}$.[1]

Typage

Le système de types de $\lambda\Pi^{\&}$ est obtenu en ajoutant aux règles pour $\lambda\Pi_{\leq}$ les trois règles suivantes.

T-ε
$$\frac{\Gamma \vdash \star}{\Gamma \vdash \varepsilon : \{\}}$$

T-&
$$\frac{\Gamma \vdash M : \{\pi x\colon A_i.B_i\}_{i \leq n} \quad \Gamma \vdash N : \pi x\colon A_{n+1}.B_{n+1} \quad \Gamma \vdash \{\pi x\colon A_i.B_i\}_{i \leq n+1} : \star}{\Gamma \vdash M \&^{\{\pi x: A_i.B_i\}_{i \leq n+1}} N : \{\pi x\colon A_i.B_i\}_{i \leq n+1}}$$

T-OAPP
$$\frac{\Gamma \vdash M : \{\pi x\colon A_i.B_i\}_{i \leq n} \qquad \Gamma \vdash N : A_j}{\Gamma \vdash M \bullet N : B_j[x\colon = N]}$$

Ces règles n'ont pas besoin d'être beaucoup commentées. Il suffit de remarquer que dans la règle d'introduction des types surchargés, (T-&), il faut contrôler la bonne formation du type surchargé résultant, et en particulier la satisfaction des trois conditions que nous avons vues dans la section précédente.

Réduction

La réduction pour l'application de fonctions surchargées dans un contexte Γ est ainsi définie:

Si 1. N est clos et en forme normale,
2. il existe $i \in [1..n]$ tel que $\Gamma \vdash N\colon A_i$ et $\forall j \in [1..n]\ \ \Gamma \vdash N\colon A_j \Rightarrow \Gamma \vdash A_i \leq A_j$

alors
$$(M_1 \&^{\{\pi x: A_h.B_h\}_{h=1..n}} M_2) \bullet N \rightarrow_{\beta\&} \begin{cases} M_1 \bullet N & \text{for } i < n \\ M_2 \cdot N & \text{for } i = n \end{cases}$$

La $\beta^{\&}$-réduction est bien plus simple que la définition ci-dessus ne laisse supposer. La règle établit que si nous passons un argument N de type A_i à la fonction surchargée $(M_1 \&^{\{\pi x: A_h.B_h\}_{h=1..n}} M_2)$ nous sélectionnons la branche définie pour A_i. Mais pour cela il faut que deux conditions soient respectées. La première est que N soit clos et en forme normale, ce qui est nécessaire pour la liaison tardive. La seconde est qu'il existe une branche plus spécifique qui soit compatible avec le type

1. La restriction dans [unique sélection] que les variables libres de A soient contenues dans $Dom(\Gamma)$ est, quoique naturelle, très technique car nécessaire pour assurer que la satisfaction de [unique sélection] soit invariante par substitution. Par exemple si nous avions imposé la condition bien plus restrictive $\Gamma \vdash A : \star$, cette propriété n'aurait pas été vérifiée.

dynamique de l'argument. Autrement dit, nous considérons l'ensemble des types que nous pouvons déduire pour N (contrairement à λ& nous travaillons ici en présence de subsumption), nous effectuons toutes les sélections possibles basées sur le minimum et parmi les branches sélectionnées nous prenons la plus spécifique.

Enfin il faut noter que lorsque la réduction est effectuée, toute expression qui apparaît dans le redex est close. Ainsi la définition de $\beta^{\&}$ ne dépend pas des déclarations de sous-typage contenues dans Γ.[2]

EXCURSUS TECHNIQUE *La condition que N soit clos et en forme normale est assez forte. Comme expliqué pour λ& dans la section 7.2 de [Cas97a] cette condition peut être affaiblie. Par exemple on peut toujours effectuer tranquillement la réduction chaque fois qu'il n'y a qu'un choix possible, même si l'argument n'est pas clos ou en forme normale. Ainsi une généralisation possible de la réduction ci-dessus est la suivante:*

Pour tout A_i tel que $\Gamma \vdash N : A_i$ *et* $\forall j \in [1..n]\ \ \Gamma \vdash N : A_j \ \Rightarrow\ \Gamma \vdash A_i \leq A_j$,
si N est clos et en forme normale **ou** $\{A_j | 1 \leq j \leq n, \Gamma \vdash A_j \leq A_i\} = \{A_i\}$, *alors*

$$(M_1 \&^{\{\pi x : A_h . B_h\}_{h=1..n}} M_2) \bullet N \rightarrow_{\beta^{\&}} \begin{cases} M_1 \bullet N & \textit{for } i < n \\ M_2 \cdot N & \textit{for } i = n \end{cases}$$

Autrement dit, si nous avons une application nous voyons quelle branche serait sélectionnée et si il n'y en a pas d'autres définies pour un domaine plus petit nous sommes sûrs que la sélection ne peut plus changer, même si N n'est pas clos et en forme normale. Cette règle est intéressante car, par exemple, elle nous permet de déduire sous un contexte où n: `int` *que:*[3]

$$((\lambda x : \texttt{int}.x + x) \& (\lambda x : \texttt{string}.x @ x)) \bullet n \quad \twoheadrightarrow \quad n + n$$

Toutefois, nous préférons utiliser la version moins générale car cette généralisation compliquerait de manière importante la théorie (par exemple la remarque en note 2 de la page 63, ne serait plus valable).

Exemples

Comme premier exemple de l'utilisation des types dépendants nous montrons ce que signifie au niveau des types de généraliser la fonction `string_to_array`, définie au début du chapitre, pour qu'elle puisse être aussi appliquée aux nombres naturels. Puisque un nombre naturel n nécessite $\lceil \log_{10} n \rceil$ chiffres pour être représenté, cette généralisation aura le type suivant

$$\{\ \pi x : \texttt{nat.char[}\lceil \log_{10} x \rceil\texttt{]}\ ,\ \pi x : \texttt{string.char[}\textit{length}(x)\texttt{]}\ \}$$

Pour un exemple plus détaillé nous allons montrer deux codages différents du produit cartésien. Le premier, qui nécessite de types indexés, est plus compliqué mais aussi plus efficace. Le deuxième est plus simple, il fonctionne pour tous les types mais puisque il utilise la liaison tardive, il demande l'utilisation des types à l'exécution.

Soit ι un type atomique avec deux constantes $1 : \iota$ et $2 : \iota$, et A un type indexé sur ι, ce que l'on peut exprimer par l'environnement suivant: $\Gamma_0 \equiv \iota : \star\ ,\ 1 : \iota\ ,\ 2 : \iota\ ,\ A : \Pi x : \iota.\star$. Maintenant ajoutons une nouvelle constante $? : \pi x : \iota.A1 \rightarrow A2 \rightarrow Ax$ (noter que $A1$, $A2$, Ax, $x1$, etc. sont les applications $A(1)$, $A(2)$, $A(x)$ et $x(1)$) avec la sémantique suivante:

$$?xa_1a_2 = \begin{cases} a_1 & \text{for } x = 1 \\ a_2 & \text{otherwise} \end{cases}$$

2. Ceci simplifie de manière significative la définition de la réduction qui autrement aurait du être paramétrée par Γ.
3. Nous utilisons @ pour dénoter la concaténation de chaînes de caractères.

(le codage de ? est montré dans le dernier exemple de cette section). Nous pouvons maintenant utiliser les types dépendants pour coder le produit cartésien de $A1$ et $A2$:

$$\begin{array}{rcl} A1 \times A2 & = & \pi x\colon \iota.Ax \\ (_,_) & = & \lambda a_1\colon A1.\lambda a_2\colon A2.\lambda x\colon \iota.?xa_1a_2 \\ \text{fst} & = & \lambda x\colon A1\times A2.x1 \\ \text{snd} & = & \lambda x\colon A1\times A2.x2 \end{array}$$

Un codage plus simple peut être obtenu en utilisant les types surchargés en combinaison avec deux type atomiques α_1 et α_2 et deux constantes $p_1 : \alpha_1$ et $p_2 : \alpha_2$ (comme d'habitude nous omettrons les ε).

$$\begin{array}{rcl} A1 \otimes A2 & = & \{\pi x\colon \alpha_1.A1\,, \pi x\colon \alpha_2.A2\} \\ (_,_) & = & \lambda a_1\colon A1.\lambda a_2\colon A2.(\lambda x\colon \alpha_1.a_1 \,\&\, \lambda x\colon \alpha_2.a_2) \\ \text{fst} & = & \lambda x\colon A1\otimes A2.x \bullet p_1 \\ \text{snd} & = & \lambda x\colon A1\otimes A2.x \bullet p_2 \end{array}$$

Il est maintenant possible de définir des opérateurs *first* et *second* génériques qui marchent avec les deux codages. Par exemple *first* peut être ainsi défini:

$$(\lambda x\colon (A1\times A2).x1 \;\&\; \lambda x\colon (A1\otimes A2).x \bullet p_1)$$

dont le type est

$$\{\pi x\colon (\pi y\colon \iota.Ay).A1\;,\; \pi x\colon \{\pi y\colon \alpha_1.A1\,, \pi y\colon \alpha_2.A2\}.A1\}$$

Toutes ces fonctions peuvent être appliquées à des paires de termes dont les types respectifs sont des sous-types de $A1$ et $A2$.

Comme dernier exemple imaginons que nous voulions utiliser les types dépendants pour coder des triplets. Nous voulons définir le codage de manière utiliser des triplets où des paires sont attendues (comme s'ils étaient des enregistrements avec étiquettes 1, 2, et 3). Ceci peut être obtenu par la concaténation des environnement suivants:

$\Gamma_0 \equiv \iota_{123}\colon\star\,,\ \iota_3 \leq \iota_{123}\colon\star\,,\ \iota_{12} \leq \iota_{123}\colon\star\,,\ \iota_2 \leq \iota_{12}\colon\star\,,\ \iota_1 \leq \iota_{12}\colon\star\,,\ 1\colon\iota_1\,,\ 2\colon\iota_2\,,\ 3\colon\iota_3$

$\Gamma_1 \equiv A\colon \Pi x\colon \iota_{123}.\star\,,\ \bot\colon \pi x\colon \iota_{123}.Ax$

L'environnement Γ_0 déclare trois types singletons ι_1, ι_2, ι_3 contenant respectivement les constantes 1, 2, et 3. Il déclare aussi deux unions de ces singletons, ι_{12} (qui contient ι_1 et ι_2) et ι_{123} (qui contient tous les autres types). L'environnement Γ_1 déclare le type A indexé sur ι_{123}, ainsi qu'une constante $\bot$ telle que $\bot i : Ai$ pour $i = 1, 2, 3$.[4] Enfin nous codons ? ainsi:

$$? = \lambda x\colon \iota_{123}.\lambda a_1\colon A1.\lambda a_2\colon A2.\lambda a_3\colon A3.(\lambda y\colon \iota_1.a_1 \& \lambda y\colon \iota_2.a_2 \& \lambda y\colon \iota_3.a_3 \& \lambda y\colon \iota_{123}.\bot y) \bullet x$$

dont le type est[5] est $\pi x\colon \iota_{123}.A1 \to A2 \to A3 \to Ax$ et dont la sémantique est:

$$?xa_1a_2a_3 = \begin{cases} a_1 & \text{for } x = 1 \\ a_2 & \text{for } x = 2 \\ a_3 & \text{for } x = 3 \end{cases}$$

4. $\bot i$ intuitivement représente une constante indéfinie de type Ai; il s'agit d'un truc technique pour que le type du résultat de ? ci de suite puisse être Ax

5. Il faut noter que strictement parlant le type de la fonctions surchargée interne n'est pas bien formé: puisque $\iota_i \leq \iota_{123}$, alors [covariant types] impose $y\colon \iota_i \vdash Ai \leq Ay$. Ceci est sémantiquement vrai: puisque ι_i est un singleton qui ne contient que i, alors $y\colon \iota_i$ implique $y = i$. Mais une telle égalité pour être prouvée demande une extension telle que celle introduite par David Aspinall dans [Asp95], ce qui nous détournerait trop de notre but initial.

Avec ces déclarations le codage basé sur les types dépendants pour les paires ne change pas. Seul le constructeur de paires doit être modifié pour prendre en compte le quatrième argument de ?:

$$\begin{array}{rcl} A1\times A2 &=& \pi x\colon \iota_{12}.Ax \\ (_,_) &=& \lambda a_1\colon A1.\lambda a_2\colon A2.\lambda x\colon \iota_{12}.?xa_1a_2(\bot 3) \\ \text{fst} &=& \lambda x\colon A1\times A2.x1 \\ \text{snd} &=& \lambda x\colon A1\times A2.x2 \end{array}$$

Les triplets sont similaires

$$\begin{array}{rcl} A1\times A2\times A3 &=& \pi x\colon \iota_{123}.Ax \\ (_,_,_) &=& \lambda a_1\colon A1.\lambda a_2\colon A2.\lambda a_3\colon A3.\lambda x\colon \iota_{123}.?xa_1a_2a_3 \end{array}$$

Notons que par la règle S-π on a $A1\times A2\times A3 \leq A1\times A2$. Ainsi grâce au sous-typage il n'est pas nécessaire de définir les deux premières projections pour les triplets car les fonctions fst et snd définies pour les paires marchent aussi pour les triplets (en jargon orienté objet nous dirions que les triplets héritent les deux premières projection des paires). Par contre nous devons définir la troisième projection pour les triplets et, si nous le voulons, aussi pour les paires. Si une paire devient dynamiquement un triplet alors cette fonction de projection rend la troisième composante du triplet, sinon elle rend $\bot 3$:

$$\text{trd} = (\lambda y\colon A1\times A2.\bot 3)\ \&\ (\lambda y\colon A1\times A2\times A3.y3)$$

dont le type est $\{\pi y\colon (\pi x\colon i_{12}.Ax).A3\ ,\ \pi y\colon (\pi x\colon i_{123}.Ax).A3\}$.

Propriétés

Dans [CC01] nous prouvons que $\lambda\Pi^{\&}$ satisfait les bonnes propriétés théoriques. Nous ne pouvons ici qu'esquisser l'ordre logique de ces preuves.

D'abord, nous prouvons la confluence du calcul: soit $\beta = \beta_1 \cup \beta_2 \cup \beta^{\&}$, si $M \twoheadrightarrow_\beta M_1$ et $M \twoheadrightarrow_\beta M_2$ alors il existe N tel que $M_1 \twoheadrightarrow_\beta N$ et $M_2 \twoheadrightarrow_\beta N$.

Après, puisque nous sommes partis d'un ensemble de règles de sous-typage, celles de $\lambda\Pi_{\leq}$, qui n'incluent pas des règles générales de transitivité et réflexivité, nous devons prouver que la relation de sous-typage est un pré-ordre, c'est-à-dire que les deux règles sont admissibles.

Ensuite, nous prouvons la correction du système de types car il satisfait la subject reduction: si $\Gamma \vdash M : A$ et $M \twoheadrightarrow N$ alors $\Gamma \vdash N : A$

Grâce à l'absence d'une règle explicite de transitivité dans les règles de sous-typage il n'est pas trop difficile de prouver que $\lambda\Pi^{\&}$ est une extension conservative tant de $\lambda\Pi_{\leq}$ que de $\lambda\&$.

Un point très délicat est que puisque $\lambda\Pi^{\&}$ étend conservativement $\lambda\&$, il hérite de ce dernier la non terminaison: dans $\lambda\&$ il est possible de coder un opérateur de point-fixe de type $(A \to A) \to A$ pour tout type A (voir §6.2 de [Cas97a]), et la même technique s'applique aussi à $\lambda\Pi^{\&}$. Cependant il est possible dans $\lambda\&$ de caractériser une classe de sous-calculs qui sont fortement normalisants. Nous montrons qu'il est possible de procéder de la même manière pour $\lambda\Pi^{\&}$. L'intérêt d'un tel procédé dérive du fait que dans $\lambda\Pi^{\&}$ le sous-typage dépend de la β-conversion. Des termes peuvent apparaître dans les types et la β-conversion des termes est utilisée pour définir le sous-typage. La normalisation forte implique la décidabilité de la β-équivalence qui à son tour implique la décidabilité du sous-typage (et donc du typage) de $\lambda\Pi_{\leq}$ et des sous-systèmes fortement normalisants de $\lambda\Pi^{\&}$.

Dans [CC01] on trouvera aussi des résultats spécifiques à $\lambda\Pi_{\leq}$, comme le fait que les règles de sous-typage de $\lambda\Pi_{\leq}$ décrivent un algorithme, l'exacte correspondance entre $\lambda\Pi_{\leq}$ et $\lambda P_{\leq}$, et des résultats de décidabilité.

2.3 Conclusion

Dans ce chapitre nous avons montré comment fusionner dans un seul formalisme types dépendants, sous-typage et surcharge à liaison tardive. Le système logique ainsi obtenu est forcement très complexe, mais aussi très expressif.

La combinaison du sous-typage avec les types dépendants ne nécessite aucune justification supplémentaire car son intérêt est amplement discuté dans de nombreux articles de la littérature. Ces mêmes articles montrent qu'une certaine forme de surcharge est aussi souhaitable, mais jusqu'à présent ce besoin était partiellement satisfait par l'utilisation de types intersection ce qui ne constitue qu'une forme très limitée de surcharge.

Le manque de théories élégantes et, plus en général, d'études de la surcharge peut expliquer, sinon justifier, l'utilisation des types intersection comme ersatz de la surcharge. Cependant il n'est pas très difficile d'ajouter les fonctions surchargés aux types dépendants, une fois que nous avons une théorie complète de la surcharge. En réalité la complexité relative de $\lambda\Pi^{\&}$ ne dérive pas de l'utilisation de la surcharge mais de l'utilisation de la liaison tardive. C'est la liaison tardive qui nécessite de conditions très compliquées sur la formation des types et qui introduit une circularité dans les définitions de typage, de sous-typage et de bonne-formation. C'est donc à cause de la liaisons tardive que nous n'avons pas pu utiliser les systèmes existants de types dépendants et sous-typage mais que nous avons du développer une nouvelle théorie, $\lambda\Pi_{\leq}$ qui grâce à l'absence de circularité est adaptée à des extensions.

La récompense d'un tel effort est une système très puissant qui, grâce justement à la liaison tardive, permet le même type de programmation modulaire et incrémentale qui a été rendu populaire par les langages orientés objets.

Troisième partie

Distribution et mobilité

Dans les deux premières parties de ce mémoire nous avons étudié les systèmes de types pour les langages orientés objets séquentiels et transposé les résultas obtenus aux systèmes de modules et à la démonstration automatique. Dans les chapitres qui suivent nous allons mener une étude similaire dans le cas de paradigmes avec mobilité forte pour des systèmes distribués. Mais tandis que dans les cadres précédents nous avions des bases bien établies —les paradigmes à objets et leurs problèmes d'expressivité et de typage dans la première partie, les systèmes de modules et de déduction automatique dans la deuxième—-, dans le contexte que nous allons étudier il n'y a ni paradigmes consolidés ni problèmes bien définis. C'est pourquoi dans cette partie nous effectuons une recherche à spectre plus large, et en particulier nous nous attellerons à proposer différentes manières d'intégrer les méthodes et la mobilité (chapitre III.1), à étudier les problèmes de sécurité pour certain modèles de mobilité assez répandus (chapitre III.2) ainsi qu'à proposer de nouveaux modèles de mobilité (chapitres III.3 et III.4).

Chapitre 1

Modélisation d'objets mobiles

Articles de référence: [BC00, BCC00, BCC01d]

Les calculs d'agents mobiles reçoivent une attention grandissante dans la communauté des langages de programmation à mesure que les avancées dans les communications et dans les matériaux poussent le développement de la programmation distribuée à grande échelle. Les *agents* sont des entités qui effectuent des calculs et interagissent avec d'autres agents: le terme "mobile" implique que ces agents sont liés à des *locations* et que ce lien peut varier dans le temps. On parle ainsi de *mobilité forte* pour indiquer le fait qu'elle concerne la définition *et* l'état d'un agent, et pour la différencier de la *mobilité faible* (aussi appelé mobilité de code) qui ne concerne que la définition des agents car du code est déchargé et exécuté localement mais aucun état n'est sauvé ni restauré.

Indépendamment des nouvelles avancées en matière de technologies de communication, la programmation orientée objets que nous avons étudiée dans la première partie de ce mémoire, s'est établie comme un standard "de facto" pour le développement de système logiciels complexes.

C'est donc tout à fait naturellement que depuis l'étude de langages à objets séquentiels nous avons réorienté nos recherches vers le domaine de la mobilité. Un premier pas logique de cette démarche est l'étude des objets en présence de mobilité, ce que nous allons présenter dans ce chapitre.

L'approche suivie est originale car elle consiste à prendre un modèle de mobilité et lui greffer les méthodes et les envois de message. Une telle approche a déjà été appliquée à des calculs de processus pour obtenir des objets concurrents. Cependant, à notre connaissance, le système que nous présentons dans ce chapitre est le premier à suivre une telle approche en présence de mobilité (il en existe un postérieur, consistant à ajouter les notions de classes et d'objets au Join Calcul qui, au moment de la rédaction de ces notes, n'a pas encore été publié).

En fait, nous poussons l'approche un peu plus loin, car au lieu de choisir parmi les modèles de mobilité définis dans la littérature, un modèle particulier, nous définissons notre extension à objets de manière assez générique de façon que des choix différents pour les actions de mobilité produisent des instances distinctes de ce modèle.

Le cadre qui en résulte peut être considéré de plusieurs manières: (i) comme un calcul *concurrent* d'objets où les objets possèdent des noms explicites, dans le style de [GH98], (ii) comme un calcul d'objets *distribués* où les objets sont affectés à des locations différentes ou (iii), plus ambitieusement, comme un nouveau modèle de programmation distribuée où l'architecture conventionnelle client-serveur et la mobilité coexistent.

Dans ce chapitre nous esquissons le modèle générique qui est indépendant du modèle de mobilité choisi. Pour des raisons d'espace nous ne donnerons l'aperçu que pour une instance particulière du modèle, notamment pour le modèle de mobilité des ambients. D'autres instances sont possibles,

notamment en choisissant les modèles de mobilité du chapitre III.4, celui du chapitre III.3, ainsi que celui qui caractérise les *Safe Ambients* [LS00] (voir aussi le chapitre III.2). Ces deux dernières instances sont présentées dans les articles de référence du chapitre.

1.1 Description du modèle générique

Notre modèle d'objets mobiles est obtenu en ajoutant aux agents d'un calcul de processus avec mobilité des listes de méthodes. Sa définition est assez indépendante de la définition des agents et des processus du calcul sous-jacent. Nous supposons seulement l'existence des constructions suivantes: $P \mid Q$ dénotant la composition parallèle de deux processus P et Q, $a[P]$ dénotant le processus (ou agent) nommé a exécutant le processus P, $(\boldsymbol{\nu} x)P$ qui restreint la porté du nom x à P, et enfin $M.P$ qui effectue l'action décrite par l'expression M et ensuite continue en tant que P. Clairement les actions incluront des primitives de mobilité des agents, mais nous pouvons pour l'instant ignorer ces primitives. Le point central de notre approche est le mécanisme de nommage des processus. Les processus nommés sont des abstractions tant des agents que des locations. En outre la définition de processus autorise l'emboîtement des locations. Elles peuvent donc être structurées de manière hiérarchique et ainsi modéliser tant les noeuds d'un système distribué que les agents sur ce système [Car99b].

Plus précisément, le modèle générique d'objets mobiles résulte de l'inclusion dans les processus nommés d'*interfaces* comme dans $a[I\,;P]$, où P est un processus et I un ensemble (plus précisément, une liste) de définitions de méthodes.

1.1.1 Syntaxe

La syntaxe des agents est définie par les productions en Figure 1.1.

Processes	P	$::=$	$\mathbf{0}$	inactivity
		$\mid$	$P \mid P$	parallel composition
		$\mid$	$a[\,\mathrm{I}\,;P\,]$	agent
		$\mid$	$(\boldsymbol{\nu} x)P$	restriction
		$\mid$	$M.P$	action
Interfaces	I	$::=$	$\ell(\boldsymbol{x}) \triangleright \varsigma(z)P$	method
		$\mid$	$\mathrm{I} :: \mathrm{J}$	sequence
		$\mid$	$\varnothing$	empty interface
Patterns	$\boldsymbol{x}$	$::=$	x	variable
		$\mid$	$(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$	tuple ($n \geqslant 1$)
Expressions	M, N	$::=$	$a, b, \ldots, x, y \ldots$	name/variable
		$\mid$	$(M_1, \ldots, M_n)$	tuple ($n \geqslant 0$)
		$\mid$	$M.M$	path
		$\mid$	$M\ \mathsf{send}\ \ell\langle M\rangle$	method invocation

FIG. 1.1: *Syntax of Agents*

Méthodes. La définition d'une méthode est de la forme $\ell(\boldsymbol{x}) \triangleright \varsigma(z)P$, où ℓ est le nom de la méthode (c'est-à-dire, le message), $\boldsymbol{x}$ est le n-uplet des paramètres formels et $\varsigma(z)P$ son corps. Comme dans

le Calcul d'Objets [AC96a] la variable z liée par ς dénote le nom "interne" de l'agent dont la portée est le processus P. Nous abrégerons la syntaxe des méthodes par $\ell(\boldsymbol{x}) \triangleright P$ chaque fois que z n'apparaît pas libre dans P, ou quand la présence de ς est inintéressante.

Expressions La syntaxe des expressions et la sémantique qui leur est associée est ce qui différencie les différents modèles de mobilité. Dans le *Seal Calcul* (chapitre III.3), par exemple, les expressions qui effectuent la mobilité sont basées sur la synchronisation de canaux et agissent sur des agents qui son passifs par rapport à leur mobilité. Dans les *Ambients Mobiles* [CG98] au contraire les expressions pour la mobilité sont déclenchées par les mêmes agents qui effectuent le mouvement. Les *Ambients Sûrs* [LS00] modifient cette dernière sémantique en y ajoutant des mécanismes de synchronisation.

Puisque nous voulons définir une extension indépendante des primitives de mobilité sous-jacentes nous nous limitons à considérer un ensemble restreint d'expressions comprenant les noms, les n-uplets, les envois de message, et les chemins composés par plusieurs expressions. Ces expressions constituent le noyau de base qui sera ensuite étendu par les expressions propres à chaque modèle de mobilité.

Examinons en particulier les envois de message, qui sont de la forme $a\ \mathsf{send}\ \ell\langle M\rangle$ où a est le nom du destinataire et ℓ le message, ce qui a pour effet d'invoquer avec argument M la méthode associé dans a au message ℓ. Grâce à l'utilisation de processus nommés le format d'un envoi demande que le destinataire soit le *nom* d'un agent plutôt que, comme dans [AC96a], l'agent (l'objet, dans [AC96a]) même.

La sémantique de l'envoi de message est discutée en section 1.1.3. Avant cela nous définissons la relation de congruence structurale en présence des méthodes.

1.1.2 Congruence structurale

La congruence structurale est définie en termes d'une relation sur les interfaces, décrite en figure 1.2, qui permet de réarranger l'ordre des méthodes sans modifier le comportement de l'agent auquel ils appartiennent.

(Eq Meth Assoc)	$(\mathrm{I} :: \mathrm{J}) :: \mathbf{L}$	$\equiv_{\mathscr{I}}$	$\mathrm{I} :: (\mathrm{J} :: \mathbf{L})$	
(Eq Meth Comm)	$\mathrm{I} :: m(\boldsymbol{x}) \triangleright P :: \ell(\boldsymbol{y}) \triangleright Q :: \mathrm{J}$	$\equiv_{\mathscr{I}}$	$\mathrm{I} :: \ell(\boldsymbol{y}) \triangleright Q :: m(\boldsymbol{x}) \triangleright P :: \mathrm{J}$	$\ell \neq m$
(Eq Meth Over)	$\mathrm{I} :: \ell(\boldsymbol{x}) \triangleright P :: \ell(\boldsymbol{x}) \triangleright Q :: \mathrm{J}$	$\equiv_{\mathscr{I}}$	$\mathrm{I} :: \ell(\boldsymbol{x}) \triangleright Q :: \mathrm{J}$	
(Eq Meth Refl)	$\mathrm{I} \equiv_{\mathscr{I}} \mathrm{I}$			
(Eq Meth Symm)	$\mathrm{I} \equiv_{\mathscr{I}} \mathrm{J} \Rightarrow \mathrm{J} \equiv_{\mathscr{I}} \mathrm{I}$			
(Eq Meth Trans)	$\mathbf{L} \equiv_{\mathscr{I}} \mathrm{I},\ \mathrm{I} \equiv_{\mathscr{I}} \mathrm{J} \Rightarrow \mathbf{L} \equiv_{\mathscr{I}} \mathrm{J}$			

FIG. 1.2: *Equivalence for Methods*

Les définitions de méthodes ayant des noms ou des arités différentes peuvent être permutées librement, tandis que s'il y a plusieurs définitions pour une même méthode [1], la plus à droite masque (*overrides*) les restantes (cette définition est directement inspirée par la relation de *bookkeeping* de [FHM94]).

La congruence structurale est la plus petite congruence qui est un monoïde commutatif par rapport à $\mid$ et $\mathbf{0}$ et est close par rapport aux règles en figure 1.3.

1. Une méthode est identifiée par son arité et le message auquel elle est associée.

(Struct Res Dead)	$(\boldsymbol{\nu}x)\mathbf{0} \equiv \mathbf{0}$	
(Struct Res Res)	$(\boldsymbol{\nu}x)(\boldsymbol{\nu}y)P \equiv (\boldsymbol{\nu}y)(\boldsymbol{\nu}x)P$	$x \neq y$
(Struct Res Par)	$(\boldsymbol{\nu}x)(P \mid Q) \equiv P \mid (\boldsymbol{\nu}x)Q$	$x \notin fn(P)$
(Struct Res Agent)	$(\boldsymbol{\nu}x)a[\,\mathrm{I}\,;P\,] \equiv a[\,\mathrm{I}\,;(\boldsymbol{\nu}x)P\,]$	$x \notin fn(\,\mathrm{I}\,) \cup \{a\}$
(Struct Path Assoc)	$(M.M').P \equiv M.(M'.P)$	
(Struct Cong Agent Meth)	$\mathrm{I} \equiv_{\mathscr{I}} \mathrm{J} \Rightarrow a[\,\mathrm{I}\,;P\,] \equiv a[\mathrm{J}\,;P\,]$	

FIG. 1.3: *Structural Congruence for Agents*

Il est intéressant de noter que la définition ci-dessus est paramétrique dans les définitions d'expression et d'ensembles de noms libres, définitions qui sont propres à chaque modèle de mobilité pris en considération.

Le comportement des agents est défini par la relation de réduction $\rightarrow$ qui satisfait les règles structurales en figure 1.4 plus celles spécifiques à chaque primitive présente dans le modèle de mobilité considéré.

$$P' \equiv P,\ P \rightarrow Q,\ Q \equiv Q' \Rightarrow P' \rightarrow Q' \qquad P \rightarrow Q \Rightarrow a[\,\mathrm{I}\,;P\,] \rightarrow a[\,\mathrm{I}\,;Q\,]$$
$$P \rightarrow Q \Rightarrow (\boldsymbol{\nu}p)P \rightarrow (\boldsymbol{\nu}p)Q \qquad P \rightarrow Q \Rightarrow P \mid R \rightarrow Q \mid R$$

FIG. 1.4: *Structural Rules for Reduction*

1.1.3 Envoi de messages

La sémantique de l'invocation de méthode est basée sur l'idée de "self-substitution" qui caractérise les calculs d'objets de [AC96a]. Cependant, à la différence de ceux-ci, puisque les agents sont nommés on remplace *self* (c'est-à-dire, la variable abstraite par ς) par le nom de l'agent plutôt que par l'agent même.

Plusieurs choix sont possibles selon où l'invocation a lieu et où le corps de la méthode sélectionnée est exécuté. Dans la suite nous montrerons quelques options sans trop nous y attarder. Une discussion plus approfondie peut être trouvée dans [BCC01b].

Nous distinguons trois modalités différentes (remote, locale et interne). La figure 1.5 résume (dans un cadre très simplifié) plusieurs choix pour la sémantique de l'envoi de message.

Remote Invocation Mode

Cette modalité correspond à la situation où l'expéditeur et le destinataire sont en parallèle. On a alors le choix entre exécuter la méthode sélectionnée à l'intérieur de l'expéditeur, ce qui amène à un protocole du type *code on demand*:

(*Code on Demand*)

$$a[\,\mathrm{I}\,;S \mid b\ \mathsf{codsend}\ \ell\langle M\rangle.P\,] \mid b[\mathrm{J}::\ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;R\,]$$
$$\rightarrow\ a[\,\mathrm{I}\,;S \mid Q\{z:=b,\boldsymbol{x}:=M\} \mid P\,] \mid b[\mathrm{J}::\ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;R\,]$$

ou bien l'exécuter à l'intérieur du destinataire, ce qui correspond à une *remote method invocation*:

(*Remote Method Invocation*)

$$a[\,\mathrm{I}\,;S \mid b\ \mathsf{rmisend}\ \ell\langle M\rangle.P\,] \mid b[\mathrm{J}::\ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;R\,] \\ \rightarrow\ a[\,\mathrm{I}\,;S \mid P\,] \mid b[\mathrm{J}::\ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;Q\{z:=b,\boldsymbol{x}:=M\} \mid R\,]$$

Local Invocation Mode

Cette modalité correspond à la situation où l'expéditeur et le destinataire sont en relation père-fils. La présence de l'invocation dans le père ou dans le fils et l'exécution de la méthode sélectionnée dans l'expéditeur ou dans le destinataire donnent lieu à quatre cas possibles, dont seulement deux sont dignes d'intérêt, car plus uniformes (voir aussi [BCC01b]). Ce sont les cas où l'invocation et l'exécution de la méthode ont lieu dans le père

(*Parent-to-Child Invocation*)

$$b\ \mathsf{downsend}\ \ell\langle M\rangle.P \mid b[\,\mathrm{I}::\ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;R\,] \\ \rightarrow\ P \mid Q\{z:=b,\boldsymbol{x}:=M\} \mid b[\,\mathrm{I}::\ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;R\,]$$

et celui où l'invocation et l'exécution de la méthode ont lieu dans le fils:

(*Child-to-Parent Invocation*)

$$a[\,\mathrm{I}::\ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;R \mid b[\mathrm{J}\,;a\ \mathsf{upsend}\ \ell\langle M\rangle.P\,]\,] \\ \rightarrow\ a[\,\mathrm{I}::\ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;R \mid b[\mathrm{J}\,;P \mid Q\{z:=a,\boldsymbol{x}:=M\}\,]\,]$$

Internal Invocation Mode

Dans les deux modalités précédentes l'expéditeur et le receveur étaient toujours deux agents différents. La dernière modalité que nous considérons correspond à la situation où l'expéditeur et le receveur coïncident:

(*Internal Invocation*)

$$a[\,\mathrm{I}::\ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;R \mid a\ \mathsf{intsend}\ \ell\langle M\rangle.P\,] \\ \rightarrow\ a[\,\mathrm{I}::\ell(\boldsymbol{x}) \triangleright \varsigma(z)Q\,;R \mid P \mid Q\{z:=a,\boldsymbol{x}:=M\}\,]$$

L'intérêt d'une telle modalité est qu'elle permet de mieux comprendre la *self-invocation* standard des langages à objets, à laquelle dailleurs elle ressemble beaucoup. Encore une fois nous n'avons pas l'espace de rentrer dans les détails. Nous nous limitons à observer que dans le cas où l'invocation et l'exécution de la méthode ont lieu dans le même agent, les règles de réduction nous donnent automatiquement la sémantique de la self-invocation. Ainsi par exemple avec la *code on demand* nous avons que la configuration (noter la *self-invocation* dans le corps de ℓ_1)

$$a[\,\mathrm{I}\,;b\ \mathsf{codsend}\ \ell_1.P\,] \mid b[\ell_1 \triangleright \varsigma(z)z\ \mathsf{codsend}\ \ell_2, \ell_2 \triangleright Q\,;\,]$$

se réduit en deux pas à $a[\,\mathrm{I}\,;P \mid Q\,]$

	Body activated in the receiver	Body activated in the sender
Remote Invocation Mode	**(rmi)**	**(cod)**
$a[\,\mathrm{I}\,;\, b\ \mathsf{remote_send}\ \ell.P\,] \mid b[\ell \triangleright Q\,;\, R]$	$a[\,\mathrm{I}\,;\, P\,] \mid b[\ell \triangleright Q\,;\, Q \mid R]$	$a[\,\mathrm{I}\,;\, Q \mid P\,] \mid b[\ell \triangleright Q\,;\, R]$
Parent-to-Child Invocation		**(pci)**
$b\ \mathsf{downsend}\ \ell.P \mid b[\ell \triangleright Q\,;\, R]$	$P \mid b[\ell \triangleright Q\,;\, R \mid Q]$	$P \mid Q \mid b[\ell \triangleright Q\,;\, R]$
Child-to-Parent Invocation		**(cpi)**
$a[\ell \triangleright Q\,;\, R \mid b[\mathrm{J}\,;\, a\ \mathsf{upsend}\ \ell.P\,]]$	$a[\ell \triangleright Q\,;\, R \mid Q \mid b[\mathrm{J}\,;\, P\,]]$	$a[\ell \triangleright Q\,;\, R \mid b[\mathrm{J}\,;\, P \mid Q\,]]$
Internal Invocation	**(int)**	
$a[\ell \triangleright Q\,;\, R \mid a\ \mathsf{intsend}\ \ \ell.P\,]$	$a[\ell \triangleright Q\,;\, R \mid P \mid Q\,]$	

FIG. 1.5: *Method Invocation modes*

1.1.4 Réplication

La récursion dans les calculs de processus est typiquement obtenue par l'introduction d'un constructeur ! dont la sémantique est établie par la règle de congruence structurale $!P \equiv !P \mid P$. Dans notre étude nous n'avons pas inclus un tel constructeur car il peut être facilement simulé. Par exemple dans le cas de sémantique Parent-to-Child, nous avons le codage suivant:

$$!P \stackrel{def}{=} (\nu p)(p[\mathtt{bang} \triangleright \varsigma(z)z\ \mathsf{downsend}\ \mathtt{bang} \mid P\,;\ ] \mid p\ \mathsf{downsend}\ \mathtt{bang})$$

où $p \notin fn(P)$.

1.2 MA^{++}

Dans la section précédente nous avons présenté un modèle générique. Ce modèle doit sa généricité à sa définition qui est paramétrique dans l'ensemble des actions de mobilité. Nous pouvons donc l'utiliser comme base de départ pour en étudier des instances différentes. Dans ce mémoire nous nous limitons à présenter une seule instance de ce modèle, c'est à dire quand le modèle de mobilité sous-jacent est celui des Ambients Mobiles de Cardelli et Gordon [CG98]. Nous le choisissons à cause de sa richesse et car il nous permet d'introduire les concepts qui seront utilisés dans le prochain chapitre. Nous dénotons le modèle résultant par MA^{++}. Pour d'autres instances (basées sur les Ambients Sûrs et le Seal Calcul) et des exemples plus approfondis le lecteur peut consulter [BCC01b].

Nous commençons en introduisant les calcul d'ambient mobiles.

1.2.1 Ambients Mobiles

Le calcul des ambients épouse notre modèle générique car les *ambients* sont des processus nommés de la forme $a[P]$, éventuellement emboîtés, qui peuvent être composés en parallèle ($P \mid Q$), exercer une "capabilité" ($M.P$), déclarer de noms locaux ($(\nu x)P$), ou ne rien faire ($\mathbf{0}$). Ce qui caractérise les ambients mobiles ces sont les capabilités in, out and open qui permettent de reconfigurer dynamiquement leur structure arborescente. La sémantique qui est donnée à ces actions induit un modèle dont les interactions sont *unilatérales* et *subjectives*: elle sont unilatérales car un des deux agents qui participent à une action de mouvement ou d'ouverture subit tout simplement l'action (ceci

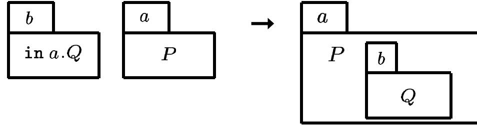
doit être comparé par exemple aux *Ambients Sûrs* où la relation de réduction demande une synchronisations entre les participants); elle sont subjectives car toute action de mouvement est exercée par l'agent objet du mouvement (ceci doit être comparé au Seal Calcul ou tout agent est déplacé par son entourage). Pour donner un exemple. considérons le système ci dessous:

$$a[\texttt{open}\ b.\texttt{in}\ c\,] \mid b[\texttt{in}\ a.\texttt{in}\ d\,].$$

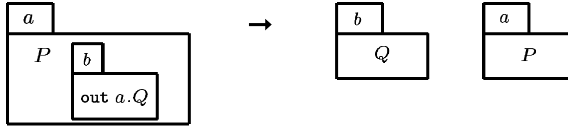
Dans cette configuration l'ambient b peut entrer dans a en exerçant la capabilité "$\texttt{in}\ a$" et se réduire à $a[\texttt{open}\ b.\texttt{in}\ c \mid b[\texttt{in}\ d\,]\,]$. Ensuite a peut dissoudre la frontière fournie par b en exerçant $\texttt{open}\ b$ et réduire à $a[\texttt{in}\ c \mid \texttt{in}\ d\,]$.

La sémantique des trois capabilités des ambients peut être décrite graphiquement comme il suit:

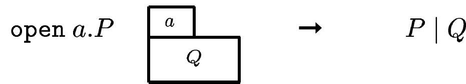
(IN)



(OUT)



(OPEN)



où tout ambient est représenté par une chemise qui peut contenir des processus ou d'autres chemises.

1.2.2 Syntaxe de MA^{++}

Formellement les processus et les méthodes de MA^{++} sont ceux défini en figure 1.1, tandis que la (désormais complète) syntaxe des expressions est:

$$M ::= a, \ldots, x \ldots \;\mid\; M\ \mathsf{send}\ \ell\langle M\rangle \;\mid\; M.M \;\mid\; \texttt{in}\ M \;\mid\; \texttt{out}\ M \;\mid\; \texttt{open}\ M$$

comme nous l'avions anticipé, nous n'avons inclus aucune primitive de communication ce qui fait de MA^{++} un sous-ensemble du noyau combinatoire des Ambients Mobiles. Toutefois, comme nous le montrerons, la communication de processus peut être encodée par les primitives de MA^{++} .

1.2.3 Réduction

Les définitions de congruence structurale et de réduction sont directement héritées du modèle générique. En figure 1.6 nous donnons les règles de réduction qui caractérisent MA^{++} .

La règle de réduction (*send*) implante la sémantique RMI que nous avons discuté dans la section 1.1.3. Les règles de réduction pour `in` et `out` sont définies exactement comme pour les *Ambients Mobiles*. La réduction pour `open` est différente selon que la suite de méthodes de l'ambient à ouvrir est vide ou pas. Si elle est vide, la réduction coïncide avec celle des Ambients Mobiles. Si par contre l'ambient a contient une suite de méthodes non vide, alors l'action $\texttt{open}\ a$ ne peut être exercée qu'à l'intérieur d'un autre ambient et son effet n'est pas seulement de libérer le processus inclus dans P, mais aussi de juxtaposer la suite de méthodes de l'ambient ouvert à celle de l'ambient ouvrant.

$$(in) \quad b[\,\mathrm{I}\,;\ \mathtt{in}\ a.P \mid Q\,] \mid a[\mathrm{J}\,;R\,] \rightarrow a[\,\mathrm{I}\,;R \mid b[\mathrm{J}\,;P \mid Q\,]\,]$$

$$(out) \quad a[\,\mathrm{I}\,;b[\mathrm{J}\,;\mathtt{out}\ a.P \mid Q\,] \mid R\,] \rightarrow b[\mathrm{J}\,;P \mid Q\,] \mid a[\,\mathrm{I}\,;R\,]$$

$$(open) \quad \mathtt{open}\ a.P \mid a[Q\,] \rightarrow P \mid Q$$

$$(update) \quad b[\,\mathrm{I}\,;\mathtt{open}\ a.P \mid a[\mathrm{J}\,;Q\,] \mid R\,] \rightarrow b[\,\mathrm{I}::\mathrm{J}\,;P \mid Q \mid R\,] \qquad \text{for } \mathrm{J} \neq \varepsilon$$

$$(send) \quad b[\,\mathrm{I}\,;a\ \mathsf{send}\ \ell\langle M\rangle.P \mid Q\,] \mid a[\mathrm{J}::\ell(\boldsymbol{x}) \triangleright \varsigma(z)R\,;S\,]$$
$$\rightarrow b[\,\mathrm{I}\,;P \mid Q\,] \mid a[\mathrm{J}::\ell(\boldsymbol{x}) \triangleright \varsigma(z)R\,;R\{z,\boldsymbol{x} := a, M\} \mid S\,]$$

FIG. 1.6: *MA^{++} reduction rules*

1.2.4 Exemples

Dans cette section nous montrons de manière succinte des codages significatifs qui peuvent être exprimés dans MA^{++} .

Parent-to-child

D'abord on peut facilement voir que quoique nous ayons choisi la RMI, les autres modalités d'invocation sont codables [BCC01b]. Par exemple la downsend peut être ainsi exprimée:

$$a\ \mathsf{downsend}\ \ell\langle M\rangle.P \stackrel{def}{=} (\boldsymbol{\nu} p,q)p[a\ \mathsf{send}\ \ell\langle M\rangle.q[\mathtt{out}\ p]\,] \mid \mathsf{open}\ q.\mathsf{open}\ p.P$$

où $(p, q \notin fn(M) \cup fn(P))$. En bref, nous créons un ambient temporaire p qui s'exécute en parallèle de receveur pour effectuer la RMI et nous utilisons l'ambient q comme un *lock* pour garantir que p ne sera détruit qu'après que le receveur aura servi l'invocation.

Par exemple nous avons la réduction suivante:

$$\begin{aligned} & a\ \mathsf{downsend}\ \ell.P \mid a[\ell \triangleright Q\,;R\,] \\ \equiv \quad & (\boldsymbol{\nu} p,q)p[\underline{a\ \mathsf{send}\ \ell\langle M\rangle}.q[\mathtt{out}\ p]\,] \mid \mathsf{open}\ q.\mathsf{open}\ p.P \mid a[\ell \triangleright Q\,;R\,] \\ \rightarrow \quad & (\boldsymbol{\nu} p,q)p[q[\underline{\mathtt{out}\ p}]\,] \mid \mathsf{open}\ q.\mathsf{open}\ p.P \mid a[\ell \triangleright Q\,;R \mid Q\,] \\ \rightarrow \quad & (\boldsymbol{\nu} p,q)p[\,] \mid q[\,] \mid \underline{\mathsf{open}\ q.\mathsf{open}\ p}.P \mid a[\ell \triangleright Q\,;R \mid Q\,] \\ \rightarrow^* \quad & P \mid a[\ell \triangleright Q\,;R \mid Q\,] \end{aligned}$$

Updates

La mise-à-jour d'une méthode ℓ pour un ambient $a[\,\mathrm{I}::\ell(\boldsymbol{x}) \triangleright \varsigma(z)P\,;Q\,]$ consiste à remplacer la définition courante P de ℓ par une nouvelle définition P' de façon à obtenir l'ambient $a[\,\mathrm{I}::\ell(\boldsymbol{x}) \triangleright \varsigma(z)P'\,;Q\,]$.

Une solution possible pour coder un tel comportement consiste à utiliser des ambients, que nous appellerons *updaters,* pour effectuer les mises-à-jour. Un updater renferme la nouvelle définition de la méthode et rentre dans l'ambient à modifier; tout ambient qui permet des mises-à-jour ouvre systématiquement les updaters qu'il reçoit.

Plus formellement nous allons encoder l'action $a\ \mathsf{update}\ \ell(\boldsymbol{x}) \triangleright \varsigma(z)P$, dont la sémantique informelle est "la méthode ℓ de l'ambient a prend la définition P", comme il suit:

$$a\ \mathsf{update}\ \ell(\boldsymbol{x}) \triangleright \varsigma(z)P \stackrel{def}{=} \mathrm{UPD}[\ell(\boldsymbol{x}) \triangleright \varsigma(z)P\,;\mathtt{in}\ a\,]$$

Ensuite nous définissons un ambient qui peut être mis-à-jour comme

$$a^{\star}[\,\mathrm{I}\,;P\,] \stackrel{def}{=} a[\,\mathrm{I}\,;!(\mathtt{open}\;\textsc{upd})\mid P\,]$$

Il est clair que la composition $a\;\mathtt{update}\;\ell(\boldsymbol{x})\triangleright\varsigma(z)P' \mid a^{\star}[\,\mathrm{I}::\ell(\boldsymbol{x})\triangleright\varsigma(z)P\,;Q\,]$ produit le comportement attendu:

$$a\;\mathtt{update}\;\ell(\boldsymbol{x})\triangleright\varsigma(z)P' \mid a^{\star}[\,\mathrm{I}::\ell(\boldsymbol{x})\triangleright\varsigma(z)P\,;Q\,] \quad \rightarrow^{*} \quad a^{\star}[\,\mathrm{I}::\ell(\boldsymbol{x})\triangleright\varsigma(z)P'\,;Q\,]$$

il faut noter que nous avons fait le choix délibéré de ne permettre que des mises-à-jour locales (c'est-à-dire, un ambient peut redéfinir seulement les méthodes de ses sous-ambients), mais bien d'autres formes d'update peuvent être codées (par exemple des updates synchrones, des self-updates, etc.).

π-calcul

Comme dernier exemple nous montrons comment coder les primitives de communication synchrone et asynchrone du π-calcul. Un codage similaire est présenté dans [CG98] et il est basé sur l'utilisation des primitives de communication des ambients. Ces primitives sont ici absentes, et notre codage utilise la possibilité spécifique à notre calcul d'échanger des messages.

Commençons par le cas synchrone. Un canal n est modélisé par un ambient "updatable" n, deux locks n^i et n^o, et un ambient auxiliaire $\bar{n}$ utilisé pour la communication. L'ambient n contient une méthode ch: un processus qui veut lire de n s'installe comme le corps de cette méthode, tandis que un processus qui veut écrire invoque ch avec comme argument l'objet de la communication:

$$\begin{array}{rcl}
(ch\;n) & \stackrel{def}{=} & n^{\star}[\mathtt{ch}(x)\triangleright\mathbf{0}\,]\mid n^{i}[\,]\\
n!\langle y\rangle.Q & \stackrel{def}{=} & \mathtt{open}\;n^{o}.n\;\mathtt{downsend}\;\;\mathtt{ch}(y).\mathtt{open}\;\bar{n}.(n^{i}[\,]\mid Q)\\
n?(x).P & \stackrel{def}{=} & \mathtt{open}\;n^{i}.\;n\;\mathtt{update}\;\mathtt{ch}(x)\triangleright(\bar{n}[\mathtt{out}\;n.P\,])\;.n^{o}[\,]
\end{array}$$

Le processus $n!\langle y\rangle.Q$ qui veut écrire y sur n essaie d'abord d'entrer en possession du lock de output n^o, ensuite envoie le message $\mathtt{ch}(y)$ à n et quand le protocole RMI est terminé (c'est-à-dire, après l'ouverture de l'ambient de transport $\bar{n}$) le processus continue comme Q en libérant le lock d'input n^i. Au début du protocole il n'y a aucun lock de output: ainsi le processus écrivant sur n est bloqué. Un processus $n?(x).P$ lisant de n d'abord s'approprie le lock $n^{\,i}$, ensuite s'installe comme corps de ch dans n, et enfin il libère le lock d'output $n^{\,o}$.

Les communication asynchrones sont obtenues par une légère variation de la définition de $n!\langle A\rangle.Q$, qui se résume à un mise entre parenthèses différente:

$$n!\langle y\rangle^{\text{asyn}}.Q \stackrel{def}{=} (\mathtt{open}\;n^{o}.n\;\mathtt{downsend}\;\;\mathtt{ch}(y).\mathtt{open}\;\bar{n}.(n^{i}[\,]))\mid Q$$

Nous pouvons donc utiliser ces techniques pour coder les primitives du π-calcul polyadique. Tout nom n du π-calcul devient un quadruplet: le nom n de l'ambient qui code le canal, les noms des deux locks n^i et n^o et l'ambient de transport auxiliaire $\bar{n}$, ce qui donne le codage de figure 1.7.

1.2.5 Système de types pour MA^{++}

Le typage des ambients hérite des idées des systèmes de types "standard" pour les ambients; toutefois la présence des méthodes permet une caractérisation plus structurée (et informative) des

$$
\begin{array}{rcll}
\langle\!\langle (\boldsymbol{\nu} n)P \rangle\!\rangle & \stackrel{def}{=} & (\boldsymbol{\nu} n, \bar{n}, n^i, n^o)(n^i[\,] \mid n^\star[\mathtt{ch}(x,\bar{x},x^i,x^o) \triangleright \mathbf{0}] \mid \langle\!\langle P \rangle\!\rangle) & \bar{n}, n^i, n^o \notin fn(\langle\!\langle P \rangle\!\rangle) \\
\langle\!\langle n!\langle y\rangle.Q \rangle\!\rangle & \stackrel{def}{=} & \mathsf{open}\ n^o.n\ \mathsf{downsend}\ \mathtt{ch}(y,\bar{y},y^i,y^o).\mathsf{open}\ \bar{n}.(n^i[\,] \mid Q) & \\
\langle\!\langle n!\langle y\rangle^{\text{asyn}}.Q \rangle\!\rangle & \stackrel{def}{=} & (\mathsf{open}\ n^o.n\ \mathsf{downsend}\ \mathtt{ch}(y,\bar{y},y^i,y^o).\mathsf{open}\ \bar{n}.n^i[\,]) \mid Q & \\
\langle\!\langle n?(x).P \rangle\!\rangle & \stackrel{def}{=} & \mathsf{open}\ n^i.n\ \mathsf{update}\ \mathtt{ch}(x,\bar{x},x^i,x^o) \triangleright (\bar{n}[\mathsf{out}\ n.P])\ .n^o[\,] & \\
\langle\!\langle P \mid Q \rangle\!\rangle & \stackrel{def}{=} & \langle\!\langle P \rangle\!\rangle \mid \langle\!\langle Q \rangle\!\rangle & \\
\langle\!\langle !P \rangle\!\rangle & \stackrel{def}{=} & !\langle\!\langle P \rangle\!\rangle & \\
\langle\!\langle \mathbf{0} \rangle\!\rangle & \stackrel{def}{=} & \mathbf{0} &
\end{array}
$$

FIG. 1.7: *Encoding of the synchronous/asynchrounous π-calculus*

interfaces. La syntaxe des types est la suivante:

$$
\begin{array}{llcl}
\text{Signatures} & \Sigma & ::= & (\ \ell_i(\mathscr{V}_i)\ )^{i\in I} \\
\text{Ambients} & \mathscr{A} & ::= & \mathsf{Amb}[\Sigma] \\
\text{Capabilities} & \mathscr{C} & ::= & \mathsf{Cap}[\Sigma] \\
\text{Processes} & \mathscr{P} & ::= & \mathsf{Proc}[\Sigma] \\
\text{Values} & \mathscr{V} & ::= & \mathscr{A} \mid \mathscr{C} \\
\text{Types} & \mathscr{T} & ::= & X \mid \mathscr{A} \mid \mathscr{C} \mid \mathscr{P}
\end{array}
$$

Les *signatures* décrivent l'interface de l'ambient en listant ses méthodes avec le type des paramètres. Plus précisément $\mathsf{Amb}[\Sigma]$ est le type des ambients avec méthodes dans Σ; $\mathsf{Cap}[\Sigma]$ est le type des capabilités qui peuvent apparaître dans un ambient dont la signature contient au moins les méthodes de Σ; $\mathsf{Proc}[\Sigma]$ est le type d'un processus pouvant apparaître dans un ambient dont la signature contient au moins les méthodes de Σ. Les valeurs, c'est-à-dire les arguments des méthodes sont des identificateurs d'ambients ou bien de capabilités. Il y a aussi dans la syntaxe des types les variables de types. Leur introduction est nécessaire pour traiter le typage du paramètre *self* (la variable liée par ς) lors de l'ouverture d'un ambient. En fait à cause de la capabilité `open` la méthode d'un ambient a peut être réinstallée dans un autre ambient qui ouvre a et qui peut avoir une interface plus riche. Ainsi le type de self peut être dynamiquement changé. Pour préserver la correction du typage les corps des méthodes sont typés sous l'hypothèse que *self* est typé par la variable de type *MyType* de [Bru94], c'est-à-dire par une variable F-bornée qui représente le type de tous les ambients où une méthode peut être réinstallée par l'`open`. En particulier nous utilisons une forme restreinte de *matching* [Bru94], où la variable X représentant le type de *self* peut apparaître bornée par un type ambient seulement dans des environnements de types (c'est-à-dire, $X \lessdot\!\# \mathscr{A}$). En outre notez que notre syntaxe ne permet pas aux variables d'apparaître dans les signatures. En conséquence le système de types ne permet pas d'exprimer la *MyType-specialisation* des méthodes [Bru94, FHM94], une technique de typage qui permet une forme restreinte de spécialisation covariante des méthodes.

Syntaxe typée et règles de typage

La syntaxe type de MA^{++} est décrite en figure 1.8 tandis que la structure des contextes et des jugements est définie par les productions en figure 1.9 où $\mathscr{W}$ a valeurs dans l'ensemble $\{X, \mathscr{A}, \mathscr{C}\}$:

L'ensemble complet des règles de typage se trouve dans [BCC01d] et est résumé en figure 1.10.

Interfaces	I	$::=$	$\ell(\boldsymbol{x}) \triangleright \varsigma(z)P \mid \mathrm{I} :: \mathrm{I} \mid \varepsilon$
Processes	P	$::=$	$\mathbf{0} \mid P \vert P \mid a[\,\mathrm{I}\,;P\,] \mid (\boldsymbol{\nu} x{:}\,\mathscr{A})P \mid M.P$
Expressions	M	$::=$	$x \mid (M_1,\ldots,M_n) \mid M \;\mathsf{send}\; \ell\langle N\rangle \mid \mathtt{in}\; M \mid \mathtt{out}\; M \mid \mathtt{open}\; M$

FIG. 1.8: *Typed syntax for ambients*

Contexts	Γ	$::=$	$\varnothing \mid \Gamma, x : \mathscr{W} \mid \Gamma, X \lessdot\!\!\# \mathscr{A}$
Judgements	J	$::=$	$\Gamma \vdash M : \mathscr{W} \mid \Gamma \vdash X \lessdot\!\!\# \mathscr{A} \mid \Gamma \vdash P : \mathscr{P} \mid \Gamma \vdash \mathscr{T} \mid \Gamma \vdash \diamond$

FIG. 1.9: *Contexts and typing judgments*

Nous ne commentons ici que les règles principales.

$$\text{(OPEN)}\quad \frac{\Gamma \vdash a : \mathsf{Amb}[\Sigma]}{\Gamma \vdash \mathsf{open}\; a : \mathsf{Cap}[\Sigma]}$$

La règle (OPEN) pour l'ouverture d'un ambient demande la connaissance précise du type de l'ambient à ouvrir: ainsi le type de l'ambient doit être un type ambient, pas une variable de type. Une ouverture est légale si la signature de l'ambient ouvrant est égale à (en fait par sous-typage, elle contient) celle de l'ambient à ouvrir. Par conséquent l'ouverture d'un ambient peut mettre à jour seulement des méthodes déjà existantes dans l'ambient ouvrant et leur type d'origine doit être préservé.

$$\text{(MESSAGE)}\quad \frac{\Gamma \vdash a : \mathscr{W} \quad \Gamma \vdash \mathscr{W} \lessdot\!\!\# \mathsf{Amb}[\,\ell(\mathscr{V}')\,] \quad \Gamma \vdash M' : \mathscr{V}'}{\Gamma \vdash a \;\mathsf{send}\; \ell\langle M'\rangle : \mathsf{Cap}[\Sigma]}$$

Cette règle établit que l'invocation d'une méthode ℓ sur une expression (un nom ou une variable) a demande que le type de a *matche* un type ambient contenant la méthode ℓ. Ce qui signifie que le type de a peut être ou un type ambient "plus long" que $\mathsf{Amb}[\,\ell(\mathscr{V}')\,]$, ou bien une variable de type qui apparaît bornée dans le contexte Γ. Puisque nous utilisons la sémantique RMI, le corps de la méthode est exécuté dans l'ambient receveur du message. Ainsi il n'est pas nécessaire de vérifier son type, car cette vérification est faite dans la règle pour les ambients:

$$\text{(AMB)}\quad (\Sigma = (\,\ell_i(\mathscr{V}_i)\,)^{i\in I})\quad \frac{\Gamma \vdash a : \mathsf{Amb}[\Sigma] \quad \Gamma, Z \lessdot\!\!\# \mathsf{Amb}[\Sigma], z{:}\, Z, x_i{:}\, \mathscr{V}_i \vdash P_i : \mathsf{Proc}[\Sigma] \quad \Gamma \vdash P : \mathsf{Proc}[\Sigma]}{\Gamma \vdash a[(\ell_i(x_i) \triangleright \varsigma(z)P_i)^{\;i\in I}\,; P] : \mathsf{Proc}[\Sigma']}$$

Cette règle type les ambients d'un façon similaire à ce que l'on fait dans [AC96a] pour les objets: chaque méthode est typée sous les hypothèses que (i) le type de *self* matche le type de l'ambient qui l'enferme, (ii) les paramètres des méthodes ont le type déclaré et (iii) les corps des méthodes doivent être typables par un type compatible à celui de l'ambient en question (car par RMI ils seront exécutés à l'intérieur de celui ci). Il faut remarquer que l'utilisation du *matching* pour le type de *self* fait que les méthodes locales à a seront typables aussi à l'intérieur d'éventuels ambients qui ouvrent

$$\text{(Env-empty)}\quad \frac{}{\varnothing \vdash \diamond} \qquad \text{(Env-}x\text{)}\quad \frac{\Gamma \vdash \mathscr{W} \quad x \notin Dom(\Gamma)}{\Gamma, x : \mathscr{W} \vdash \diamond} \qquad \text{(Env-}X\text{)}\quad \frac{\Gamma \vdash \diamond \quad X \notin Dom(\Gamma)}{\Gamma, X \mathrel{\lhd\#} \mathscr{A} \vdash \diamond}$$

$$\text{(Type X)}\quad \frac{\Gamma, X \mathrel{\lhd\#} \mathscr{A}, \Gamma' \vdash \diamond}{\Gamma, X \mathrel{\lhd\#} \mathscr{A}, \Gamma' \vdash X} \qquad \text{(Type Amb)}\quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathsf{Amb}[\Sigma]} \qquad \text{(Type Cap)}\quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathsf{Cap}[\Sigma]} \qquad \text{(Type Proc)}\quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathsf{Proc}[\Sigma]}$$

$$\text{(Match X)}\quad \frac{\Gamma, X \mathrel{\lhd\#} \mathscr{A}, \Gamma' \vdash \diamond}{\Gamma, X \mathrel{\lhd\#} \mathscr{A}, \Gamma' \vdash X \mathrel{\lhd\#} \mathscr{A}} \qquad \text{(Match Amb)}\quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathsf{Amb}[(\ell_i(\mathscr{V}_i))^{i \in 1..n+k}] \mathrel{\lhd\#} \mathsf{Amb}[(\ell_i(\mathscr{V}_i))^{i \in 1..n}]}$$

$$\text{(Sub Cap)}\quad \frac{\Sigma \subseteq \Sigma'}{\mathsf{Cap}[\Sigma] \leq \mathsf{Cap}[\Sigma']} \qquad \text{(Sub Proc)}\quad \frac{\Sigma \subseteq \Sigma'}{\mathsf{Proc}[\Sigma] \leq \mathsf{Proc}[\Sigma']} \qquad \text{(Subsump Cap)}\quad \frac{\Gamma \vdash M : \mathscr{C} \quad \mathscr{C} \leq \mathscr{C}'}{\Gamma \vdash M : \mathscr{C}'} \qquad \text{(Subsump Proc)}\quad \frac{\Gamma \vdash P : \mathscr{P} \quad \mathscr{P} \leq \mathscr{P}'}{\Gamma \vdash P : \mathscr{P}'}$$

$$\text{(name/var)}\quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash x : \Gamma(x)} \qquad (\varepsilon)\quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \varepsilon : \mathsf{Cap}[\Sigma]} \qquad \text{(path)}\quad \frac{\Gamma \vdash M_1 : \mathsf{Cap}[\Sigma] \quad \Gamma \vdash M_2 : \mathsf{Cap}[\Sigma]}{\Gamma \vdash M_1.M_2 : \mathsf{Cap}[\Sigma]}$$

$$\text{(open)}\quad \frac{\Gamma \vdash M : \mathsf{Amb}[\Sigma]}{\Gamma \vdash \mathsf{open}\ M : \mathsf{Cap}[\Sigma]} \qquad \text{(inout)}\quad \frac{\Gamma \vdash M : \mathscr{W} \quad \Gamma \vdash \mathscr{W} \mathrel{\lhd\#} \mathsf{Amb}[\Sigma] \quad (M' \in \{\mathtt{in}\ M, \mathtt{out}\ M\})}{\Gamma \vdash M' : \mathsf{Cap}[\Sigma']}$$

$$\text{(Message)}\quad \frac{\Gamma \vdash M : \mathscr{W} \quad \Gamma \vdash \mathscr{W} \mathrel{\lhd\#} \mathsf{Amb}[\ \ell(\mathscr{V}')\ ] \quad \Gamma \vdash M' : \mathscr{V}'}{\Gamma \vdash M\ \mathsf{send}\ \ell\langle M' \rangle : \mathsf{Cap}[\Sigma]}$$

$$\text{(pref)}\quad \frac{\Gamma \vdash M : \mathsf{Cap}[\Sigma] \quad \Gamma \vdash P : \mathsf{Proc}[\Sigma]}{\Gamma \vdash M.P : \mathsf{Proc}[\Sigma]} \qquad \text{(par)}\quad \frac{\Gamma \vdash P : \mathsf{Proc}[\Sigma] \quad \Gamma \vdash Q : \mathsf{Proc}[\Sigma]}{\Gamma \vdash P \mid Q : \mathsf{Proc}[\Sigma]}$$

$$\text{(restr)}\quad \frac{\Gamma, x{:}\ \mathscr{A} \vdash P : \mathsf{Proc}[\Sigma]}{\Gamma \vdash (\boldsymbol{\nu} x{:}\ \mathscr{A})P : \mathsf{Proc}[\Sigma]} \qquad \text{(dead)}\quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0} : \mathsf{Proc}[\Sigma]}$$

$$\text{(Amb)}\quad (\Sigma = (\ \ell_i(\mathscr{V}_i)\ )^{i \in I}) \quad \frac{\Gamma \vdash M : \mathsf{Amb}[\Sigma] \quad \Gamma, Z \mathrel{\lhd\#} \mathsf{Amb}[\Sigma], z{:}\ Z, x_i{:}\ \mathscr{V}_i \vdash P_i : \mathsf{Proc}[\Sigma] \quad \Gamma \vdash P : \mathsf{Proc}[\Sigma]}{\Gamma \vdash M[(\ell_i(x_i) \triangleright \varsigma(z)P_i)\ ^{i \in I}\ ; P] : \mathsf{Proc}[\Sigma']}$$

FIG. 1.10: *Typing rules for MA*$^{++}$

a (cf. regle `open`)

(MATCH AMB)

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathsf{Amb}[(\ell_i(\mathscr{V}_i))^{i\in 1..n+k}] \mathrel{<\!\!\#} \mathsf{Amb}[(\ell_i(\mathscr{V}_i))^{i\in 1..n}]}$$

(SUB CAP)

$$\frac{\Sigma \subseteq \Sigma'}{\mathsf{Cap}[\Sigma] \leq \mathsf{Cap}[\Sigma']}$$

(SUB PROC)

$$\frac{\Sigma \subseteq \Sigma'}{\mathsf{Proc}[\Sigma] \leq \mathsf{Proc}[\Sigma']}$$

Une relation de sous-typage est définie pour les types des capabilités et des processus: un type $\mathsf{Cap}[\Sigma]$ (respectivement, $\mathsf{Proc}[\Sigma]$) est le sous-type de tout type de capabilité (respectivement, de processus) dont la signature contient (de façon ensembliste) Σ. La relation résultant rappelle fortement le *width*-subtyping de certains systèmes de types pour les objets. Toutefois le *width*-subtyping ne doit pas être utilisé avec les types ambients car il permettrait une utilisation incorrecte de la capabilité `open`: intuitivement lorsqu'on ouvre un ambient, l'exacte connaissance de sa signature est nécessaire pour assurer que toute mise-à-jour des méthodes est reflétée dans les types. Si nous savions seulement que la signature d'un ambient à ouvrir est contenue dans une signature Σ nous ne pourrions pas vérifier que la mise-à-jour d'une méthode présente dans l'ambient mais non tracée par Σ est bien typée. Cependant grâce au *matching* nous avons un typage correct quand un ambient est ouvert par un ambient "plus grand".

Les règles restantes sont assez standard. Le système complet, présenté en figure 1.10, satisfait les propriétés fondamentales parmi lesquelles la subject reduction

Théorème 1.2.1 (Subject Reduction)
If $\Gamma \vdash P : \mathsf{Proc}[\Sigma]$ *and* $P \rightarrow Q$ *then* $\Gamma \vdash Q : \mathsf{Proc}[\Sigma]$.

Chapitre 2

Analyses de Sécurité

Articles de référence: [BC01, BC02]

La récente explosion de l'utilisation d'Internet et l'apparition conjointe de la mobilité de systèmes à la fois software et hardware, posent de nouveaux enjeux tant sur le plan de la flexibilité des outils conçus pour "programmer" Internet que, plus encore, sur le plan de la sécurité du réseau.

Selon des études récentes (source IGI Consulting) pour l'année 2003 il y aura 300 millions de terminaux connectés à Internet par voie hertzienne: le début d'une énorme infrastructure de computation globale. En fait, la configuration vers laquelle Internet tend de plus en plus à évoluer est celle d'une énorme machine concurrente partitionnée en systèmes emboîtés (firewalls, LAN's, workstations, applets) et dont la topologie se modifie en permanence.

Il se pose donc le problème de l'étude d'un formalisme pour la programmation de cette machine qu'est Internet et de propriétés des données traitées.

Les données, en tant que ressources sensibles doivent être *sûres* et *sécurisées* : sûres dans le sens où elles doivent posséder certaines propriétés qui en assurent la correction et la fiabilité; sécurisées dans le sens où elles doivent être gérées de manière à empêcher toute utilisation malveillante ou non autorisée.

Les travaux détaillés dans les précédentes parties de cette thèse sont centrés autour du typage. Le chapitre précédent étend cette étude aux cas de mobilité, mais encore une fois les propriétés étudiées ne se limitent qu'au typage. Or le typage, quoique ingrédient indispensable à la production de logiciel de qualité, ne couvre que l'aspect sûreté, et touche seulement de manière marginale la sécurité. En outre, la dissémination croissante des données ainsi que l'expansion d'Internet posent de nouveaux enjeux en matière de sécurité, tout spécialement en présence de programmes mobiles.

Dans ce chapitre nous décrivons donc une analyse statique de propriétés de sécurité des ambients. Pour cela nous nous concentrons sur le coeur de mobilité des ambients. Donc nous ne considérons ni les envois de message introduits dans le chapitre précédent ni les primitives de communications des ambients. Le traitement de ces dernières ne pose pas trop de difficultés et il est détaillé dans [BC02].

Quoique les techniques développées dans ce chapitre reposent essentiellement sur le typage, les types que nous allons introduire ne servent pas à sélectionner les termes qui ne causent pas d'erreurs à l'exécution mais plutôt à décrire le comportement de tout terme. En fait dans le système que nous allons présenter tout terme est typable.

Ces types seront ensuite utilisés pour vérifier que le comportement d'un terme, décrit par le type, ne viole pas une certaine politique de sécurité.

Pour que nos types décrivent le comportement des termes de manière la plus précise possible nous considérons une variante des Ambients Mobiles appelée *Ambients Sûrs* (Safe Ambients [LS00]). Nous avons vu dans le chapitre précédent que les Ambients Mobiles sont des agents nommés (ou

locations) qui enferment des ensembles de processus et qu'ils peuvent être emboîtés. Les ambients sûrs diffèrent des ambients mobiles par la notion d'interaction sous-jacente: dans les seconds l'interaction est "unilatérale", dans le sens que lors d'une action de mobilité un seul des deux partenaires de l'interaction exécute l'action. Par contre dans les ambients sûrs la réduction demande que les actions se synchronisent avec des "co-actions" correspondantes. Par exemple, reprenons l'exemple de la section 1.2.1 du chapitre précédent:

Mobile Ambients $a[\mathtt{open}\ b.\mathtt{in}\ c] \mid b[\mathtt{in}\ a.\mathtt{in}\ d]$.

dans cette configuration l'ambient b peut entrer dans l'ambient a en exerçant la *capabilit é* $\mathtt{in}\ a$ et ainsi passer dans la configuration $a[\mathtt{open}\ b.\mathtt{in}\ c \mid b[\mathtt{in}\ d]]$. Ensuite a peut dissoudre la clôture fournie par b en exerçant $\mathtt{open}\ b$, et réduire à $a[\mathtt{in}\ c \mid \mathtt{in}\ d]$.

Aucune des deux réductions n'est légale dans les Ambients Sûrs. Pour obtenir le comportement que nous venons de décrire, les deux ambients a et b doivent être ainsi définis

Safe Ambients $a[\mathsf{coin}\ a.\mathtt{open}\ b.\mathtt{in}\ c] \mid b[\mathtt{in}\ a.\mathsf{coopen}\ b.\mathtt{in}\ d]$.

Maintenant le mouvement de b dans a dérive d'un accord entre les deux partenaires: b exerçant la capabilité $\mathtt{in}\ a$ et a exerçant la *co-capabilité* $\mathsf{coin}\ a$. La configuration résultante, c'est-à-dire $a[\mathtt{open}\ b.\mathtt{in}\ c \mid b[\mathsf{coopen}\ b.\mathtt{in}\ d]]$, se réduit à $a[\mathtt{in}\ c \mid \mathtt{in}\ d]$, une fois de plus comme résultat d'une synchronisation entre $\mathtt{open}\ b$ et $\mathsf{coopen}\ b$.

Ici nous décrivons une variante des Ambients Sûrs que nous appelons *Ambients S écurisés* (Secure Safe Ambients) que nous avons définis dans [BC01] et dont la caractéristique distinctive est celle de posséder un système de types pour exprimer (et ensuite vérifier) les invariants comportementaux des ambients. L'aspect le plus significatif est que le système de types est capable de garder trace tant du comportement implicite des ambients que de leur comportement explicite: le type d'un ambient prend en compte non seulement les capabilités que l'ambient isolé peut exercer, mais aussi le comportement résultant de capabilités que l'ambient peut acquérir en interagissant avec l'environnement. Un tel degré de précision est nécessaire pour une vérification correcte des politiques de sécurité car la mobilité implicite (c'est-à-dire acquise) est à l'origine de nombreux types d'attaques tels que les *Chevaux de Troie* ou autres combinaisons d'agents malicieux.

Exemple 2.0.2 Considérons à nouveau les ambients (sécurisés) a et b introduits ci-dessus et plaçons-les en parallèle avec un troisième ambient c:

$$a[\mathsf{coin}\ a.\mathtt{open}\ b.\mathtt{in}\ c] \mid b[\mathtt{in}\ a.\mathsf{coopen}\ b.\mathtt{in}\ d] \mid c[\ \mathsf{coin}\ c.P \mid d[\mathsf{coin}\ d.Q]\ ]$$

Supposons que d contienne des données confidentielles qui doivent être accessibles à tout ambient qui s'exécute *dans* c (ce qui est permis par la capabilité $\mathsf{coin}\ d$) mais pas aux ambients *entrants* dans c.

Une fois cette politique de sécurité établie la question est de savoir si c doit ou pas laisser a rentrer sans crainte qu'il puisse accéder aux données de d. Si l'on ne regarde que la mobilité *explicite* de a, c'est-à-dire les capabilités qui sont à sa disposition, alors le mouvement de a dans c semble inoffensif car a n'essaie pas d'entrer dans d. Toutefois a peut acquérir dans un deuxième temps la capabilité pour accéder d: a peut laisser b rentrer, rentrer ensuite dans c et, une fois dans c, ouvrir b pour pourvoir accéder à d. □

Exemple 2.0.3 Une manière différente pour a d'attaquer c est de faire sortir b après être entré dans c:

$$a[\mathsf{coin}\ a.\mathtt{in}\ c.\mathsf{coout}\ a] \mid b[\mathtt{in}\ a.\mathtt{out}\ a.\mathtt{in}\ d] \mid c[\ \mathsf{coin}\ c.P \mid d[\mathsf{coin}\ d.Q]\ ]$$

de cette manière a opérerait comme un cheval de Troie. □

2.1 Aperçu

Le système de types défini dans [BC01] et décrit dans ce chapitre permet la définition formelle de politiques de sécurité pour les ambients ainsi que la détection statique de toute tentative de violation d'une politique. En particulier il détecte les attaques basées sur l'acquisition implicite de capabilités telles que nous les avons présentées dans les exemples précédents.

Trois ingrédients sont essentiels dans ce système de types:

1. Les ambients sont classifiés par des *domaines*: chaque domaine est associé à un *comportement* que les ambients du domaine partagent et doivent respecter, et à une *politique de sécurité* qui protège les ambients du domaine des interactions avec l'environnement non désirées.
2. Les *types processus* décrivent le comportement d'un processus en utilisant les domaines comme unité d'abstraction car ils sont formés par des ensembles de *capabilités sur les types*. Par exemple si a est un ambient de domaine A et P un processus qui peut déclencher la capabilité $\texttt{in}\ a$, alors le type de P trace ce comportement en incluant la capabilité $\texttt{in}\ A$.
3. Enfin chaque domaine est associé à un ensemble de *contraintes de sécurité* qui en définit la politique de sécurité: les contraintes établissent les droits d'accès de tout ambient traversant la frontière d'un ambient du domaine.

Le système de types que nous allons esquisser satisfait la propriété de *subject reduction* ainsi qu'une propriété de sécurité très forte qui montre que les types fournissent une approximation complète du comportement: précisément, nous montrons que si un processus P évoluant dans un contexte bien typé $\mathscr{C}$ exerce une capabilité, alors cette capabilité est tracée dans le type statique de P.

Enfin nous allons étudier une version distribuée du système et établir le résultat de sécurité ci-dessus aussi dans le cas où $\mathscr{C}$ n'est pas bien typé, c'est-à-dire dans le cas où P évolue dans le cadre d'un environnement potentiellement hostile. Pour cela un certain nombre de contrôles dynamiques se révéleront nécessaires et l'examen de ces vérifications fait ressortir un trait aussi intéressant qu'inattendu: en regardant les contrôles dynamiques nécessaires à la version distribuée on retrouve exactement les vérifications effectuées par les trois composantes de l'architecture de sécurité de la Java Virtual Machine: le *Class Loader*, le *Bytecode Verifier* et le *Security Manager*; ce qui à notre connaissance constitue la première justification formelle d'une telle architecture.

2.2 Syntaxe

Les termes de notre langage sont ceux des *Ambients Sûrs* à la différence près que les types sont des *domaines*:

$$\begin{array}{llll} \textit{Processes} & P & ::= & \mathbf{0} \mid \alpha.P \mid (\boldsymbol{\nu} a{:}\, D)P \mid P \mid P \mid a[P] \mid\ !P \\ \textit{Capabilities} & \alpha & ::= & \texttt{in}\ a \mid \mathsf{coin}\ a \mid \texttt{out}\ a \mid \mathsf{coout}\ a \mid \texttt{open}\ a \mid \mathsf{coopen}\ a \end{array}$$

Il en va de même pour la sémantique:

$$\begin{array}{ll} \textbf{(in)} & b[\texttt{in}\ a.P \mid Q] \mid a[\mathsf{coin}\ a.R \mid S] \rightarrow a[R \mid S \mid b[P \mid Q]] \\ \textbf{(out)} & a[\, b[\texttt{out}\, a.P \mid Q] \mid \mathsf{coout}\, a.R \mid S] \rightarrow b[P \mid Q] \mid a[R \mid S] \\ \textbf{(open)} & \texttt{open}\ a.P \mid a[\mathsf{coopen}\ a.Q \mid R] \rightarrow P \mid Q \mid R \\ \textbf{(context)} & P \rightarrow Q \ \Rightarrow\ \mathscr{E}[P] \rightarrow \mathscr{E}[Q] \\ \textbf{(struct)} & P' \equiv P \rightarrow Q \ \Rightarrow\ P' \rightarrow Q \end{array}$$

où $\mathscr{E}[\,]$ dénotes les contextes d'évaluation:

Evaluation Contexts

$$\mathscr{E}[\,] \quad ::= \quad [\,] \ \mid \ (\boldsymbol{\nu} a{:}\ D)\mathscr{E}[\,] \ \mid \ P \mid \mathscr{E}[\,] \ \mid \ \mathscr{E}[\,] \mid P \ \mid \ a[\mathscr{E}[\,]]$$

Et $\equiv$ est la plus petite congruence qui soit un monoïde commutatif pour **0** et $\mid$ et satisfait:

$$\begin{array}{ll} !P \equiv !P \mid P & \\ (\boldsymbol{\nu} a{:}\ D)\mathbf{0} \equiv \mathbf{0} & \\ (\boldsymbol{\nu} a{:}\ A)(\boldsymbol{\nu} b{:}\ B)P \equiv (\boldsymbol{\nu} b{:}\ B)(\boldsymbol{\nu} a{:}\ A)P & \text{for } a \neq b \\ (\boldsymbol{\nu} a{:}\ D)(P \mid Q) \equiv P \mid (\boldsymbol{\nu} a{:}\ D)Q & \text{for } a \notin fn(P) \\ (\boldsymbol{\nu} a{:}\ D)b[P\,] \equiv b[(\boldsymbol{\nu} a{:}\ D)P\,] & \text{for } a \neq b \end{array}$$

2.2.1 Types

Les domaines, dénotés par A, B, C et D, constituent l'unité d'abstraction au niveau des types: dans le système de types l'effet d'une capabilité est observé sur les domaines plutôt que sur les (noms des) ambients

Type Capabilities

$$M ::= \texttt{in}\ D \mid \mathsf{coin}\ D \mid \texttt{out}\ D \mid \mathsf{coout}\ D \mid \texttt{open}\ D \mid \mathsf{coopen}\ D$$

Process Types

$$\mathsf{P} ::= (\mathbf{L}, \mathbf{M}, \mathbf{N}) \qquad\qquad (\mathbf{L}, \mathbf{M}, \mathbf{N} \in 2^{M})$$

Notation Si $\mathsf{P} = (\mathbf{L}, \mathbf{M}, \mathbf{N})$, nous écrivons $\mathsf{P}^{\uparrow}$ pour $\mathbf{L}$, $\mathsf{P}^{=}$ pour $\mathbf{M}$ et $\mathsf{P}^{\downarrow}$ pour $\mathbf{N}$; si P et Q sont des types processus alors $\mathsf{P} \subseteq \mathsf{Q}$ dénote l'inclusion par composantes. De même $\mathsf{P} \cup \mathsf{Q}$ dénote l'union par composantes. Soit $\mathbf{M}$ un ensemble de capabilités de type, et P un type processus nous écrivons $\mathsf{P} \cup^{\downarrow} \mathbf{M}$ (respectivement, à $\mathsf{P} \cup^{=} \mathbf{M}$ et à $\mathsf{P} \cup^{\uparrow} \mathbf{M}$) pour dénoter le processus résultant de l'union de $\mathbf{M}$ à $\mathsf{P}^{\downarrow}$ (respectivement, $\mathsf{P}^{=}$ et $\mathsf{P}^{\uparrow}$). Enfin, soit M une capabilité de type, et $\overline{M}$ sa co-capabilité, nous écrirons $M \in \mathsf{sync}(\mathbf{L}, \mathbf{M})$ pour $M \in \mathbf{L}$ et $\overline{M} \in \mathbf{M}$. $\square$.

Les types processus sont des triplets d'ensembles de capabilités de types. Ils décrivent les capabilités qu'un processus peut exercer et tracent le *niveau* auquel l'effet d'une capabilité peut être observé. Les trois composants des types représentent les différents niveaux: si $P : \mathsf{P}$ alors $\mathsf{P}^{\uparrow}$ décrit les effets observables au niveau de l'ambient qui renferme P, $\mathsf{P}^{=}$ décrit les effets observables au niveau de P et, enfin, $\mathsf{P}^{\downarrow}$ décrit les capabilités exercées *dans* P, chaque fois que P est de la forme $a[P'\,]$. Par exemple si $a : A$ alors:

- $\texttt{in}\ a.P : \mathsf{P} \Rightarrow \texttt{in}\ A \in \mathsf{P}^{\uparrow}$, car l'effet d'exercer $\texttt{in}\ a$ est observé au niveau de l'éventuel ambient renfermant P;
- $b[\texttt{in}\ a.P\,] : \mathsf{P} \Rightarrow \texttt{in}\ A \in \mathsf{P}^{=}$, car maintenant c'est $b[\texttt{in}\ a.P\,]$ qui exerce $\texttt{in}\ a$;
- $\texttt{open}\ a.P : \mathsf{P} \Rightarrow \texttt{open}\ A \in \mathsf{P}^{=}$, car $\texttt{open}\ a$ est exercé au niveau du processus en parallèle avec $\texttt{open}\ a.P$
- $b[\texttt{open}\ a.P\,] : \mathsf{P} \Rightarrow \texttt{open}\ A \in \mathsf{P}^{\downarrow}$, car $\texttt{open}\ a$ est exercé dans b.

2.2.2 Environnements et typage

Nous avons besoin de deux environnements différents, les *Environnements de Type* dénotés par E et les *Environnement de Domaine* dénotés par Π:

Type Envirnoments	E	:	*Ambient Names* $\rightarrow$ *Ambient Domains*
Domain Envirnoments	Π	:	*Ambient Domains* $\rightarrow$ *Process Types*

Les premiers associent à chaque (nom d') ambient le domaine auquel il appartient; les deuxième associent à chaque domaine le type partagé par tous ses ambients. Ainsi tandis que les environnements de type partitionnent les ambients dans les domaines, les environnements de domaine décrivent les interactions potentielles entre les domaines et imposent des invariants comportementaux pour les processus enfermés dans des ambients d'un domaine.

Définition 2.2.1 [Closure and Boundedness] Soit Π un environnement de domaine, P un type processus et D et H des domaines. Nous introduisons les notations suivantes:

$$\Pi \vdash \mathsf{P}\ \mathsf{closed} \stackrel{def}{=} \quad \mathtt{open}\ H \in \mathsf{sync}(\mathsf{P}^{=}, \Pi(H)^{=}) \Rightarrow \Pi(H) \subseteq \mathsf{P}$$

$$\Pi \vdash D\ \mathsf{bounds}\ \mathsf{P} \stackrel{def}{=} \quad \mathsf{P}^{\uparrow} \subseteq \Pi(D)^{=}\ \wedge\ \mathsf{P}^{=} \subseteq \Pi(D)^{\downarrow}\ \wedge\ (\mathsf{coopen}\ D \in \Pi(D)^{=} \Rightarrow \mathsf{P} \subseteq \Pi(D))$$

$$\Pi \vdash D\ \mathsf{closed} \stackrel{def}{=} \quad \begin{cases} \mathtt{in}\ H \in \mathsf{sync}(\Pi(D)^{=}, \Pi(H)^{=}) \Rightarrow \Pi \vdash H\ \mathsf{bounds}\ \Pi(D) \\ \mathtt{out}\ H \in \mathsf{sync}(\Pi(D)^{=}, \Pi(H)^{\downarrow}) \Rightarrow \Pi(D) \subseteq \Pi(H) \end{cases}$$

□

La condition de clôture sur les types processus formalise l'intuition qu'un processus peut exercer toutes les capabilités des ambients qu'il peut ouvrir. La *boundedness* de P par D assure que le type processus $\Pi(D)$ constitue une approximation correcte du type P d'un quelconque processus enfermé dans des ambients de domaine D: ceci est exprimé par les deux premières inclusions qui mettent en relation les différents niveaux de l'ambient et du processus qu'il contient; la dernière inclusion prend en compte la possibilité qu'un ambient puisse être ouvert et que donc toutes les capabilités du processus enfermé montent d'un niveau. Enfin la clôture d'un domaine assure les invariants précédents en présence de mobilité: le comportement d'un ambient a de domaine D doit prendre en compte le comportement de tout ambient qui entre dans a ainsi que le comportement de tout ambient qui sort de a (car si a laisse ces ambients sortir il est virtuellement responsable de leur comportement).

Définition 2.2.2 [Coherence] Soit Π un environnement de domaine. La notation $\Pi \vdash \diamond$ (lire Π est *cohérent*) est définie comme suit:

$$\Pi \vdash \diamond \stackrel{def}{=} fn(\Pi) \subseteq \mathsf{Dom}(\Pi)\ \wedge\ \forall D \in \mathsf{Dom}(\Pi).\ (\Pi \vdash D\ \mathsf{closed} \wedge \Pi \vdash \Pi(D)\ \mathsf{closed})$$

□

Les règles de typage se trouvent en figure 2.1 et servent à dériver des jugements de la forme $\Pi, E \vdash P\colon \mathsf{P}$ où E est un environnement de type, Π un environnement de domaine et $\mathsf{Img}(E) \subseteq \mathsf{Dom}(\Pi)$ (c'est-à-dire l'*image* de E est contenue dans le *domaine* de Π). Les règles (DEAD), (PAR), (REPL), (RESTR) sont standard. Le typage des préfixes dans les règles (ACTION) a été motivé auparavant. Enfin la règle (AMB) indique qu'un ambient $a[P]$ possède le type que Π associe au domaine de a pourvu que D “bounds” le type de P dans Π. Cette règle est techniquement intéressante car contrairement aux règles correspondantes dans les systèmes de type précédents pour les Ambients [CG99b, CGG99] elle établit une relation significative entre le type d'un ambient et celui du processus qu'il enferme, ce qui est essentiel pour tracer le comportement implicite du système.

Théorème 2.2.3 (Subject Reduction) *Si* $\Pi, E \vdash P : \mathsf{P}$ *et* $P \rightarrow Q$*, alors* $\Pi, E \vdash Q : \mathsf{P}$.

2.2.3 Exemples

Montrons le comportement du typage en considérant les exemples 2.0.2 et 2.0.3. Soient $E \equiv a\colon A, b\colon B, c\colon C, d\colon D$, et considérons l'attaque

$$a[\mathsf{coin}\ a.\mathtt{open}\ b.\mathtt{in}\ c] \mid b[\mathtt{in}\ a.\mathsf{coopen}\ b.\mathtt{in}\ d].$$

$$\frac{\Pi \vdash \diamond \quad fn(\mathsf{P}) \subseteq \mathsf{Dom}(\Pi) \quad \Pi \vdash \mathsf{P}\ \mathsf{closed}}{\Pi \vdash \mathsf{P}}\ (\text{TYPE PROC}) \qquad \frac{\Pi \vdash \diamond \quad \mathsf{Img}(E) \subseteq \mathsf{Dom}(\Pi)}{\Pi, E \vdash \diamond}\ (\text{ENV}) \qquad \frac{\Pi, E \vdash \diamond \quad a \in \mathsf{Dom}(E)}{\Pi, E \vdash a : E(a)}\ (\text{NAME})$$

$$\frac{\Pi, E \vdash \diamond}{\Pi, E \vdash \mathbf{0} : (\varnothing, \varnothing, \varnothing)}\ (\text{DEAD}) \qquad \frac{\Pi, E \vdash P : \mathsf{P} \quad \Pi, E \vdash Q : \mathsf{P}}{\Pi, E \vdash P \mid Q : \mathsf{P}}\ (\text{PAR}) \qquad \frac{\Pi, E \vdash P : \mathsf{P}}{\Pi, E \vdash !P : \mathsf{P}}\ (\text{REPL})$$

$$\frac{\Pi \vdash P \colon \mathsf{P} \quad \Pi, E \vdash a \colon D \quad \mathsf{cap}\ D \in \mathsf{P}^{\uparrow}}{\Pi, E \vdash \mathsf{cap}\ a.P : \mathsf{P}}\ (\text{ACTION}^{\uparrow}) \quad \mathsf{cap} \in \{\mathtt{in}, \mathsf{coin}, \mathtt{out}, \mathsf{coopen}\}$$

$$\frac{\Pi, E \vdash P \colon \mathsf{P} \quad \Pi, E \vdash a \colon D \quad \mathsf{cap}\ D \in \mathsf{P}^{=}}{\Pi, E \vdash \mathsf{cap}\ a.P : \mathsf{P}}\ (\text{ACTION}^{=}) \quad \mathsf{cap} \in \{\mathsf{coout}, \mathtt{open}\}$$

$$\frac{\Pi, E, a : D \vdash P : \mathsf{P} \quad a \notin \mathsf{Dom}(E)}{\Pi, E \vdash (\boldsymbol{\nu} a \colon D)P : \mathsf{P}}\ (\text{RESTR}) \qquad \frac{\Pi, E \vdash P : \mathsf{P} \quad \Pi, E \vdash a : D \quad \Pi \vdash D\ \mathsf{bounds}\ \mathsf{P}}{\Pi, E \vdash a[P] : \Pi(D)}\ (\text{AMB})$$

$$\frac{\Pi, E \vdash P : \mathsf{P} \qquad \Pi \vdash \mathsf{Q} \qquad \mathsf{P} \subseteq \mathsf{Q}}{\Pi, E \vdash P : \mathsf{Q}}\ (\text{SUBSUMPTION})$$

FIG. 2.1: *Typing Rules*

Soit P_b le type du processus enfermé dans b: il est facile de vérifier que $\{\mathsf{coopen}\ B, \mathtt{in}\ D\} \subseteq \mathsf{P}_b^{\uparrow}$. De $\Pi \vdash B$ bounds P_b, on obtient $\mathsf{coopen}\ B \in \Pi(B)^{=}$, et donc $\mathtt{in}\ D \in \Pi(B)^{\uparrow}$. Soit P_a le type du processus enfermé dans a. Puisque $\mathtt{open}\ B \in \mathsf{sync}(\mathsf{P}_a, \Pi(B)^{=})$, alors par la clôture de P_a nous avons $\Pi(B)^{\uparrow} \subseteq \mathsf{P}_a^{\uparrow} \subseteq \Pi(A)^{=}$ (la dernière inclusion suit de $\Pi \vdash A$ bounds P_a). Ainsi $\mathtt{in}\ D \in \Pi(A)^{=}$ et l'attaque est détectée.

Une analyse similaire s'applique à l'attaque

$$a[\mathsf{coin}\ a.\mathtt{in}\ c.\mathsf{coout}\ a] \mid b[\mathtt{in}\ a.\mathtt{out}\ a.\mathtt{in}\ d].$$

Ici $\mathtt{in}\ D \in \Pi(A)^{=}$ dérive de $\mathtt{out}\ A \in \mathsf{sync}(\Pi(B)^{=}, \Pi(A)^{\downarrow})$, qui par clôture implique $\Pi(B) \subseteq \Pi(A)$.

2.2.4 Sûreté

La signification opérationnelle du système de types est établie en montrant que les types processus constituent une approximation sûre du comportement des processus qu'ils typent. Pour cela il faut introduire un relation $P \Downarrow \alpha^{\eta}$ qui définit le comportement d'un processus P en termes des capabilités α que P peut exercer (au niveau $\eta \in \{\uparrow, =, \downarrow\}$) dans un contexte. Après on relie le système de types avec cette notion de comportement par un résultat de sûreté qui établit que pour tout processus bien typé P s'exécutant dans un contexte bien typé, si $P \Downarrow \alpha^{\eta}$ alors α est tracé dans la composante η du type de P. Plus précisément nous avons que:

Théorème 2.2.4 (Safety) *Pour tout processus* P *et contexte* $\mathscr{C}$ *si* $\Pi, E \vdash P : \mathsf{P}$ *et* $\Pi, E \vdash \mathscr{C}[P] : \mathsf{P}'$ *alors* $P \Downarrow (\mathsf{cap}\ a)^\eta$ *implique* $\mathsf{cap}\ E(a) \in \mathsf{P}^\eta$.

Pour définir et démontrer cette propriété il faut introduire la notion de *résidu* d'un ambient, notion fort intéressante mais que nous n'avons pas l'espace de définir précisément. Le lecteur intéressé est invité à consulter [BC01] pour les détails. Dans le même article il trouvera la description d'un algorithme de *reconstruction de types* qui pour tout processus P et tout environnement de type E tels que $fn(P) \subseteq \mathsf{Dom}(E)$ retourne les plus petits type processus P et environnement Π tel que $\Pi, E \vdash P : \mathsf{P}$. Si nous dénotons les types et l'environnement retournés par l'algorithme respectivement par $\mathscr{R}_{\mathsf{type}}(E, P)$ et $\mathscr{R}_{\mathsf{env}}(E, P)$ alors nous pouvons démontrer la propriété suivante:

Corollaire 2.2.5 (Minimalité) *Soit* P *un processus et* E *un environnement de type tel que* $fn(P) \subseteq \mathsf{Dom}(E)$. *Alors*

$$(\mathscr{R}_{env}(E, P), \mathscr{R}_{type}(E, P)) = \min\{(\Pi, \mathsf{P}) \mid \Pi, E \vdash P : \mathsf{P}\}$$

L'importance pratique d'un tel algorithme est discutée après l'introduction des politique de sécurité.

2.3 Politiques de sécurité

Les politiques de sécurité sont exprimées par le biais d'*environnements de sécurité* qui associent chaque domaine à un ensemble de *contraintes de sécurité*:

$$\textit{Security Envs} \quad \Sigma : \textit{Ambient Domains} \rightarrow \textit{Security Constraints}$$

Un environnement de sécurité établit la structure de sécurité pour un certain système d'ambients. Pour un Π et E donnés nous pouvons vérifier si un processus bien typé P est sûr dans Σ en vérifiant si Π satisfait Σ (noté $\Pi \models \Sigma$). La notion de satisfaction demande que $\mathsf{Dom}(\Sigma) = \mathsf{Dom}(\Pi)$. Sa définition dépend des contraintes de sécurité, dont la structure à leur tour dépend du type de politique de sécurité que nous voulons exprimer. Nous discutons deux choix différents de contraintes de sécurité:

Les **Contraintes de domaine** induisent des politiques de sécurités assez manichéennes où l'on partitionne les domaines en *fiables* et *non-fiables* et où les interactions sont permises seulement avec les premiers. Ces contraintes de sécurité peuvent être exprimées par des tables de la forme $\mathsf{S} = \langle \mathsf{in} = \mathscr{D}_{\mathsf{in}},\ \mathsf{out} = \mathscr{D}_{\mathsf{out}} \rangle$ où $\mathscr{D}_{\mathsf{in}}$ et $\mathscr{D}_{\mathsf{out}}$ sont des ensembles de domaines. Si D est un domaine et $\Sigma(D) = \mathsf{S}$ alors un ambient peut rentrer dans (respectivement, sortir de) un ambient de D seulement si son domaine appartient $\mathscr{D}_{\mathsf{in}}$ (respectivement, à $\mathscr{D}_{\mathsf{out}}$). Dans cette option $\Pi \models \Sigma$ si et seulement si, pour tout D dans $\mathsf{Dom}(\Pi)$, on a

(i) $\{A \mid \mathtt{in}\ D \in \mathsf{sync}(\Pi(A)^=, \Pi(D)^=)\} \subseteq \Sigma(D).\mathsf{in}$

(ii) $\{A \mid \mathtt{out}\ D \in \mathsf{sync}(\Pi(A)^=, \Pi(D)^\downarrow)\} \subseteq \Sigma(D).\mathsf{out}$.

Le modèle de sécurité induit par l'utilisation de contraintes de domaine est très proche de la politique de sécurité typique de JDK 1.1.x. Dans JDK 1.0.x toute définition non locale est considérée non-fiable. La même discipline s'applique sous JDK 1.1.x à la différence près qu'une classe déchargée du réseau peut devenir fiable si elle est signée électroniquement par un tiers de confiance (dans notre cas par un domaine de $\mathscr{D}_{\mathsf{in}}$).

Les **Contraintes de Capabilité** autorisent des politiques de sécurité plus fines car elles permettent de définir l'ensemble des capabilités de type que les ambients entrants et sortants peuvent exercer. Autrement dit, elle permettent de circonscrire le comportement des ambients en transit de manière assez précise. Ces types de contraintes peuvent être exprimées par des tables de la forme $\mathsf{S} = \langle \mathsf{in} = \mathsf{P}_{\mathsf{in}},\ \mathsf{out} = \mathsf{P}_{\mathsf{out}} \rangle$, dont les entrées sont des types processus. Si D est un domaine et $\Sigma(D) = \langle \mathsf{in} = \mathsf{P}_{\mathsf{in}},\ \mathsf{out} = \mathsf{P}_{\mathsf{out}} \rangle$, alors:

- P_{in} définit les seules capabilités que des processus entrants dans des ambients du domaine D peuvent exercer: les trois ensembles $\mathsf{P}_{\mathsf{in}}^\downarrow$, $\mathsf{P}_{\mathsf{in}}^=$, et $\mathsf{P}_{\mathsf{in}}^\uparrow$ spécifient les capabilités qui peuvent être

exercées au niveau du l'ambient entrant, du processus enfermé et dans l'ambient entrant respectivement. La première spécification est utile pour prévenir de fuites d'information, la seconde pour contrôler les interaction locales de l'ambient entrant et la troisième est utile si l'on veut ouvrir (ou entrer) l'ambient entrant.

– $\mathsf{P}_{\mathsf{out}}$ est le triplet définissant les capabilités octroyées aux processus sortants des ambients de domaine D, où les trois entrées $\mathsf{P}^{\uparrow}_{\mathsf{out}}$, $\mathsf{P}^{=}_{\mathsf{out}}$, et $\mathsf{P}^{\downarrow}_{\mathsf{out}}$ sont définies comme ci-dessus.

Dans cette option $\Pi \models \Sigma$ si et seulement si, pour tout A, B dans $\mathsf{Dom}(\Pi)$, $\mathtt{in}\ A \in \mathsf{sync}(\Pi(B)^{=}, \Pi(A)^{=})$ implique $\Pi(B) \subseteq \Sigma(A).\mathsf{in}$, et $\mathtt{out}\ A \in \mathsf{sync}(\Pi(B)^{=}, \Pi(A)^{\downarrow})$ implique $\Pi(B) \subseteq \Sigma(A).\mathsf{out}$. Les contraintes de capabilité sont assez proches des *permission collections* utilisées dans l'architecture de JDK 1.2 (alias Java 2) pour imposer des politiques de sécurité basées sur le contrôle des accès et l'inspection de la pile des appels de méthode.

Dans [BC01] on trouvera une définition de contraintes de sécurité encore plus générales que les précédentes basée sur l'utilisation de formules de la logique du premier ordre.

Indépendemment des contraintes prises en considération pour un processus P et un environnement de type E tel que $fn(P) \subseteq \mathsf{Dom}(E)$ nous disons que E et P satisfont une politique de sécurité Σ si et seulement si $\mathscr{R}_{\mathsf{env}}(E, P) \models \Sigma$. Un corollaire du théorème 2.2.4 est que $\mathscr{R}_{\mathsf{env}}(E, P) \models \Sigma$ implique qu'aucun ambient qui apparaît dans P peut violer la politique de sécurité définie par Σ.

On note aussi l'importance “pratique” de l'existence d'un algorithme de reconstruction de types: le programmeur doit définir son application P, associer à toute variable dans P un domaine par E et définir, par Σ, pour chaque domaine les contraintes de sécurité auxquelles ses ambients sont soumis. C'est l'algorithme de reconstruction qui s'occupera d'engendrer le plus petit Π qui permettra de vérifier si P satisfait la politique Σ.

2.4 Version distribuée

Les systèmes de types comme le précédent possèdent des propriétés intéressantes. Toutefois ils présentent une faiblesse intrinsèque car ils ne fonctionnent que si une information globale sur les domaines et sur leurs types est toujours disponible: la dérivation d'un jugement $\Pi, E \vdash P : \mathsf{P}$ demande que les environnements Π et E contiennent des hypothèses pour tout ambient référencé dans P ainsi que pour leurs domaines. Ceci n'est clairement pas réaliste dans un calcul fondationnel pour des systèmes hautement distribués.

Pour gommer ce défaut nous définissons une version distribuée du calcul (DSSA) où tout ambient (c'est-à-dire toute “location” dans le système de processus) est décoré par un environnement de type, par un environnement de domaine et par une contrainte d'intégrité, ce qui donne la syntaxe suivante:

Distributed Processes

$$P ::= \mathbf{0} \mid \alpha.P \mid (\nu a{:}\, D)P \mid P \mid P \mid a[P]^{\mathsf{S}}_{\Pi,E} \mid !P$$

où α, Π et E sont définis comme auparavant, et pour S nous prenons une contrainte de capabilité.

Pour avoir une intuition il n'est pas inutile de penser aux fichier `class` de Java. Les fichier `class` contiennent le bytecode des applets ainsi que des informations de type et de sécurité qui sont utilisées pour la vérification du bytecode et le linking dynamique. En particulier un fichier `class` déclare le type des méthodes et des champs de la classe associée (ce que l'on appelle les *type-assertions*) et le type des identificateurs référencés par la classe (les *type-assumptions*). Quand une nouvelle classe est déchargée le vérificateur contrôle que les type-assertions sont vérifiées sous les types-assumptions.

En DSSA un ambient $a[P]^{\mathsf{S}}_{\Pi,E}$ peut être imaginé comme un fichier `class` où $a[P]$ représente le bytecode et le couple Π, E représente tant les type-assertions que les type-assumptions: intuitivement pour tout b qui apparaît dans $a[P]$ le type $\Pi(E(b))$ est une type-assertion si soit $b = a$ soit b

est le nom d'un ambient défini dans P, sinon il s'agit d'une type-assumption (b apparaît dans une capabilité de P sans y être défini).

Le système de type pour DSSA est le même qu'auparavant: les informations qui décorent les ambients ne sont pas prises en compte pour le typage statique mais au contraire pour effectuer des contrôles dynamiques au moment de la réduction. Ce qui nous conduit à une réduction typée formée par les règles **(open)**, **(struct)** et **(context)** de la section 2.2 plus celles en figure 2.2.

(in) $b[\mathtt{in}\ a.P \mid Q]^{\mathsf{S}_b}_{\Pi_b,E_b} \mid a[\mathsf{coin}\ a.R \mid S]^{\mathsf{S}_a}_{\Pi_a,E_a} \rightarrow a[R \mid S \mid b[P \mid Q]^{\mathsf{S}_b}_{\Pi_b,E_b}]^{\mathsf{S}_a}_{\Pi,E}$ $(*)$

$(*)$ *provided that, given* $\Pi, E = \Pi_b{\cdot}\Pi_a, E_b{\cdot}E_a$, *one has*
$\Pi, E \vdash b[\mathtt{in}\ a.P \mid Q] : \Pi(E(b))$, $\Pi(E(b)) \subseteq \mathsf{S}_a.\mathsf{in}$ *and* $\Pi \vdash E(a)\ \mathsf{bounds}\ \Pi(E(b))$

(out) $a[\mathsf{coout}\ a.P \mid Q \mid b[\mathtt{out}\ a.R \mid S]^{\mathsf{S}_b}_{\Pi_b,E_b}]^{\mathsf{S}}_{\Pi,E} \rightarrow b[R \mid S]^{\mathsf{S}_b}_{\Pi_b,E_b} \mid a[P \mid Q]^{\mathsf{S}}_{\Pi,E}$ $(**)$

$(**)$ *provided that* $\Pi, E \vdash b[\mathtt{out}\ a.R \mid S] : \Pi(E(b))$, *and* $\Pi(E(b)) \subseteq \mathsf{S}.\mathsf{out}$

FIG. 2.2: *New reduction rules for DSSA*

La notation $\Pi \cdot \Pi'$ dénote l'environnement résultant de la juxtaposition de Π' à Π, où donc les hypothèses de Π' cachent d'éventuelles hypothèses correspondantes de Π. De même pour $E{\cdot}E'$

La règle **(in)** étend la règle précédente par des conditions supplémentaires qui assurent que les environnements locaux aux deux ambients en question sont mutuellement compatibles et que les contraintes de sécurité sont satisfaites. En premier lieu, la règle demande que l'environnement de a soit étendu par l'environnement de b (dans le *reductum* a est décoré par Π, E qui étend Π_a, E_a). En outre la réduction demande à l'ambient entrant b (i) d'être bien typé dans l'environnement étendu et (ii) de satisfaire les contraintes de sécurité de a. Enfin la condition $\Pi \vdash E(a)\ \mathsf{bounds}\ \Pi(E(b))$ impose que l'ambient b ne modifie pas le comportement extérieur de a: a laisse entrer des nouveaux ambients seulement s'ils se conforment à la discipline locale de comportement.

La règle **(out)** effectue des contrôles similaires: noter en particulier que si a était bien typé, alors le contrôle de types pour b ne serait pas nécessaire. Toutefois, *a priori* nous ne pouvons faire aucune hypothèse sur a, et nous sommes donc obligés de vérifier que b possède bien le type que l'on suppose.

Une analyse plus approfondie de la règle **(in)** montre une intéressante correspondance entre les contraintes imposées par la cible du mouvement et les fonctions implantées par les trois composants de l'architecture de sécurité de la JVM: le *Class Loader*, le *Bytecode Verifier*, et le *Security Manager* [LY97].

$\Pi, E = \Pi_b{\cdot}\Pi_a, E_b{\cdot}E_a$ **:** Les hypothèses locales (à a) sur le type de chaque nom cachent les hypothèses remotes pour ce nom. En conséquence, l'agent entrant b ne peut pas remplacer (*spoof*) une définition de l'hôte cible a. Ceci est la politique de sécurité mise en oeuvre par le *Class Loader* qui fourni la séparation des espaces de noms et empêche les attaques de type "confusion de types".

$b[\mathtt{in}\ a.P \mid Q]^{\mathsf{S}_b}_{\Pi_b,E_b} : \Pi(E(b))$ **:** la cible du mouvement, l'ambient a, contrôle que l'agent entrant possède le type que (i) il déclare avoir si $b \notin \mathsf{Dom}(E_a)$ ou (ii) que a s'attend qu'il ait si $b \in \mathsf{Dom}(E_a)$. Ceci correspond à la politique de sécurité assurée par le *bytecode verifier*.

$\Pi(E(b)) \subseteq \mathsf{S}_a.\mathsf{in}$ **:** L'ambient a contrôle que l'agent entrant n'effectue que des actions qui sont explicitement permises par la contrainte $\mathsf{S}_a.\mathsf{in}$. Ceci correspond essentiellement à la politique de sécurité imposée par le Security Manager à la différence près que tandis que le Security Manager effectue ces contrôles dynamiquement ici notre système fait cela au moment du chargement (*class loading*).

Il faut noter que, intuitivement, tous ces contrôles sont effectués par a, l'ambient dont les frontières sont traversées.

Toutes les propriétés du système non distribué sont valables aussi pour DSSA. En plus dans DSSA il est possible de donner une formulation bien plus générale du théorème 2.2.4 selon laquelle la sûreté est garantie aussi dans le cas où le processus considéré se trouve dans un contexte non-typé et, donc, potentiellement hostile (ce qui n'est pas valable en pour le système précédent)

Théorème 2.4.1 (Local Safety) *Pour tout processus* P *et contexte* $\mathscr{C}$ *si* $\Pi, E \vdash P : \mathsf{P}$ *alors* $P \Downarrow (\mathsf{cap}\ a)^\eta$ *implique* $\mathsf{cap}\ E(a) \in \mathsf{P}^\eta$.

Notons que dans cette formulation aucune hypothèse sur le typage de $\mathscr{C}[P]$ n'est requise.

La propriété énoncée par le théorème est un résultat très intéressant pour des systèmes hautement distribués où un typage global ne peut pas être possible. Par exemple des sous-systèmes distincts peuvent utiliser des hypothèses de types incompatibles, néanmoins tant que ces hypothèses ne rentrent pas explicitement en conflit, le système reste correct car les réductions typées assurent une sûreté locale. Ainsi un agent peut en toute confiance laisser un autre agent inconnu rentrer car tant que les réductions typées sont correctement implantées les contraintes de sécurité ne peuvent pas être violées.

2.5 Conclusion

Nous avons montré que des techniques classiques de théorie des types fournissent des instruments effectifs pour caractériser les propriétés comportementales d'agents mobiles. Nous espérons avoir convaincu le lecteur que la capture du comportement implicite est essentiel pour assurer des interactions sécurisées entre agents. Nous avons aussi montré que lors de la définition d'une version distribuée du système l'on retrouve des caractéristiques spécifiques de systèmes réels

Chapitre 3

Un modèle orienté sécurité

Articles de référence: [VC99b, CGZ01]

Dans les deux derniers chapitres nous avons vu le modèle de mobilité des Ambients Mobiles. Ce modèle présente une concision et une élégance formelles indiscutables. Toutefois il possède des caractéristiques qui font que la distance entre le calcul formel et une réalisation distribuée paraît difficile à combler. Parmis ces caractéristiques on citera la capabilité `open`, le fait d'avoir une mobilité *unilatérale* et *subjective* (voir section 1.2.1) et, plus généralement, une mauvaise propension à la sécurité. Nous discuterons amplement de ce dernier point dans le chapitre 4; pour l'instant il suffit de noter que les caractéristiques citées ci-dessus ne se prêtent guère à l'écriture (ou à la spécification) de programmes dont la sécurité puisse être facilement et automatiquement vérifiée.

C'est pourquoi nous avons défini le Seal Calcul, un calcul pour la mobilité apte à modéliser une programmation sécurisée sur des systèmes hautement distribués. L'un des objectifs du projet était d'obtenir un modèle de computation qui puisse être implanté comme une bibliothèque d'un langage de programmation, afin de pouvoir programmer des applications basées sur des agents mobiles dans des domaines sensibles à la sécurité. Cet effort a en fait débouché sur JavaSeal, une bibliothèque Java développée à l'Université de Génève qui offre un noyau pour la programmation sécurisée d'agents mobiles.

Le Seal Calcul est un calcul de processus qui a été défini pour satisfaire les principes suivants (voir [VC99b]):

1. Localités explicites.
2. Absence d'état global.
3. Connectivité restreinte.
4. Reconfiguration dynamique.
5. Contrôle des accès aux ressources.

Mis à part le dernier, tous ces principes s'appliquent aussi aux Ambients Mobiles, mais de par le fait que Seal a été défini en partant de considérations pratiques et de sécurité le modèle qui en résulte est très différent.

Seal peut être sommairement décrit comme un π-calcul enrichi par une hiérarchie de locations, de la mobilité et des communication à distance (remote).

Dans ce chapitre nous présentons une variante du Seal Calcul qui constitue une évolution des travaux exposés dans les articles de référence. Cette version est l'objet d'un article qui est actuellement à un stade très avancé de préparation.

3.1 Abstractions et fonctionnalités

Le Seal calcul unifie plusieurs concepts de la programmation distribuée en trois abstractions: les *locations*, les *processus* et les *ressources*. Les locations représentent des espaces physiques tels que ceux délimités par des espaces d'adressage, des machines, des routeurs, des firewalls, des réseaux locaux ou non. Les locations incorporent aussi des espaces délimités logiquement tels que les domaines de protection, les "bac-à-sable" (sandbox) ou les applications. Les processus, eux, représentent tout flot de contrôle, tels que les threads (fils) ou les processus d'un système d'exploitation. Enfin, les ressources unifient les ressources physiques, telles que les locations de mémoire et les interfaces de périphériques, avec les services offerts par d'autres applications, les systèmes d'exploitation, ou le middleware. Syntaxiquement nous avons:

Processus Les processus sont ceux que nous avons décrits au début de la section 1.1 et sur lesquels nous allions greffer notre modèle générique d'objets mobiles. Autrement dit la composition parallèle $P \mid Q$, la restriction $(\boldsymbol{\nu} x)P$, la composition séquentielle $M.P$, la réplication $!P$ (ce qui dans le modèle générique était codé) et bien sûr la location ou agent $x[P\,]$ que nous détaillons immédiatement.

Locations Les *seals* sont des locations nommées possédant une structuration hiérarchique. Un exemple de configuration est montrée en figure 3.1. Elle montre un seal plus extérieur qui repré-

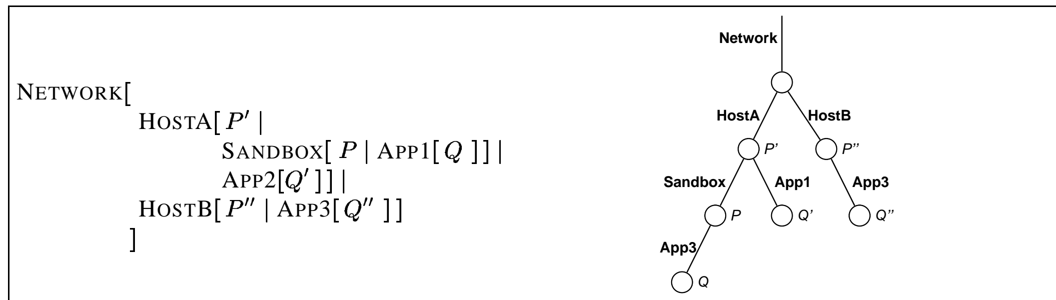


FIG. 3.1: *Seal calculus term and configuration tree.*

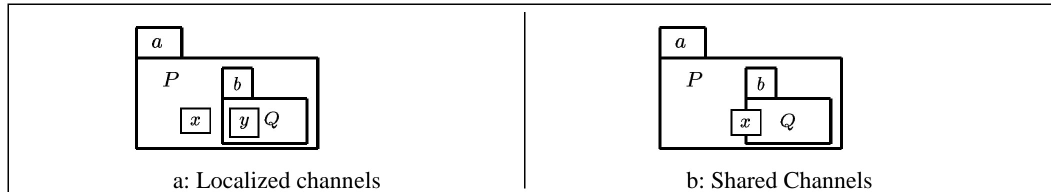
sente un réseau dont les hôtes sont représentés par d'autres seals qui contiennent à leur tour des applications et une sandbox. Pour une telle configuration nous dirons que, par exemple, HOSTA et HOSTB sont les *enfants* de NETWORK et NETWORK leur parent. Dans la même figure on trouve une représentation alternative de la même configuration. Dans l'arbre les noeuds, qui sont étiquetés par des processus, représentent des seals, tandis que les arêtes sont étiquetées par les noms des seals. Leur position sur les arêtes sert à souligner le faible lien entre la location et son nom. En fait, contrairement aux ambients (où les noms sont aussi importants que des passwords), le nom d'un seal n'est guère plus qu'une simple marque utilisée par son parent pour le dénoter.

Une autre caractéristique distinctive du Seal Calcul est que la barrière de protection fournie par les locations ne peut jamais être dissoute (cf. `open` dans les chapitres précédents). Ainsi par exemple le processus contenu dans un seal peut être désactivé (en bloquant toute interaction avec le contexte qui l'entoure), peut être détruit avec sa location, mais il ne peut jamais en être séparé.

Ressources Les seules ressources du Seal Calcul sont les *canaux*. Les canaux sont des structures computationelles nommées utilisées pour synchroniser les processus. Dans ce chapitre nous présenterons deux manières différentes d'interpréter les canaux. Dans la première interprétation les canaux seront *localisés*, tout comme les processus. Les dénotations des canaux spécifieront la localité du

canal. Ainsi, un canal x sera dénote par x^{η} où η est $*$ quand le canal est local, est $\uparrow$ quand le canal se trouve dans le parent et est n quand le canal se trouve dans le seal enfant nommé n. Dans la deuxième interprétation les canaux seront *partagés* entre deux seals en relation parent-enfant. Ainsi $x^{\uparrow}$ dénote le canal x partagé avec le parent, x^{n} le canal x partagé avec l'enfant n et, dans une variante de cette deuxième interprétation, $x^{\downarrow}$ dénote le canal x partagé avec tous les enfants; x^{*} dénote toujours un canal local.

Par exemple les dessins en figure 3.2 répresentent tous les deux la configuration $a[P \mid b[Q\,]\,]$, où le seal a enferme un processus P et un sous-seal b qui à son tour enferme un processus Q. La



a: Localized channels b: Shared Channels

FIG. 3.2: *Ressources*

figure 3.2.a correspond à l'interprétation à canaux localisés et présente deux canaux x et y, le premier localisé dans a et le deuxième dans b. Une synchronisation entre P et Q peut avoir lieu sur n'importe lequel des deux canaux. Pour se synchroniser sur x le processus P utilisera x^{*} car pour P il s'agit d'un canal local, et Q utilisera $x^{\uparrow}$ car il s'agit d'un canal localisé dans le seal parent. Similairement, pour une synchronisation sur y, P utilisera y^{b} et Q utilisera y^{*}. La figure 3.2.b illustre le cas à canaux partagés. Pour se synchroniser P et Q doivent nécessairement passer par un canal partagé entre a et b tel que x. Pour une telle synchronisation P utilisera x^{b} car il s'agit d'un canal partagé avec b, et Q utilisera $x^{\uparrow}$ car il s'agit d'un canal partagé avec le parent. Dans la variante des canaux partagés avec $\downarrow$, le processus P a la possibilité de dénoter x aussi bien par x^{b} que par $x^{\downarrow}$, mais tandis qu'une action sur x^{b} ne peut synchroniser qu'avec un processus situé dans b, une action sur $x^{\downarrow}$ peut aussi synchroniser avec un processus se trouvant dans un enfant difféerent de b. [1]

Interaction Puisque les processus sont localisés, leurs interactions peuvent être soit *locales* soit *distantes*. Une interaction sera locale quand deux processus résidant à la même location se synchronisent sur un canal qui sera nécessairement local (c'est-à-dire , indexé par $*$). Elle sera distante quand les deux processus qui se synchronisent se trouvent dans deux seals différents qui sont en relation parent-enfant. Selon l'interprétation choisie cette synchronisation aura lieu sur un canal partagé ou sur un canal localisé dans l'un de deux seals en question.

Ce sont là les seules formes d'interaction possibles. Par exemple, elles ne permettent pas la communication entre deux seals ayant le même parent, et encore moins entre seals arbitraires. Tout autre type d'interaction doit être codée en termes des ces interactions élémentaires.

La synchronisation sur des canaux est utilisée aussi bien pour la communication (le canal est utilisé pour passer un nom) que pour la mobilité (le canal est utilisé pour déplacer un seal). À chacune de ces deux formes d'interaction correspond une paire d'actions:

Communication $\overline{x}^{\eta}(y).P$ dénote un processus qui attend d'émettre y sur le canal x^{η} pour après devenir P; $x^{\eta}(y).P$ dénote un processus qui attend de lire un input, par example z, sur le canal

1. Le lecteur pourrait se demander pourquoi la possibilité d'utiliser $\downarrow$ est considérée seulement dans le cas avec canaux partagés. En fait il ne serait pas plus difficile de permettre à P en figure 3.2.a d'utiliser aussi $y^{\downarrow}$ pour dénoter le canal localisé dans b. L'utilité de $\downarrow$ est de permettre à un seal de se synchroniser avec n'importe lequel de ses enfants, sans nécessairement spécifier son nom. Le nombre restreint des interactions du Seal Calcul rendent une telle fonctionnalité pratiquement indispensable dès que l'on tâche de simuler des configurations un peu complèxes. Toutefois en présence de canaux localisés une telle fonctionnalité existe déjà car tout enfant peut se synchroniser sur un canal local de son parent ce qui ne demande pas de spécifier un nom. Ainsi dans un tel cas $\downarrow$ perd son intérêt principal.

x^η pour après devenir $P\{y{:}=z\}$, c'est-à-dire P où toute occurrence libre de y est remplacée par z.

Mobilité $\overline{x}^\eta\langle\!|y|\!\rangle.P$ dénote un processus qui attend d'envoyer son seal enfant y sur le canal x^η pour après devenir P; $x^\eta\langle\!|z|\!\rangle.P$ dénote un processus qui attend de recevoir un seal sur le canal x^η pour le réactiver avec le nom z et après devenir P.

Ainsi par exemple nous pouvons reprendre la configuration en figure 3.2.a correspondant aux canaux localisés et avoir la réduction suivante:

$$a[\,\underbrace{\overline{x}^b(w).R}_{P} \mid b[\underbrace{x^*(z).S}_{Q}]\,] \quad\rightarrow\quad a[R \mid b[S\{z{:}=w\}]\,]$$

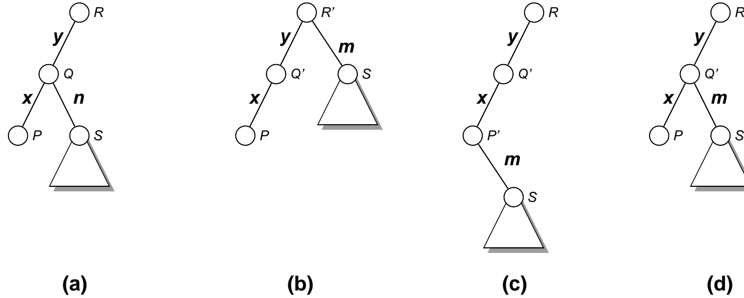
tandis que dans le cas des canaux partagés de figure 3.2.b nous avons des réductions telles que celle-ci:

$$a[\,\underbrace{x^b(z).R}_{P} \mid b[\underbrace{\overline{x}^{\uparrow}(w).S}_{Q}]\,] \quad\rightarrow\quad a[R\{z{:}=w\} \mid b[S]\,]$$

Mobilité Une des spécificités du Seal Calcul est qu'un seal et tout ce qu'il contient (processus et enfants) peut être déplacé et même dupliqué ou détruit grâce aux actions de mobilité que l'on vient de décrire.

Sur la représentation arborescente des seals la mobilité est facilement reconnaissable car elle prend une forme particulière de réécriture d'arbres. Un déplacement déconnecte un sous-arbre enraciné dans une arête y et le greffe ou bien dans le parent de y ou bien dans un des enfants de y. La réécriture peut changer l'étiquette de l'arête associée au seal déplacé et peut créer un nombre fini de copies de son arbre.

Le diagramme ci-dessous montre une configuration initiale (a) —qui est la représentation graphique du terme $R \mid y[Q \mid x[P] \mid n[S \mid \cdots]\,]$— et les trois configurations finales qui peuvent être obtenues en déplaçant le seal nommé n: (b) est obtenue en déplaçant n dans le parent et en le renommant m; (c)est obtenue en déplaçant n dans x et en le renommant m; (d) est obtenue en déplaçant n localement et en le renommant m.



Ces transformations correspondent à des réductions sur les termes. Par exemple, la transformation de (a) à (b) lors qu'elle est implantée en utilisant un canal c localisé dans y correspond à la réduction suivante

$$c^y\langle\!|m|\!\rangle.R' \mid y[\overline{c}^*\langle\!|n|\!\rangle.Q' \mid x[P] \mid n[S \mid \cdots]\,] \quad\rightarrow\quad R' \mid m[S \mid \cdots] \mid y[Q' \mid x[P]\,]$$

où $R = c^y\langle\!|m|\!\rangle.R'$ et $Q = \overline{c}^*\langle\!|n|\!\rangle.Q'$.

Comme nous avons remarqué il est possible de dupliquer des seals. Ceci n'est qu'un cas spécial de mobilité où l'action de réception crée plusieurs copies du seal reçu. La forme complète de l'action

est $x^\eta\{y_1 \dots y_n\}.P$ et elle dénote un processus qui attend *un seul* seal sur le canal x^η pour en réactiver n copies nommées $y_1, \dots, y_n$ et après devenir P. Une utilisation évidente de cette caractéristique est l'implantation d'une opération de copie:

$$(\mathsf{copy}\, x\, \mathsf{as}\, z).P \quad \stackrel{def}{=} \quad (\boldsymbol{\nu} y)(\,\overline{y}^*\{x\} \mid y^*\{x\,, z\}.P\,)$$

Intuitivement, le résultat de $\mathsf{copy}\, n\, \mathsf{as}\, m \mid n[P]$ se réduit à $n[P] \mid m[P]$. La même technique peut être utilisée pour coder une action de destruction: il suffit d'utiliser $n = 0$ lors de la réception:

$$(\mathsf{destroy}\, x\,).P \quad \stackrel{def}{=} \quad (\boldsymbol{\nu} y)(\,\overline{y}^*\{x\} \mid y^*\{\ \}.P\,)$$

Protection L'utilisation de synchronisations à distance et de la mobilité constitue une menace potentielle vis à vis de à la sécurité car un processus inconnu qui peut avoir migré d'une location non fiable (distrusted) a un moyen d'accéder aux ressources d'un hôte. Ceci est particulièrement inquiétant dans le cas de canaux localisés car en accédant à un canal local un processus extérieur pourrait avoir une influence directe sur les interactions internes. Pour remédier à cela le Seal Calcul est déjà muni au niveau syntaxique de plusieurs fonctionnalités de sécurité.

Une première ligne de défense est constituée par le fait que les actions de mobilité du Seal Calcul sont *bilatérales* et *objectives*: elle sont bilatérales car un déplacement nécessite un agrément entre deux partenaires; elle sont objectives car tout seal est déplace par l'environnement qui l'entoure (cf. l'unilatéralité et la subjectivité des actions des Ambient Mobiles: voir Section 1.2.1 dans le chapitre sur les objets mobiles). De cette manière l'environnement d'un seal contrôle tout mouvement, ce qui fait que, par exemple, les Chevaux de Troie sont facilement détectables et évitables. Par contre, comme nous avons vu dans le Chapitre 2 un modèle avec mobilité subjective requiert des techniques d'analyse complexe et pas extrêmement fines.

Une deuxième ligne de défense est fournie par la propriété de *m édiation*, c'est-à-dire l'impossibilité d'interaction directe entre seals distincts qui ne sont pas en relation parent-enfant. Ceci, couplé avec l'organisation hiérarchique des seals amène à des styles de programmation qui s'appuient sur des techniques d'interposition que l'on retrouve dans les travaux sur les machines virtuelles récursives menés au sein de la communauté systèmes d'exploitation. Tout cela, par exemple, rend très difficile l'existence de canaux cachés (covert channels).

Le troisième ingrédient de sécurité est qu'un seal a le contrôle total des noms de ses enfants. Ceci pour deux raisons: d'abord, parce qu'un seal pour permettre à un autre seal de rentrer doit effectuer une opération de réception lors de laquelle il choisi le nom pour le nouveau seal; ensuite, parceque même si le nom d'un enfant a échappé à la portée d'un père, la propriété de médiation assure qu'il n'y a aucun moyen d'utiliser une telle connaissance pour interagir directement avec l'enfant en question.

Un dernier ingrédient de sécurité est le fait que la barrière de protection fournie par un seal ne peut jamais être dissoute. Cette barrière doit être considérée comme un domaine de sécurité. Un processus enfermé dans une location peut migrer avec sa barrière, être dupliqué ou détruit avec elle, mais il ne pourra jamais la quitter.

Tous ces ingrédients constituent les défenses fournies au niveau syntaxique, sur lequelles nous pouvons construire d'autres couches de sécurité, telles que des systèmes de types, des systèmes de sécurité multiniveaux, ou des systèmes de contrôle des accès et/ou du flot de l'information.

En section 3.4 nous montrerons par exemple comment le système de type impose des restrictions très fines sur l'utilisation des noms des canaux dont un seal dispose pour interagir avec l'extérieur. En particulier nous verrons que l'interface entre un seal et l'environnement qui l'entoure ne peut faire référence qu'à des noms "publics" et que, même s'il est possible de choisir dynamiquement les noms à utiliser dans l'interface lors de la création d'un seal, une fois le seal instancié ces noms ne pourront plus être changés.

3.2 Syntaxe et Sémantique

La syntaxe du Seal Calcul est définie par les productions suivantes, où $n \geq 0$.

Processes

P	$::=$	$\mathbf{0}$	inactivity
	$\mid$	$P \mid P$	composition
	$\mid$	$!P$	replication
	$\mid$	$(\boldsymbol{\nu} x)P$	restriction
	$\mid$	$M.P$	action
	$\mid$	$x[P]$	seal

Actions

α	$::=$	$\overline{x}^{\eta}(y_1, \cdots, y_n)$	output
	$\mid$	$x^{\eta}(y_1, \cdots, y_n)$	input
	$\mid$	$\overline{x}^{\eta}\{y\}$	send
	$\mid$	$x^{\eta}\{y_1, \cdots, y_n\}$	receive

Locations

η	$::=$	$*$	local
	$\mid$	$\uparrow$	up
	$\mid$	z	down
	$\vdots$	$\downarrow$	*all down*

Nous utilisons u, v, x, y, z pour les variables, P, Q, R, S pour les processus, $\vec{x}_n$ pour dénoter le n-uplet $x_1 \cdots, x_n$, et simplement $\vec{x}$ quand l'arité n'est pas importante ou lorsqu'elle est claire dans le contexte. Nous utilisons aussi $(\boldsymbol{\nu}\vec{x}_n)P$, ou plus simplement $(\boldsymbol{\nu}\vec{x})P$, comme abréviation pour $(\boldsymbol{\nu}x_1)\ldots(\boldsymbol{\nu}x_n)P$. Comme d'habitude nous travaillerons modulo α-conversion et supposerons que dans un même processus toutes les variables liées sont distinctes.

Nous avons discuté la sémantique intuitive des termes dans la section précédente. Nous voulons seulement souligner que le π-calcul polyadique est une sous-calcul du Seal Calcul puisque il est obtenu en prohibant l'utilisation des processus seal, des action send et receive, et des location up, down et all-down (pour cette dernière nous avons utilisé une barre pointillée pour souligner qu'elle n'est pas présente dans toutes les versions).

3.2.1 Synchronisation

En section 3.1 nous avons anticipé que nous considérions trois interprétations différentes des canaux: les canaux localisés, les canaux partagés et leur variante avec $\downarrow$. Ce qui produit trois variations différentes du Seal Calcul, que nous nommons respectivement Seal Calcul *localis é*, *partagé* et *multipartagé*. Les trois variations diffèrent dans la manière dont les processus distants se synchronisent sur les canaux. C'est pourquoi les trois variations sont obtenues par une définition unique du Seal Calcul paramétrique dans la modalité de synchronisation. Formellement nous introduisons une relation ternaire $\mathsf{synch}_{_}(_,_) \in \mathbf{Variables} \times \mathbf{Locations} \times \mathbf{Locations}$. Intuitivement, $\mathsf{synch}_y(\eta_1, \eta_2)$ indique que pour tout canal x une action sur x^{η_1} effectuée dans un seal parent peut synchroniser avec une coaction sur x^{η_2} effectuée dans un seal enfant y.

Définition 3.2.1 [y-correspondance] Soient η_1, η_2 des locations et y une variable.

1. $\mathsf{synch}^S_y(\eta_1, \eta_2) \stackrel{def}{=} (\eta_1 = y \wedge \eta_2 = \uparrow)$.
   Si $\mathsf{synch}^S_y(\eta_1, \eta_2)$ est vrai alors nous disons que 'η_1 et η_2 sont locations "y-*shared*-corresponding".
2. $\mathsf{synch}^M_y(\eta_1, \eta_2) \stackrel{def}{=} (\eta_1 = y \wedge \eta_2 = \uparrow) \vee (\eta_1 = \downarrow \wedge \eta_2 = \uparrow)$.
   Si $\mathsf{synch}^M_y(\eta_1, \eta_2)$ est vrai alors nous disons que η_1 et η_2 sont locations "y-*multi*-corresponding".
3. $\mathsf{synch}^L_y(\eta_1, \eta_2) \stackrel{def}{=} (\eta_1 = y \wedge \eta_2 = *) \vee (\eta_1 = * \wedge \eta_2 = \uparrow)$.
   Si $\mathsf{synch}^L_y(\eta_1, \eta_2)$ est vrai alors nous disons que η_1 et η_2 sont "y-*located*-corresponding".

□

3.2.2 Sémantique de réduction

La sémantique du Seal Calcul est donnée par les définitions de congruence structurale et de sémantique de réduction

Définition 3.2.2 [Congruence structurale] La congruence structurale $\equiv$ est la plus petite congruence sur les processus qui satisfait les axiomes suivants.

$$\begin{array}{rclrcll} P \mid \mathbf{0} & \equiv & P & (\boldsymbol{\nu} x)(\boldsymbol{\nu} y)P & \equiv & (\boldsymbol{\nu} y)(\boldsymbol{\nu} x)P & x \neq y \\ P \mid Q & \equiv & Q \mid P & (\boldsymbol{\nu} x)(P \mid Q) & \equiv & P \mid (\boldsymbol{\nu} x)Q & x \notin fn(P) \\ P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R & (\boldsymbol{\nu} x)\mathbf{0} & \equiv & \mathbf{0} & \end{array}$$

□

Considérons le processus $(\boldsymbol{\nu}\vec{x})P$. Puisque nous travaillons modulo alpha-conversion nous pouvons supposer que $x_1, \ldots, x_n$ sont tous différents. En outre la relation de congruence structurale que l'on vient de définir nous dit que leur ordre n'est pas significatif car les n restrictions peuvent être librement permutées. Ces deux observations ensemble nous disent que chaque fois qu'un vecteur $\vec{x}$ est lié par une restriction nous pouvons sans perte de généralité le considérer comme un ensemble (bien sûr cela n'est pas vrai si le vecteur est lié par une action de input). Ce qui justifie l'utilisation dans la suite des notations telles que $(\boldsymbol{\nu}\vec{x} \cap \vec{y})P$ ou $(\boldsymbol{\nu}\vec{x} \setminus \vec{y})P$ (où $\cap$ et $\setminus$ dénotent l'intersection et la différence ensemblistes et avec la convention que $(\boldsymbol{\nu}\varnothing)P = P$).

La définition 3.2.2 est formée par des règles standard du π-calcul. Ces règles sont aussi présentes dans la définition des Ambients. Toutefois par rapport à ces derniers la congruence structurale du Seal Calcul présente une différence de poids: l'absence de l'équivalence suivante [CG98]

$$(\boldsymbol{\nu} x)y[P] \quad \equiv \quad y[(\boldsymbol{\nu} x)P] \qquad x \neq y \tag{3.1}$$

L'absence de cette règle n'est pas due aux caractéristiques particulières du Seal Calcul mais à la simple présence, en Seal, de la duplication. Considérer les processus $(\boldsymbol{\nu} x)y[P]$ et $y[(\boldsymbol{\nu} x)P]$ équivalents n'est pas correct car si nous composons les deux processus avec le processus de duplication $Q = \mathsf{copy}\ y\ \mathsf{as}\ z$ nous obtenons

$$y[(\boldsymbol{\nu} x)P] \mid Q \rightarrow y[(\boldsymbol{\nu} x)P] \mid z[(\boldsymbol{\nu} x)P] \quad \text{and} \quad (\boldsymbol{\nu} x)y[P] \mid Q \rightarrow (\boldsymbol{\nu} x)(y[P] \mid z[P])$$

ainsi le premier processus conduit à un système ou les seal y et z ont chacun un canal privé x, tandis que dans l'autre cas il partagent un canal commun x. Ce discours n'est pas valable seulement pour la duplication en Seal: la règle d'extrusion (3.1) devrait être éliminée des Ambients si ces derniers étaient enrichis par une action de duplication telle que celle de la section 3.1. Et les conséquences seraient les mêmes que celles que nous discutons dans ce qui suit.

Relation de réduction

L'absence de la règle (3.1) a plusieurs conséquences dont la plus importante est la nécessité de gérer l'extrusion des noms localement restreints qui sont communiqués à distance. Considérons l'exemple suivant:

$$n[(\boldsymbol{\nu} y).(\overline{x}^{\uparrow}(y).\ \ldots)] \tag{3.2}$$

le seal n crée localement un nom privé y et après il essaye de le communiquer à travers un canal x à son parent. Il existe deux possibilités: ou bien nous censurons un tel comportement, ou bien nous faisons extruder de n la restriction $(\boldsymbol{\nu} y)$ de façon que sa portée inclue tant n que le processus du parent destinataire de y. La première solution est trop restrictive car la possibilité de communiquer

des noms privés, en plus d'être l'une des caractéristiques fondamentales du π-calcul, est très utile et utilisée (par exemple dans les protocole cryptographiques). Ainsi il faut utiliser la deuxième solution de façon que, par exemple, dans la variation avec canaux partagés on ait des réductions telles que

$$x^n(z).P \mid n[(\boldsymbol{\nu} y).(\overline{x}^{\uparrow}(y).Q)\,] \quad \rightarrow \quad (\boldsymbol{\nu} y)(P\{z{:}=y\} \mid n[Q\,])$$

On peut noter que la restriction $(\boldsymbol{\nu} y)$, qui est locale à n, est extrudée à l'extérieur de n de manière à inclure dans sa portée le processus destinataire de y. Puisque dans Seal nous ne pouvons pas utiliser la règle (3.1), il faut que l'extrusion soit directement gérée par la réduction. Nous devons donc déterminer l'ensemble des variables à extruder lors d'une réduction et cet ensemble doit être déterminé de manière univoque, quitte à retomber dans le même problème que celui de l'équivalence (3.1). La solution adoptée est celle d'extruder toute restriction locale qui concerne un nom communiqué au parent, et aucune autre.

Ceci est obtenu par les règles de réduction en figure 3.3 (auxquelles on ajoute les règles pour le contexte et pour la congruence structurale habituelles), et par la troisième de ces règles en particulier. Mais procédons par ordre et commençons par noter que les règles non-locales sont paramétriques

$$\begin{array}{rcll}
x^*(\vec{u}).P \mid \overline{x}^*(\vec{v}).Q & \rightarrow & P\{\vec{u}{:}=\vec{v}\} \mid Q & \\
\overline{x}^{\eta_1}(\vec{v}).P \mid y[(\boldsymbol{\nu}\vec{z})(x^{\eta_2}(\vec{u}).Q_1 \mid Q_2)] & \rightarrow & P \mid y[(\boldsymbol{\nu}\vec{z})(Q_1\{\vec{u}{:}=\vec{v}\} \mid Q_2)] & \vec{v}\cap\vec{z}=\varnothing \\
x^{\eta_1}(\vec{u}).P \mid y[(\boldsymbol{\nu}\vec{z})(\overline{x}^{\eta_2}(\vec{v}).Q_1 \mid Q_2)] & \rightarrow & (\boldsymbol{\nu}\vec{v}\cap\vec{z})(P\{\vec{u}{:}=\vec{v}\} \mid y[(\boldsymbol{\nu}\vec{z}\setminus\vec{v})(Q_1 \mid Q_2)]) & \\
x^*\{\vec{u}_n\}.P_1 \mid \overline{x}^*\{v\}.P_2 \mid v[Q] & \rightarrow & P_1 \mid u_1[Q] \mid \cdots \mid u_n[Q] \mid P_2 & \\
\overline{x}^{\eta_1}\{v\}.P \mid v[R] \mid y[(\boldsymbol{\nu}\vec{z})(x^{\eta_2}\{\vec{u}_n\}.Q_1 \mid Q_2)] & \rightarrow & P \mid y[(\boldsymbol{\nu}\vec{z})(Q_1 \mid Q_2 \mid u_1[R] \mid \cdots \mid u_n[R])] & \\
x^{\eta_1}\{\vec{u}_n\}.P \mid y[(\boldsymbol{\nu}\vec{z})(\overline{x}^{\eta_2}\{v\}.Q_1 \mid v[R] \mid Q_2)] & \rightarrow & P \mid u_1[R] \mid \cdots \mid u_n[R] \mid y[(\boldsymbol{\nu}\vec{z})(Q_1 \mid Q_2)] &
\end{array}$$

où $fn(R)\cap\vec{z}=\varnothing$, $x\notin\vec{z}$, et $\mathsf{synch}_y(\eta_1,\eta_2)$ est vrai.

FIG. 3.3: *Reduction rules for the Seal Calculus*

dans la définition de synch: une variation différente du Seal Calcul est obtenue selon que synch correspond à l'interprétation localisée, partagée où multipartagée des canaux.

Les trois premières règles définissent la sémantique des communications. La première décrit la communication locale, qui coïncide avec celle du π-calcul. La deuxième décrit la communication d'un n-uplet $\vec{v}$ d'un parent à son enfant y, qui a lieu si (i) η_1 et η_2 sont des locations y-corresponding, (ii) le canal x n'est par restreint dans y (i.e., $x\notin\vec{z}$) et (iii) il n'y a pas de capture des variables communiquées (i.e., $\vec{v}\cap\vec{z}=\varnothing$). La troisième règle est celle qui gère l'extrusion des restrictions locales, car elle correspond au cas où un enfant y communique un n-uplet $\vec{v}$ de variables à son parent. Comme auparavant η_1 et η_2 doivent être des locations y-corresponding et x ne doit pas être restreint localement (i.e., $x\notin\vec{z}$). Les restrictions locales (dans y) de variables qui sont communiquées au parent (i.e., $\vec{v}\cap\vec{z}$) sont extrudées, tandis que les restrictions pour les autres variables (i.e., $\vec{z}\setminus\vec{v}$) restent dans y.

Les trois autres règles concernent la mobilité. Pour la première règle le corps du seal spécifié dans l'action send est déplacé en n copiés nommées comme spécifié par l'action receive. La deuxième établit qu'un seal peut être déplacé dans un enfant y pourvu que (i) η_1 et η_2 sont locations y-corresponding, (ii) le canal x n'est pas restreint dans y (i.e., $x\notin\vec{z}$) et (iii) les variables libres dans le corps déplacé ne sont pas capturées (i.e., $fn(R)\cap\vec{z}=\varnothing$). Ces deux règles ressemblent aux deux premières règles de communication. Par contre la troisième règle diffère de la règle de communication correspondante car aucune extrusion n'est effectuée. Ce qui ne signifie pas que ce ne serait pas utile. En fait on peut immaginer deux manières par lesquelles une variable localement

restreinte pourrait sortir du seal où elle fut à l'origine déclarée: par une communication explicite comme pour le seal dans (3.2), ou par une action de mobilité, comme dans cette légère variante du seal défini dans (3.2)

$$n[(\boldsymbol{\nu}y)(\overline{x}^{\uparrow}\{m\} \mid m[\overline{y}()\,])\,] \tag{3.3}$$

Si m sortait de n il ferait sortir de la portée de n la variable y. On pourrait donc, comme dans le cas de la communication, extruder la restriction sur y (c'est ce que nous faisions dans [VC99b]). Nous avons choisi une discipline plus restrictive car la dernière règle en figure 3.3 demande que le corps d'un seal sortant ne contienne comme libre aucune variable localement restreinte. Ce qui implique que toute variable libre dans le corps d'un seal sortant doit être déjà connue au parent, soit car il s'agit d'une variable non locale, soit car la variable lui a été précédemment communiquée. Ainsi une configuration comme celle de (3.3) réduit seulement si y a été explicitement communiquée comme par exemple dans:

$$n[(\boldsymbol{\nu}y)(\overline{u}^{\uparrow}(y).\overline{x}^{\uparrow}\{m\} \mid m[\overline{y}()\,])\,]$$

L'action de output cause l'extrusion de $(\boldsymbol{\nu}y)$ et le déplacement peut donc avoir lieu.

Il existe deux raisons pour préférer un tel comportement plus restrictif. D'abord ce choix est plus orienté sécurité car de cette manière les ressources locales (les variables privées) ne peuvent être exportées que par une communication explicite, ce qui barre la route à d'éventuels Chevaux de Troie en sortie. En outre ce choix est plus proche de l'implantation car de toute manière les variables libres d'un seal déplacé sont des "handles" pour lesquelles l'implantation doit passer une référence, quitte à les rendre inutilisables.

3.2.3 Discussion

Revenons sur les cinq principes de design énoncés dans l'introduction du chapitre et examinons comment le calcul les traite.

1. Les localités sont clairement explicites; le modèle Seal différencie les ressources locales de celles distantes, et elles ne peuvent pas se mélanger car les barrières de protection fournies par les locations ne peuvent pas être dissoutes.
2. Le modèle Seal ne dépend jamais d'un état global. Un seal ne peut utiliser que les noms de ses enfants, et ses seules interactions sont avec ces enfants ou le parent, ce qui signifie que la synchronisation n'implique jamais plus de deux niveaux emboîtés.
3. Les interactions sont restreintes aux interaction locales ou de voisinage (parent ou enfant). Toute autre forme de communication doit être explicitement programmée. Ce qui signifie que nous pouvons modéliser facilement des opérations "off-line" ou les effets d'un firewall.
4. La reconfiguration dynamique est obtenue par trois ingrédients: la mobilité, le passage des noms et la liaison dynamique. La mobilité supporte la migration, la duplication et la destruction des seals, et elle peut modéliser une reconfiguration topologique des locations. Le passage de noms peut être utilisé pour explicitement reconfigurer pour un nouvel environnement un seal y ayant migré, tandis qu'une reconfiguration implicite est obtenue par la liaison dynamique de $\uparrow$ et, dans le cas de canaux multi-partagés, de $\downarrow$. Le premier dénote l'environnement parent et est dynamiquement lié au parent courant, ce qui permet la reconfiguration dynamique d'un seal après un déplacement, et éventuellement la possibilité d'une mise à jour transparente des services. Le second dénote tout seal enfant et est dynamiquement lié pour prendre en compte tout seal entrant, ce qui permet une reconfiguration automatique pour toute arrivée d'un nouveau seal.
5. Nous avons longuement argumenté que la syntaxe même du calcul a été définie pour fournir différentes caractéristiques de sécurité qui offrent un contrôle d'accès aux ressources de base. Ainsi l'extrusion de noms privés d'un seal doit être explicitement programmée, la mobilité

est objective est nécessite d'une entente mutuelle, la connectivité non locale est limitée, les locations séparent les espaces de noms des seals, etc. Ceci fourni une base cohérente au dessus de laquelle des outils plus fins de sécurité peuvent être implantés.

3.3 Équivalences

Au moment de la rédaction de ce mémoire l'étude de la théorie d'équivalence du Seal Calcul est en pleine investigation. Des travaux préliminaires sont contenus dans [CV99] et ces résultats sont partiellement utilisés dans [VC99a]. L'idée de l'étude dans [CV99] est de déterminer ce qu'une notion adéquate d'équivalence sémantique entre agents pourrait être. Par exemple dans [CG98, CG99a] Cardelli et Gordon introduisent et étudient une équivalence contextuelle à la Morris pour les Ambients Mobiles selon laquelle le processus $(\boldsymbol{\nu} n)n[P\,]$, lorsque n n'apparaît pas libre dans P, ne peut pas être distingué de $\mathbf{0}$. L'idée est que puisque le nom n n'est connu ni à l'extérieur ni à l'intérieur du processus, aucun autre ambient ne peut exercer une capabilité sur lui, et donc c'était comme si l'ambient n n'existait pas. Ceci se résume dans la *perfect firewall equation*:

$$(\boldsymbol{\nu} n)n[P\,] \simeq \mathbf{0} \quad \text{for } n \notin \mathit{fn}(P)$$

Nous nous sommes donc demandés si ce firewall était si parfait que ça. En fait l'équation ci-dessus n'assure pas que n n'aura aucune interaction avec le contexte: n peut très bien rentrer dans un ambient qui s'exécute en parallèle, ou sortir de l'ambient où il se trouve; autrement dit n possède une totale liberté de mouvement. Plus formellement cela signifie que si l'on considère par exemple la *commitment semantics* définie pour les Ambients Mobiles dans [CG00] le processus $(\boldsymbol{\nu} n)n[P\,]$ peut émettre les actions $\mathtt{in}\ m$ et $\mathtt{out}\ m$.[2] Ce qui signifie que dans aucune relation de bisimulation raisonnable basée sur le système de [CG00] le processus $\mathbf{0}$, qui n'émet rien, ne sera équivalent à $(\boldsymbol{\nu} n)n[P\,]$. Il est donc légitime de se poser des questions sur l'adéquation de l'observation utilisée pour la définition de $\simeq$. Si par exemple nous nous plaçons dans un cas distribué avec “partial trust” (comme celui de la section 2.4 du chapitre 2) où donc les agents évoluent dans un milieu potentiellement hostile et leur correction ne peut être vérifiée que localement, il est clair qu'une relation comme $\simeq$ n'est pas adéquate. En fait on voudrait pouvoir vérifier que n n'a pas la possibilité de se déplacer —par exemple dans le cas où n ait pu accéder à des information confidentielles— car, autrement, il pourrait profiter de la complicité d'un autre ambient, dont la correction (sémantique) n'a pas été vérifiée, pour communiquer ces informations. C'est pourquoi dans [CV99] nous avons défini une commitment semantics pour Seal, montré qu'être équivalent à $\mathbf{0}$ implique l'absence de toute interaction avec le contexte et démontré que dans Seal $(\boldsymbol{\nu} n)n[P\,] \approx \mathbf{0}$ est vrai (même si n apparaît dans P).

Toutefois ceci ne répond pas à notre question initiale, c'est-à-dire ce qu'une notion adéquate d'équivalence pour les agents est: nous avons trouvé une relation qui, *mutatis mutandis*, est plus fine que celle de [CG98, CG99a] et qui semble être appropriée au cas distribué ci-dessus; toutefois rien ne nous assure que cette relation soit adéquate.

Un début de réponse à cette question est apporté, dans le cas des ambients, par un récent travail de Merro et Hennessy [MH02] qui poursuit les mêmes objectifs que [CV99] mais de manière plus générale et élégante. Leur travail part de la constatation faite par Sangiorgi dans [San01] que la théorie algébrique des Ambients n'est pas très riche car proche de la congruence structurelle. Le but de [MH02] est donc de modifier les Ambients Mobiles de façon qu'ils aient une théorie équationnelle riche, raisonnable, adéquate et utilisable en pratique. Que signifient ces quatre propriétés? Riche: qu'elle prouve des équivalences plus intéressantes que la simple congruence structurelle;

2. Plus précisément selon le système en [CG00] $(\boldsymbol{\nu} n)n[P\,]$ peut émettre $\overline{\mathit{enter}\ m}$ et $\mathit{exit}\ m$.

raisonnable: qu'elle soit une équivalence contextuelle qui préserve la réduction et une quelque propriété observationnelle simple; adéquate: qu'elle soit assez insensible aux changements d'observation (techniquement de *barbs*); utilisable: qu'elle puisse être exprimée en termes de bisimulation, dont la nature co-inductive assure l'existence de techniques de preuve puissantes.

Un premier pas dans cette direction a été fait par Levi et Sangiorgi [LS00] qui ont enrichi les ambients par des coactions. Ces coactions doivent se synchroniser avec les actions pour que la réduction ait lieu, ce qui permet d'avoir une théorie équationnelle bien plus satisfaisante. Toutefois il s'agit une fois de plus d'une équivalence contextuelle et elle ne présente pas les deux dernières propriétés recherchées. Dans [MH02] Merro et Hennessy étendent (et modifient) le système de [LS00] en ajoutant aux Ambients, en plus des coactions, des *passwords*: une action et la coaction correspondante se synchroniseront seulement s'ils sont en possession du même password. Merro and Hennessy définissent ensuite une équivalence basée sur la bisimulation et montrent qu'elle coïncide avec une équivalence contextuelle qui est invariante sur une grande variété d'observations. Autrement dit, ils montrent que leur extension satisfait bien les quatre propriétés recherchées.

Il est très intéressant de noter que les modifications faites au Calcul des Ambients pour obtenir un tel résultat, le rapprochent de plus en plus du Seal Calcul: [LS00] demande que la mobilité soit l'effet d'une synchronisation entre processus; [MH02] ne fait rien d'autre que de demander que la mobilité ait lieu sur des canaux (car les passwords de Merro et Hennessy peuvent être très facilement assimilés à des canaux) et que la coaction d'une `out` soit positionnée au même endroit q'une action de receive dans Seal. Le tout dernier pas qui sépare ce formalisme de Seal est que Seal utilise une mobilité objective (l'envoi de l'agent est effectué par l'environnement plutôt que par l'agent même) tandis que pour les formalismes des Ambients la mobilité est subjective (l'envoi de l'agent est effectué par l'agent même). Il ne serait donc pas étonnant que les mêmes résultats établis par Merro et Hennessy pour les Ambients soient aussi valables pour le Seal Calcul, sans que ce dernier ne doive être modifié. C'est ce que nous montrons, quoique partiellement, dans [CZ02], où nous définissons une congruence observationelle pour le Seal Calcul, nous introduisons une relation de bisimilarité, et nous montrons que cette dernière est correcte par rapport à la congruence.

3.4 Système de types

Dans cette section nous développons un système de types pour chaque variante du Seal Calcul. Les types caractériseront les interactions qu'un agent peut avoir avec son environnement.

3.4.1 Interfaces

L'idée est de décrire toutes les interaction d'un seal avec son environnement en collectant tous les canaux sur lesquels de telles interaction peuvent avoir lieu. Ce qui induit une notion d'interface comme un ensemble de *canaux vers le haut*, c'est-à-dire des canaux qui peuvent se synchroniser avec un processus qui réside dans le parent.

Il n'est pas nécessaire de collecter dans une interface tous les canaux vers le haut d'un seal. Les canaux sur lesquels le seal est à l'écoute suffisent:

> *L'*interface *d'un seal est est l'ensemble des canaux vers le haut sur lesquels les processus locaux du seal sont censés écouter ainsi que leurs types.*

Une justification argumentée de ce choix se trouve dans [CGZ01]. Pour ce rapport nous nous limitons à donner deux exemples expliquant pourquoi un tel choix est raisonnable. Considérons une machine en réseau. Pour le monde extérieur l'interface d'une telle machine est donnée par l'ensemble des "ports" sur lesquels un dæmon est à l'écoute, avec le type des messages acceptés. Ainsi l'interface d'un serveur ftp et mail typique sera [21: *ftp*; 23: *telnet*; 79: *finger*; 110: *pop3*; 143: *imap*; ...].

Un exemple différent vient de l'analogie avec le monde orienté objets. Si nous considérons un seal comme un objet alors un processus qui y est renfermé écoutant sur un canal vers le haut m peut être assimilé à une méthode associée au message m. En d'autres termes, l'envoi d'un message m avec argument v à un objet x (en syntaxe Java `x.m(v)`) peut être modélisé en Seal par l'action $\overline{m}^{x}(v)$, laquelle sera bien typée dans notre système de types seulement si une paire m: M est présente dans l'interface de x (avec v de type M).

Plus généralement nous considérerons des interfaces telles que $[x_1\colon \star; x_2\colon \mathtt{Ch}\,T; x_3\colon A; x_4\colon \mathtt{Id}\,A]$ qui caractérisent des agents qui peuvent: (i) se synchroniser avec une opération d'output sur x_1 dans le parent; (ii) lire sur x_2 le nom d'un canal de type $\mathtt{Ch}\,T$ (c'est-à-dire le nom d'un canal qui transporte des objets de type T); (iii) recevoir sur x_3 un seal possédant l'interface A; (iv) lire sur x_4 le nom d'un seal d'interface A. Il faut noter que la synchronisation sur x_3 demandera l'utilisation de primitives de mobilité tandis que celle sur x_4 demandera des primitives de communication.

3.4.2 Syntaxe des types

Les trois variantes du Seal calcul partagent la même syntaxe pour les types (où $m \geq 1, n \geq 0$)

Exchange Types

T	$::=$	$M_1 \times \cdots \times M_m$	messages
	$\mid$	A	agents

Annotations

Z	$::=$	$\curvearrowright$	mobile
	$\mid$	$\veebar$	immobile

Message Types

M	$::=$	$\star$	empty
	$\mid$	$\mathtt{Ch}\,T$	channel names
	$\mid$	$\mathtt{Id}^Z A$	agent names

Interfaces

A	$::=$	$[x_1\colon T_1; \cdots; x_n\colon T_n]$

T classifie les valeurs *échangeables*, c'est-à-dire les entités computationnelles qui peuvent être déplacées ou communiquées sur un canal. Celles-ci sont soit des (n-uplets de) *messages* — valeurs de bases, noms de canaux, noms de seals— soit des seals (plus précisément des corps de seals)

M classifie les *messages*, c'est-à-dire les entités qui peuvent être communiquées pas des actions d'input/output sur des canaux. Un message peut être une synchronisation sans contenu (de type $\star$), le nom d'un canal (type $\mathtt{Ch}\,T$), ou le nom d'un seal (type $\mathtt{Id}^Z A$).

Z spécifie l'attribut de mobilité d'un seal (voir [CGG99]): un attribut $\curvearrowright$ caractérise un seal mobile tandis qu'un attribut $\veebar$ caractérise un seal immobile.

A classifie les agents en enregistrant leurs interactions possibles avec l'environnement. Les interfaces sont utilisées dans les types $\mathtt{Id}^Z A$ pour classifier les noms dénotant des agents d'interface A, et dans les types $\mathtt{Ch}\,A$ pour classifier les canaux sur lesquels des agents d'interface A peuvent être déplacés.

Quelques modifications doivent être apportées à la syntaxe non-typée de la section 3.2 pour prendre en compte les types. Ainsi nous devons typer explicitement les variables lors de leur liaison: $(\boldsymbol{\nu} x\colon M)$ et $x^\eta(y_1\colon M_1, \ldots, y_n\colon M_n)$ (ce dernier est abrégé par $x^\eta(\vec{y}\colon \vec{M})$). De la même manière la définition de *fn* et les règles de congruence structurale nécessitent de petits réajustements.

3.4.3 Règles de typage

Les règles de typage permettent de dériver des jugements de différentes formes, dont la plus importante est $\Gamma \vdash_u P$ qui établi que un processus P est bien typé lorsqu'il est contenu dans le seal u.

La plupart des règles sont communes aux trois variantes avec, bien sûr, de petites différences pour le typage des canaux. Nous commençons par le système pour la variante à canaux partagés. Après avoir expliqué les règles en détail nous montrerons comment modifier ce système pour les autres variantes.

Canaux partagés

Dans le cas de canaux partagés nous voulons que l'interface d'un seal enregistre tous les canaux de input et receive qui apparaissent avec une location $\uparrow$. Le système de types interdira une opération de write ou send sur un canal x^z si x n'apparaît pas, avec le type correct, dans l'interface de z.

Environment

(Env Empty) $\dfrac{}{\varnothing \vdash \diamond}$

(Env Add) $\dfrac{\Gamma \vdash M}{\Gamma, x\colon M \vdash \diamond}$ $\quad x \notin \mathsf{Dom}(\Gamma)$

(Var) $\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash x\colon \Gamma(x)}$

Well-formed types

(Type $\star$) $\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash \star}$

(Type $\mathtt{Id}$) $\dfrac{\Gamma \vdash A}{\Gamma \vdash \mathtt{Id}^Z A}$

(Type $\mathtt{Ch}$) $\dfrac{\Gamma \vdash T}{\Gamma \vdash \mathtt{Ch}\, T}$

(Type Tuple) $\dfrac{\Gamma \vdash M_1 \ldots \Gamma \vdash M_n}{\Gamma \vdash M_1 \times \cdots \times M_n}$

(Type Interface) $\dfrac{\Gamma \vdash \diamond \quad \forall i \in 1..n \;\; \Gamma \vdash x_i\colon \mathtt{Ch}\, T_i}{\Gamma \vdash [x_1\colon T_1, \ldots, x_n\colon T_n]}$

Processes

(Dead) $\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash_u \mathbf{0}}$

(Par) $\dfrac{\Gamma \vdash_u P_1 \qquad \Gamma \vdash_u P_2}{\Gamma \vdash_u P_1 \mid P_2}$

(Bang) $\dfrac{\Gamma \vdash_u P}{\Gamma \vdash_u !P}$

(Res) $\dfrac{\Gamma, x\colon M \vdash_u P}{\Gamma \vdash_u (\boldsymbol{\nu} x\colon M)P}$

(Seal) $\dfrac{\Gamma \vdash x : \mathtt{Id}^Z A \qquad \Gamma \vdash_x P}{\Gamma \vdash_u x[P]}$

(Output Local) $\dfrac{\Gamma \vdash x : \mathtt{Ch}\, \vec{M} \qquad \Gamma \vdash \vec{y} : \vec{M} \qquad \Gamma \vdash_u P}{\Gamma \vdash_u \overline{x}^*(\vec{y}).P}$

(Input Local) $\dfrac{\Gamma \vdash x : \mathtt{Ch}\, \vec{M} \qquad \Gamma, \vec{y}\colon \vec{M} \vdash_u P}{\Gamma \vdash_u x^*(\vec{y}\colon \vec{M}).P}$

(Output Up) $\dfrac{\Gamma \vdash x : \mathtt{Ch}\, \vec{M} \qquad \Gamma \vdash \vec{y} : \vec{M} \qquad \Gamma \vdash_u P}{\Gamma \vdash_u \overline{x}^\uparrow(\vec{y}).P}$

(Input Up) $\dfrac{\Gamma \vdash u : \mathtt{Id}^Z A \qquad \Gamma \vdash x : \mathtt{Ch}\, \vec{M} \qquad \Gamma, \vec{y}\colon \vec{M} \vdash_u P}{\Gamma \vdash_u x^\uparrow(\vec{y}\colon \vec{M}).P}$ $\quad (x\colon \vec{M}) \in A$

(Output Down) $\dfrac{\Gamma \vdash z : \mathtt{Id}^Z A \qquad \Gamma \vdash \vec{y} : \vec{M} \qquad \Gamma \vdash_u P}{\Gamma \vdash_u \overline{x}^z(\vec{y}).P}$ $\quad (x\colon \vec{M}) \in A$

(Input Down) $\dfrac{\Gamma \vdash_u z : \mathtt{Id}^Z A \qquad \Gamma \vdash x : \mathtt{Ch}\, \vec{M} \qquad \Gamma, \vec{y}\colon \vec{M} \vdash_u P}{\Gamma \vdash_u x^z(\vec{y}\colon \vec{M}).P}$

(Snd Local) $\dfrac{\Gamma \vdash x : \mathtt{Ch}\, A \qquad \Gamma \vdash y : \mathtt{Id}^\frown A \qquad \Gamma \vdash_u P}{\Gamma \vdash_u \overline{x}^*\{y\}.P}$

(Rcv Local)
$$\frac{\Gamma \vdash x : \mathtt{Ch}\,A \quad \Gamma \vdash y_i : \mathtt{Id}^{Z_i} A\ (i=1..n) \quad \Gamma \vdash_u P}{\Gamma \vdash_u x^*\{\vec{y}\}.P}$$

(Snd Up)
$$\frac{\Gamma \vdash x : \mathtt{Ch}\,A \quad \Gamma \vdash y : \mathtt{Id}^{\frown} A \quad \Gamma \vdash_u P}{\Gamma \vdash_u \overline{x}^{\uparrow}\{y\}.P}$$

(Rcv Up)
$$\frac{\Gamma \vdash u : \mathtt{Id}^{Y} B \quad \Gamma \vdash x : \mathtt{Ch}\,A \quad \Gamma \vdash y_i : \mathtt{Id}^{Z_i} A\ (i=1..n) \quad \Gamma \vdash_u P}{\Gamma \vdash_u x^{\uparrow}\{\vec{y}\}.P} \quad (x\colon A) \in B$$

(Snd Down)
$$\frac{\Gamma \vdash z : \mathtt{Id}^{Z} A \quad \Gamma \vdash y : \mathtt{Id}^{\frown} B \quad \Gamma \vdash_u P}{\Gamma \vdash_u \overline{x}^{z}\{y\}.P} \quad (x\colon B) \in A$$

(Rcv Down)
$$\frac{\Gamma \vdash z : \mathtt{Id}^{Y} B \quad \Gamma \vdash x : \mathtt{Ch} A \quad \Gamma \vdash y_i : \mathtt{Id}^{Z_i} A\ (i=1..n) \quad \Gamma \vdash_u P}{\Gamma \vdash_u x^{z}\{\vec{y}\}.P}$$

Nous ne discutons que les règles les plus importantes.

(Env Add): Les environnements sont des listes de déclarations variable/type-message. Une nouvelle déclaration peut être ajoutée à un environnement Γ si la variable n'est pas déjà déclarée dans Γ et si son type est bien formé sous Γ.

(Res) Quoique apparemment très simple (Res) est en réalité très sophistiquée car par son biais le système de types impose des conditions subtiles sur les noms qui apparaissent dans les interfaces.

En premier lieu il est facile de vérifier que si $\Gamma, x\colon T \vdash_u P$ est prouvable alors $\Gamma, x\colon T \vdash \diamond$ ce qui implique $x \notin \mathsf{Dom}(\Gamma)$. Ceci signifie qu'aucune confusion n'est possible lors du typage de deux liaisons emboîtées d'une même variable.

Considérons maintenant les processus

$$(\boldsymbol{\nu} x\colon \mathtt{Ch}\ M)(\boldsymbol{\nu} y\colon \mathtt{Id}\ [x\colon M])\,y[x^{\uparrow}(u\colon M)\,]$$

et

$$(\boldsymbol{\nu} x\colon \mathtt{Ch}\ M)(\boldsymbol{\nu} y\colon \mathtt{Id}\ [x\colon M])\,y[(\boldsymbol{\nu} x\colon \mathtt{Ch}\ M)x^{\uparrow}(u\colon M)\,].$$

Le premier est bien typé, le second non: dans le second l'action de input utilise un canal x qui est différent de celui qui apparaît dans l'interface. Tout canal utilisé par un seal pour lire de son environnement doit déjà exister dans l'environnement où le seal est déclaré, ce qui est une propriété désirable: les noms d'une interface doivent être publics.

En termes de l'exemple de la section 3.4.1 ceci signifie que nous pouvons déclarer qu'une machine x possède l'interface [23 : *telnet*] seulement si le canal nommé 23 et le type *telnet* sont tous deux déjà connus (c'est-à-dire déclarés) dans l'environnement.

Un aspect différent du même phénomène apparaît si l'on essaye de renommer les canaux qui apparaissent dans l'interface d'un seal. Par exemple, le processus

$$(\boldsymbol{\nu} y\colon \mathtt{Id}\ [u\colon M])\,y[\overline{x}^*(z) \mid x^*(u).u^{\uparrow}(v\colon M)\,]$$

n'est pas bien typé car le canal u sur lequel l'input est effectué n'a rien à voir avec le canal u déclaré dans l'interface. Ce qui signifie que les canaux apparaissant dans l'interface d'un seal *actif*, c'est-à-dire un seal qui n'est préfixé par aucune action, ne peuvent pas être liés par une opération. D'autre part considérons

$$x^*(u\colon M).((\boldsymbol{\nu} y\colon \mathtt{Id}\,[u\colon M])\,y[u^{\uparrow}(v\colon M)\,])$$

qui est semblable au processus précédent à la différence près que l'opération de input préfixe le seal (qui n'est donc plus actif). Ce processus est bien typé et il modélise un "générateur" de seal paramétriques dans le canal de leur interface. Placé en parallèle avec, par exemple, $(\boldsymbol{\nu} z\colon M)\overline{x}^*(z)$ il se réduit à

$$(\boldsymbol{\nu} z\colon M)(\boldsymbol{\nu} y\colon \mathtt{Id}\,[z\colon M])\,y[z^{\uparrow}(v\colon M)\,])$$

ce qui correspond à une instance du générateur avec interface $[z\colon M]$.

(Input _): Toute action $x^{\eta}(\vec{y}\colon \vec{M}).P$ lie les variables $\vec{y}$ dans P. Elle doivent donc être ajoutées à l'environnement de typage de P à condition qu'elles aient le même type que celui déclaré pour x. Dans (Input Local) cela est suffisant. Dans (Input Down) nous contrôlons aussi que le nom du seal d'où on veut lire ait bien été déclaré. Dans (Input Up) nous demandons que le canal en question ait bien été inclus dans l'interface du seal courant u.

(Output _): Les règles (Output Local) et (Output Up) se limitent à contrôler que les types des canaux et des arguments coïncident. Dans (Output Down) on contrôle aussi que le canal apparaît, avec le type correct, dans l'interface du seal cible de l'écriture.

(Rcv _): Les règles de mobilité ne diffèrent guère de celles de communication. La seule différence notable est que puisqu'une opération de receive ne lie pas les noms de seal qu'elle spécifie, ces derniers ne sont pas ajoutés à l'environnement utilisé pour typer la continuation.

L'autre point à remarquer est que pour être envoyé sur un canal un seal doit avoir été déclaré comme mobile. Le fait que les attributs de mobilité ne sont pas mémorisés dans les interfaces permet de changer l'attribut de mobilité d'un seal de mobile à immobile, comme dans l'exemple suivant:

$$(\boldsymbol{\nu} x\colon \mathtt{Ch}\ A)(\boldsymbol{\nu} a\colon \mathtt{Id}^{\frown}A)(\boldsymbol{\nu} b\colon \mathtt{Id}^{\underline{\vee}}A)\ \overline{x}^*\{a\}\mid x^*\{b\}\mid a[P\,] \quad \rightarrow \quad (\boldsymbol{\nu} b\colon \mathtt{Id}^{\underline{\vee}}A)\ b[P\,]$$

Ce qui n'est pas possible dans [CGG99].

Canaux multi-partagés

Les modifications à apporter au système précédent pour prendre en compte les canaux multipartagés sont très simples: il suffit d'ajouter pour $\downarrow$ les même règles que pour les communications et les déplacements locaux:

(Output $\downarrow$)

$$\frac{\Gamma \vdash x : \mathtt{Ch}\ \vec{M} \qquad \Gamma \vdash \vec{y} : \vec{M} \qquad \Gamma \vdash_u P}{\Gamma \vdash_u \overline{x}^{\downarrow}(\vec{y}).P}$$

(Input $\downarrow$)

$$\frac{\Gamma \vdash x : \mathtt{Ch}\ \vec{M} \qquad \Gamma, \vec{y}\colon \vec{M} \vdash_u P}{\Gamma \vdash_u x^{\downarrow}(\vec{y}\colon \vec{M}).P}$$

(Snd $\downarrow$)

$$\frac{\Gamma \vdash x : \mathtt{Ch}\ A \qquad \Gamma \vdash y : \mathtt{Id}^{\frown}A \qquad \Gamma \vdash_u P}{\Gamma \vdash_u \overline{x}^{\downarrow}\{y\}.P}$$

(Rcv $\downarrow$)

$$\frac{\Gamma \vdash x : \mathtt{Ch}\ A \qquad \Gamma \vdash y_i : \mathtt{Id}^{Z}A\ {\scriptstyle (i=1..n)} \qquad \Gamma \vdash_u P}{\Gamma \vdash_u x^{\downarrow}\{\vec{y}\}.P}$$

Toutefois, contrairement à ce qui se passe avec les actions vers les enfants dans le système précédent, si une action de write ou de send est effectuée sur $x^{\downarrow}$ le système de type n'assure pas l'existence d'un seal qui écoute (plus précisément qui dans son interface déclare d'écouter) sur $x^{\uparrow}$.

Canaux localisés

Dans le cas de canaux localisés, le fait de collecter dans l'interface d'un seal tout canal qui y apparaît avec une étiquette $\uparrow$ n'est pas très intéressant: les échanges vers le bas se synchronisent sur des canaux locaux à la cible de l'échange, il est donc plus intéressant d'inclure dans l'interface d'un seal ses canaux locaux plutôt que ceux étiquetés par $\uparrow$. En plus dans le système précédent il était intéressant de collecter *tous* les canaux étiquetés par $\uparrow$ car tout canal ainsi étiqueté mais pas inclus dans l'interface serait complètement inutile. Par contre dans la variante à canaux localisés un canal local non inclus dans l'interface peut toujours se synchroniser localement. Donc dans la variante à canaux localisés l'interface d'un seal contient *quelques* canaux locaux sur lesquels le seal s'engage à écouter. Ce qui induit une intéressante interprétation du système de types comme système de contrôle d'accès aux ressources: un seal déclare dans son interface les ressources locales qu'il met à disposition de l'environnement, toute autre ressource locale étant privé.

Les règles qui caractérisent un tel système de types sont vite définies: elles sont les mêmes règles que celle pour les canaux partagés à l'exception près que dans les règles (Input Up) et (Rcv Up) chaque condition $(x{:}\ T) \in A$ est remplacée par la prémisse $\Gamma \vdash x{:}\ \texttt{Ch}\ T$.

3.4.4 Propriétés

Les systèmes de types des différentes variantes décrivent tous des algorithmes de contrôle de types, qui en plus terminent toujours. Leur corrections est établie par la propriété de subject reduction qui est énoncée de la façon suivante:

Théorème 3.4.1 (Subject Reduction) *Pour tout u,P, Q, si* $\Gamma \vdash_u P$ *et* $P \rightarrow Q$ *alors* $\Gamma \vdash_u Q$.

3.5 Conclusion

Dans ce chapitre nous avons vu le Seal calcul, un calcul de mobilité qui par rapport à d'autres formalismes similaires, possède la caractéristique d'avoir été conçu dès le départ pour prendre en compte des aspects de sécurité. Pour des raisons d'espace nous n'avons pas donné des exemples de son utilisation, mais le lecteur intéressé en trouvera plusieurs dans [CGZ01, VC99b, VC99a, Zap00] ainsi qu'une comparaison détaillée avec des travaux similaires. Plusieurs aspects de Seal sont en cours d'investigation ou doivent encore être explorés. Le premier d'entre eux: la théorie équationnelle du Seal Calcul. Nous sommes toutefois confortés dans nos choix par le fait que des récents essais parus dans la littérature de modifier le calcul des Ambients afin de le doter d'une théorie équationnelle intéressante [LS00, MH02] tendent à introduire les primitives du Seal Calcul, primitives qui sont sûrement adaptées à une modélisation des propriétés liées à la sécurité, comme nous le montrons dans le prochain chapitre.

Chapitre 4

Boxed Ambients

Articles de référence: [BCC01c, BCC01a]

Dans le chapitre précédent nous avons vu le Seal Calcul. Il s'agit d'un calcul plus proche d'une implantation que le calcul des Ambients étudié dans les premiers chapitres de cette partie. C'est exactement pour cela que son étude théorique est compliquée. Nous avons vu que le simple fait de devoir gérer des noms de canaux, impose au niveau des types des restrictions significatives. De manière analogue le modèle de mobilité adopté et la possibilité de dupliquer ou de détruire des agents rendent le Seal Calcul plus sécurisé opérationellement et plus riche équationellement que les Ambients, mais aussi plus compliqué à étudier et à utiliser.

Il serait donc plus aisé de travailler avec les ambients mobiles, si à défaut d'avoir des propriétés de sécurité intrinsèques dans leur définition même, ils en permettaient une définition aisée. Malheureusement, comme nous allons le montrer dans la première partie de ce chapitre les Ambients Mobiles ne possèdent pas de primitives qui permettent de caractériser de manière satisfaisante les concepts classiques de sécurité. C'est pourquoi dans la deuxième partie de ce chapitre nous définissons un calcul d'agents mobiles avec des caractéristiques intermédiaires entre les Ambients et les Seals, agents que nous appellerons Ambients Emboîtés ou *Boxed Ambients* (BA). Les BA hériteront des ambients mobiles l'anonymat des communications (aucun canal n'est spécifié) et les primitives de mobilité `in` et `out` (mais pas la `open`) et ils héritent des Seals le modèle de synchronisation des communications à canaux localisés.

4.1 Ambients mobiles et sécurité à plusieurs niveaux

Dans cette section nous allons essayer de définir des politiques mandatoires (c'est-à-dire valables pour tout le système) de contrôle des accès (Mandatory Access Control, ou MAC) dans les ambients mobiles. Dans les modèles classiques de sécurité des accès aux ressources il est coutume de différencier des *sujets* qui peuvent accéder des *objets* par des opérations d'écriture *write* ou de lecture *read*.

Dans un système de sécurité à plusieurs niveaux chaque sujet et chaque objet sont associés à un niveau de sécurité. Les niveaux de sécurité sont ordonnés de sorte à former un treillis. Ceci permet de classifier les accès des sujets aux objets comme des *read-up* (resp. *read-down*) si le sujet accède en lecture à un objet de niveau plus haut (resp. plus bas) et de même pour *write-up* et *write-down*. Ceci couvre aussi le cas d'accès indirect, c'est-à-dire d'accès résultant de la composition d'une suite d'actions atomiques: ainsi l'écriture dans un objet o de niveau non comparable d'une donnée lue dans un objet de niveau supérieur à celui de o est aussi considéré comme un *write-down*.

En utilisant cette classification il est possible de distinguer deux politiques de sécurité mandatoires: la politique dite *militaire* qui interdit les read-up et les write-down (autrement dit, l'information ne peut se propager que du bas vers le haut), et la politique dite *commerciale* qui interdit les read-up et les write-up (autrement dit, les sujets de niveau plus haut possèdent plus de droits).

4.1.1 Ambients mobiles

Dans les chapitres précédents nous avons déjà plusieurs fois rencontré les ambients mobiles et longuement décrit la sémantique de leurs actions de mobilité (voir section 1.2.1 page 74). Pour compléter la description des Ambient Mobiles il ne nous reste à décrire que les communications. Dans le calcul des ambients toute communication est anonyme. La configuration $(\boldsymbol{x})P \mid \langle M\rangle$ représente la composition parallèle de deux processus, le processus d'output $\langle M\rangle$ qui dépose le message M, et le processus d'input $(\boldsymbol{x})P$ qui lit le message M et continue comme $P\{\boldsymbol{x} := M\}$. Dans ce contexte la capabilité `open` joue un rôle fondamental car l'ouverture d'un ambient permet la synchronisation entre les processus situés dans l'ambient ouvrant et l'ambient ouvert. Par exemple la synchronisation entre l'input $(x)P$ et l'output $\langle M\rangle$ dans le système $(x)P \mid \texttt{open}\, b \mid b[\langle M\rangle \mid Q\,]$ est rendue possible par l'action $\texttt{open}\, b$ qui libère le message $\langle M\rangle$.

C'est justement cette interdépendance des communications et des primitives de mobilité qui rend très difficile l'application des modèles classiques de sécurité dans les ambients mobiles. Ceci est assez évident lorsque on considère un exemple concret.

4.1.2 Un problème simple d'accès aux ressources

Supposons que nous ayons un système consistant en un ensemble de ressources $\{r_1, \ldots, r_n\}$ et un agent a qui exécute un programme P et veut accéder à des ressources du système. Pour le contrôle des accès on se référera typiquement à [DoD85] et on implantera un manager de ressource, ce qui dans le calcul des ambients peut être ainsi représenté:

$$a[P\,] \mid m[r_1[\cdots] \mid \cdots \mid r_n[\cdots] \mid R\,]$$

Ici m est le manager de ressources qui exécute un processus R pour contrôler l'accès à chaque ressource. Pour accéder à, disons, r_i, l'agent doit d'abord connaître le nom m du moniteur pour pouvoir y rentrer, ce qui donne

$$m[a[P\,] \mid r_1[\cdots] \mid \cdots \mid r_n[\cdots] \mid R\,]$$

Cette configuration nous montre clairement que R n'a aucun rôle actif dans le système: avec les primitives des Ambients il n'y a rien que R puisse faire car l'interaction entre $a[P\,]$ et r_i ne peut résulter que d'une action autonome de l'agent ou de la ressource [1]. Le rôle de l'ambient m est donc réduit au rôle de son nom: il n'est que le premier mot-de-passe demandé pour l'accès. C'est donc à chaque r_i d'inclure son propre manager pour contrôler et permettre l'accès à son contenu.

Le problème donc peut être ainsi simplifié

Initial configuration: $\quad a[P\,] \mid r[R \mid \langle M\rangle\,]$

ou a veut accéder la ressource r dont le manager est R et dont le contenu est par exemple un message M que l'agent veut lire.

4.1.3 Solutions possibles

Examinons donc les solutions possibles à ce problème de contrôle des accès.

1. L'utilisation des *Ambients Sûrs* [LS00] n'améliorerait pas la situation.

Dissolution d'agent

Une première solution proposé en [CG98] est basée sur le protocole suivant: pour accéder r, l'agent a doit d'abord rentrer dans r:

Enter: $r[R \mid \langle M \rangle \mid a[P]\,]$

L'idée du protocole est que R est le processus $!\mathtt{open}\ p$ qui libère les clients autorisés qui sont rentrés dans la ressource en utilisant un ambient de transport p. Autrement dit, l'obtention de l'accès demandé que le client connaisse le nom de la ressource et le nom du "port" p utilisé pour l'accès. L'agent donc se renommera en p pour adhérer au protocole:

Renaming: $r[!\,\mathtt{open}\,p \mid \langle M \rangle \mid p[(x)P]\,]$

et le manager autorise l'accès en ouvrant p:

Read Access: $r[!\,\mathtt{open}\,p \mid \langle M \rangle \mid p[(x)P]\,] \rightarrow r[!\,\mathtt{open}\,p \mid \langle M \rangle \mid (x)P\,]$

Le protocole est élégant et solide car il demande la connaissance de deux mots-de-passe. Toutefois il présente un certain nombre d'aspects non satisfaisants.

Un premier problème est qu'il parait peu réaliste qu'un agent qui veuille lire une ressource soit prêt à se faire dissoudre. Un problème plus sérieux est que l'ouverture de $p[P]$ peut "polluer" le manager R ou la ressource même car il n'y a aucune manière de savoir ce que P fera un fois libéré: le contenu de p pourrait bien être $N.P$, où N est une suite de in et out qui amènerait r ailleurs en le rendant inaccessible aux autres clients.

Un problème supplémentaire surgit si nous essayons de classifier le protocole selon les principes MAC. L'action du protocole qui permet l'accès à la ressource est faite par le manager qui ouvre l'agent entrant. Puisque le processus contenu dans p est un processus de input on peut classifier cet accès comme un *read* (si p avait contenu un processus de output, il se serait agit d'une *write*). Toutefois affirmer que $\mathtt{open}\ p \mid p[P]$ est un *read* (ou un *write*) est assez contre-intuitif car $p[P]$ plutôt que d'effectuer activement un accès, subit l'action: il parait peu probable que nous puissions nous baser sur une telle notion d'accès. Le problème est que le protocole est strictement lié aux effets de open mais ceci inverse les rôles des participants car c'est le sujet plutôt que l'objet qui est accédé, en fait, ouvert.

Dissolution de ressource

Le problème de la solution précédente pourrait être évité en changeant la perspective. Nous pourrions définir un protocole différent où le rôle actif du sujet est rendu par une combinaison d'open et d'input/output. Ainsi par exemple, le processus $\mathtt{open}\ r.(x)P$ pourrait être interprété dans un protocole comme un accès en lecture à la ressource r. Ceci pourrait fonctionner de manière raisonnable pour les lectures, quoique le fait que l'accès ait comme effet de bord la dissolution de la ressource ne soit pas très convaincant. Mais encore moins convaincante est l'interprétation de $\mathtt{open}\ r.\langle M \rangle$ comme accès en écriture: après avoir dissout r le output $\langle M \rangle$ n'a plus rien à voir avec une écriture sur r.

Agents et messagers

Pour éviter une dissolution indiscriminée lors de l'accès [CG98] suggère une approche différentes basée sur un protocole similaire au précédent mais où les agents ne peuvent jamais être ouverts et ils utilisent pour la communication des ambients spéciaux qui font fonction de messagers. L'idée est d'avoir deux classes de messagers

- *output messenger*: $o[M.\langle N \rangle\,]$; où M est un chemin qui conduit à la location où le message N doit être déposé.

- *input messengers*: $i[M.(x)\, o[M^{-1}.\langle x\rangle\,]\,]$; où M est un chemin qui conduit à la location où une valeur doit être lue. Après lecture le messager parcourt le chemin inverse M^{-1} pour déposer le message qu'il vient de lire.

Ainsi un exemple d'accès en lecture serait:

$$a[\texttt{open}\ o.(x)P \mid i[\texttt{out}\ a.\texttt{in}\ r.(x)o[\texttt{out}\ r.\texttt{in}\ a.\langle x\rangle\,]\,]\,] \;\mid\; r[!\,\texttt{open}\ i \mid \langle N\rangle\,]$$

Le protocole nécessite encore la collaboration du manager de ressource, qui doit ouvrir le *input-messenger*. Une fois de plus il serait contre-intuitif d'affirmer que $\texttt{open}\ i \mid i[P]$ est un accès en lecture. Néanmoins si i pouvait être univoquement identifié comme un *input-messenger*, alors cette classification serait plus réaliste. Le problème est qu'il n'y a pas de moyen syntaxique pour différencier les messagers des agents, de même qu'il n'y a pas moyen de détecter une tentative d'ouvrir un agent "pur". C'est pourquoi la définition des accès dans les ambients ainsi que leur classification précise reste problématique, sinon impossible.

Nous pourrions faire appel à un système de types pour partitionner les ambients entre agents (c'est-à-dire ambients qui ne sont jamais ouverts) et messagers. En utilisant cette partition il serait alors possible de classifier les accès selon les politiques MAC. Toutefois il resterait un problème: considérons par exemple cette configuration initiale

$$a[P' \mid i[M.(x)\, o[M^{-1}.\langle x\rangle\,]\,]\,] \mid r[!\,\texttt{open}\ i \mid \langle N\rangle\,]$$

par une suite de réductions le messager rejoint sa destination où il y est ouvert et il consomme n. À ce stade la structure du système est

$$a[P'\,] \mid r[!\,\texttt{open}\ i \mid o[M^{-1}.\langle N\rangle\,]\,]$$

Ceci code un accès en écriture de r à a. Autrement dit, à chaque *read* de a correspond forcément une *write* de r: si le premier accès est un *read-up* alors le deuxième sera un *write-down*. Donc le protocole a comme effet de mélanger read-up et write-down et, dualement write-up et read-down. Ainsi, par un tel codage il est possible de modéliser la politique militaire mais pas celle commerciale.

4.1.4 Enseignements

Quoique nous ne voyons aucune autre approche significative pour le problème des accès, la liste des solutions présentées pourrait être incomplète. Pour ce qui concerne les approches discutées toutes représentent des solutions partielles et ne sont donc pas adéquates pour exprimer un modèle général de sécurité. Les deux premières semblent artificielles car l'intuition se perd dans le codage, tandis que la dernière récupère partiellement l'intuition sous-jacente mais aux frais d'une vision partielle car elle est incapable d'exprimer la politique *commerciale*.

En conséquence, quoique éventuellement incomplète, notre analyse nous donne des bases pour une conclusion. Certainement, le calcul des ambients permet un contrôle d'accès aux ressources, dans le sens qu'il met à disposition des primitives qui permettent de coder des protocoles. Toutefois le calcul *n'offre pas de support* pour ces mécanismes et politiques car il est dépourvu de primitives permettant une modélisation complète ou naturelle. Nous avons montré qu'il est possible de raisonner au niveau de *protocoles* d'accès, tandis que lorsque nous regardons au niveau des *primitives* d'accès il n'y a aucun principe général que l'on puisse appliquer.

La conclusion est que le *support* pour le contrôle des accès aux ressources avec des Ambients Mobiles nécessite des constructions d'interaction et de communication plus fines ou tout au moins différentes de celles actuelles. Les nouvelles primitives doivent être définies de manière à complémenter les restrictions existantes sur la mobilité des ambients basée sur l'autorisation, sans la dénaturer. En d'autre mots, l'accès à des ressources distantes devrait encore demander de la mobilité, et donc de l'autorisation: les accès locaux par contre devraient être primitifs. Pour voir comment ceci peut être obtenu considérons une fois de plus le protocole basé sur les messagers. Nous pouvons le

redéfinir de manière équivalente ainsi

$$a[\mathtt{in}\ r.i[\mathtt{out}\ a.(x)o[\mathtt{in}\ a.\langle x\rangle\ ]\ ]\mid \mathtt{open}\ o.(x)\mathtt{out}\ r.P\ ]\ \mid\ r[!\ \mathtt{open}\ i\mid\langle M\rangle\ ]$$

En d'autres termes, c'est maintenant l'agent que est responsable du déplacement nécessaire pour rejoindre la ressource, tandis que le messager n'effectue que un `in` et un `out` nécessaires pour l'accès maintenant "local". Après le déplacement de a dans r et de i dehors de a la structure du système (sans a et réplication) est: $r[\mathtt{open}\ i\mid\langle M\rangle\mid i[(x)P\ ]\ ]$. C'est ici que le *read* a lieu. Or au lieu de le coder par un `open` nous pouvons le rendre primitif d'une manière directement inspiré du Seal Calcul, et ainsi éviter l'utilisation de open. Si nous dénotons par $(x)^{\uparrow}$ l'input de l'ambient père, alors l'accès en lecture est simplement $r[\langle M\rangle\mid i[(x)^{\uparrow}P\ ]\ ]$; mais alors tout le protocole peut être simplifié: $a[\mathtt{in}\ r.(x)^{\uparrow}.P\ ]\mid r[\langle M\rangle\ ]$.

Un choix de primitives de communication basé sur cette observation est décrit dans la prochaine section, où nous définissons de manière formelle les *Boxed Ambients*. Nous montrerons que les nouvelles primitives fournissent au calcul des constructions de protection des ressources et de contrôle des accès plus efficaces, sans par ailleurs affecter l'expressivité et l'esprit computationnel des Ambients Mobiles, ni leur élégance.

4.2 Boxed Ambients

Comme nous l'avons anticipé, les BA sont obtenus à partir des Ambients Mobiles en éliminant la capabilité `open` et en ajoutant la possibilité de communication à distance du Seal Calcul à canaux localisés (section 3.1 et suivantes, page 94) à la différence près que les canaux sont anonymes. Par exemple dans

$$n[(x)^{p}P\mid p[\langle M\rangle\mid(x)Q\mid q[\langle N\rangle^{\uparrow}\ ]\ ]\ ]$$

l'ambient n demande de lire la valeur M locale à p, tandis que q essaye d'écrire à son parent la valeur N. L'input $(x)^{p}P$ ne peut se synchroniser que avec l'output $\langle M\rangle$ locale à p. Tandis que $(x)Q$ peut non-déterministiquement se synchroniser soit avec $\langle M\rangle$ soit avec $\langle N\rangle^{\uparrow}$. Bien sûr un éventuel système de types demandera que M et N aient le même type. Toutefois il est intéressant de noter que des échanges de types différents peuvent avoir lieu dans le même ambient sans possibilité de confusion:

$$n[(x)^{p}P\mid(x)^{q}Q\mid p[\langle M\rangle\ ]\mid q[\langle N\rangle\ ]\ ]$$

Les valeurs M et N peuvent très bien avoir des types différents car il n'y a pas de risque de confusion: $(x)^{p}P$ demande une lecture à p tandis que $(x)^{q}Q$ la demande à q. Formellement nous avons les productions suivantes:

Expressions

M	$::=$	$a, b, \ldots$	names
	$\mid$	$x, y, \ldots$	variables
	$\mid$	$\mathtt{in}\ M$	enter M
	$\mid$	$\mathtt{out}\ M$	exit M
	$\mid$	$(M_1, .., M_k)$	tuple, $k \geqslant 0$
	$\mid$	$M.M$	path

Locations

η	$::=$	M	names and variables
	$\mid$	$\uparrow$	enclosing ambient
	$\mid$	$\star$	local

Processes

P	$::=$	$\mathbf{0}$	stop
	$\mid$	$M.P$	action
	$\mid$	$(\boldsymbol{\nu} x)P$	restriction
	$\mid$	$P \mid P$	composition
	$\mid$	$M[P]$	ambient
	$\mid$	$!P$	replication
	$\mid$	$(\boldsymbol{x})^{\eta}P$	patterned input
	$\mid$	$\langle M\rangle^{\eta}P$	synchr. output

Patterns

$\boldsymbol{x}$	$::=$	x	variable
	$\mid$	$\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k$	tuple, $k \geqslant 0$

Patterns et capabilités sont comme dans le calcul des ambients polyadique, sauf que l'action `open` est absente. La congruence structurale est exactement la même que celle des ambients mobiles (voir section 2.2 page 85). Pour ce qui concerne la mobilité celle-ci est régie par les règles classiques des ambients:

$$
\begin{array}{lrcl}
(\textit{enter}) & a[\mathtt{in}\, b.P \mid Q] \mid b[R] & \rightarrow & b[a[P \mid Q] \mid R] \\
(\textit{exit}) & a[b[\mathtt{out}\, a.P \mid Q] \mid R] & \rightarrow & b[P \mid Q] \mid a[R]
\end{array}
$$

La nouveauté réside dans la réduction pour la communication qui imite les règles correspondantes du Seal Calcul:

$$
\begin{array}{lrcl}
(\textit{local}) & (x)P \mid \langle M\rangle Q & \rightarrow & P\{x := M\} \mid Q \\
(\textit{input } n) & (x)^n P \mid n[\langle M\rangle Q \mid R] & \rightarrow & P\{x := M\} \mid n[Q \mid R] \\
(\textit{input} \uparrow) & \langle M\rangle P \mid n[(x)^\uparrow Q \mid R] & \rightarrow & P \mid n[Q\{x := M\} \mid R] \\
(\textit{output } n) & \langle M\rangle^n P \mid n[(x)Q \mid R] & \rightarrow & P \mid n[Q\{x := M\} \mid R] \\
(\textit{output} \uparrow) & (x)P \mid n[\langle M\rangle^\uparrow Q \mid R] & \rightarrow & P\{x := M\} \mid n[Q \mid R]
\end{array}
$$

À tout ceci on doit ajouter les règles pour la réduction sous contexte et pour l'utilisation de la congruence structurale comme celles montrées en section 2.2 page 85.

Le choix des communications du Seal Calcul a des conséquences intéressantes:

- En premier lieu les primitives de communication ont une interprétation immédiate et très naturelle comme accès. Par exemple, l'input $(x)^n$ peut être considéré comme une requêté de lecture du canal anonyme situé dans l'enfant n, ou plus simplement comme un accès en lecture de n. De même $\langle M\rangle^\uparrow$ peut s'interpréter comme un accès en écriture sur le parent.
- En deuxième lieu, nous avons maintenant à notre disposition un support complet et flexible pour la protection des ressources. Un agent qui rentre dans une ressource pour y accéder ne doit pas être ouvert: le manager de la ressource peut servir d'intermédiaire et garder sous contrôle les requêtes de *read* et *write* effectuées par l'agent. Par exemple le problème en section 4.1.2 trouve dans ce cadre une solution assez naturelle et élégante où le manager de ressource m a son rôle originel. Considérons à nouveau la configuration

  $$m[a[P] \mid r_1[\cdots] \mid \cdots \mid r_n[\cdots] \mid R]$$

  où tous les ambients sont boxed et a est rentré dans le manager des ressources. Nous ne devons pas inclure un manager dans chaque ressource car R peut jouer le rôle de médiateur. Par exemple R pourrait être la composition parallèle $R_1 \mid \cdots \mid R_n$ où chaque R_i est le processus $!\,(x)\langle x\rangle^{r_i}$ attendant un output de a pour le faire suivre à la i-ème ressource. D'autres R_i's pourraient être moins généreux et ignorer l'input de a pour lui demander explicitement: $!\,(x)\,{}^a\langle x\rangle^{r_i}$.
- Le modèle de communication épouse bien le modèle de sécurité des ambients basé sur la connaissance de noms pour l'accès en mobilité.
- Enfin une sécurité à plusieurs niveaux peut facilement être modélisée en enrichissant les types par des niveaux de sécurité comme nous l'esquissons en section 4.8.

Enfin, dans la formulation actuelle les output sont synchrones. Pour l'instant nous pouvons considérer l'output asynchrone comme le cas spécial de l'output synchrone où la continuation est le processus **0** ou bien nous inspirer de [Bou92] et formaliser le cas asynchrone en ajoutant la règle suivante: $\langle M\rangle^\eta P \equiv \langle M\rangle^\eta \mid P$. Toutefois nous verrons plus loin que ces deux interprétations ne sont équivalentes que dans le cadre d'un typage "simple" car avec l'introduction des "moded types" (section 4.6) elle ne seront plus satisfaisantes car la première est trop restrictive et la deuxième incorrecte.

4.3 Canaux

Pour illustrer les possibilités des ambients boxed nous allons montrer comment simuler différentes formes de canaux.

π-calcul asynchrone Nous commençons par montrer comment simuler les processus du π-calcul asynchrone $c\langle y\rangle$ (output) et $c(x)P$ (input). L'idée est d'utiliser le boxed ambient $c[!(x)\langle x\rangle\,]$ pour modéliser un canal c. Ceci correspond à un buffer d'une position qui attend un input local et le dépose en output, ce qui donne le codage suivant

$$\begin{array}{lll} \langle\!\langle (\boldsymbol{\nu} x)P \rangle\!\rangle = (\boldsymbol{\nu} x)\langle\!\langle P \rangle\!\rangle & \langle\!\langle !P \rangle\!\rangle = !\langle\!\langle P \rangle\!\rangle & \langle\!\langle c\langle x\rangle \rangle\!\rangle = !c[!(x)\langle x\rangle\,] \mid \langle x\rangle^{c} \\ \langle\!\langle P \mid Q \rangle\!\rangle = \langle\!\langle P \rangle\!\rangle \mid \langle\!\langle Q \rangle\!\rangle & \langle\!\langle \mathbf{0} \rangle\!\rangle = \mathbf{0} & \langle\!\langle c(x)P \rangle\!\rangle = !c[!(x)\langle x\rangle\,] \mid (x)^{c}\langle\!\langle P \rangle\!\rangle \end{array}$$

Bien sûr il y bien d'autres manières de simuler le π-calcul qui en outre satisfont de bonnes propriétés théoriques. Néanmoins nous avons choisi ce codage car il peut être étendu au cas de Seal comme nous le montrons dans la suite.

π-calcul synchrone Soient $cx.P$ et $\bar{c}x.Q$ respectivement les processus de input et output sur un canal c dans le π-calcul *synchrone*. Ces processus peuvent être codés dans le π-calcul *asynchrone* polyadique de la manière suivante: $\bar{c}x.P = (\boldsymbol{\nu} r)c\langle x,r\rangle r()P$, $cy.Q = c(y,r)r\langle\rangle P$ où $r \notin fn(P)$. Autrement dit, le processus de output envoie avec le message un canal privé r où le lecteur peut notifier la réception.

Il est donc possible de coder le π-calcul synchrone en BA simplement en composant les deux codages et en effectuant quelques simplifications:

$$\begin{array}{lcll} \langle\!\langle \bar{c}x.P \rangle\!\rangle & = & (\boldsymbol{\nu} r)(r[()\langle\rangle\,] \mid \langle x,r\rangle^{c}()^{r}\langle\!\langle P \rangle\!\rangle) & r \notin fn(P) \\ \langle\!\langle cy.Q \rangle\!\rangle & = & !c[!(u,v)\langle u,v\rangle\,] \mid (y,s)^{c}\langle\rangle^{s}\langle\!\langle Q \rangle\!\rangle & s \notin fn(Q) \end{array}$$

Le reste est inchangé.

Seal Calcul Il n'est pas trop difficile de réutiliser les codages précédents pour coder la communication du Seal-Calcul. Par simplicité nous ne montrons que le cas asynchrone (output sans continuation) et avec canaux localisés. La seule difficulté est que le codage doit effectuer les déplacements opportuns pour accéder à l'ambient qui code le canal voulu et que la traduction d'un processus dépend du nom n du seal qui le contient:

$$\begin{array}{lcl} \langle\!\langle \bar{c}^{m}(x) \rangle\!\rangle_{n} & = & (\boldsymbol{\nu} p)p[\mathtt{in}\ m.\mathtt{in}\ c.\langle x\rangle^{\uparrow}\,] \\ \langle\!\langle \bar{c}^{\uparrow}(x) \rangle\!\rangle_{n} & = & (\boldsymbol{\nu} p)p[\mathtt{out}\ n.\mathtt{in}\ c.\langle x\rangle^{\uparrow}\,] \\ \langle\!\langle c^{m}(x).P \rangle\!\rangle_{n} & = & (\boldsymbol{\nu} p)p[\mathtt{in}\ m.\mathtt{in}\ c.(x)^{\uparrow}\mathtt{out}\ c.\mathtt{out}\ m.\langle x\rangle\,] \mid (x)^{p}\langle\!\langle P \rangle\!\rangle_{n} \\ \langle\!\langle c^{\uparrow}(x).P \rangle\!\rangle_{n} & = & (\boldsymbol{\nu} p)p[\mathtt{out}\ n.\mathtt{in}\ c.(x)^{\uparrow}\mathtt{out}\ c.\mathtt{in}\ n.\langle x\rangle\,] \mid (x)^{p}\langle\!\langle P \rangle\!\rangle_{n} \\ \langle\!\langle a[P] \rangle\!\rangle_{n} & = & a[\,\langle\!\langle P \rangle\!\rangle_{a}\,] \end{array}$$

Le codage des actions sur des canaux locaux sont celles du codage du π-calcul asynchrone.

4.4 Typage

Un système de types très simple suffit à décrire de manière assez précise le comportement des processus. Les types ambient et processus sont définis comme des constructeurs à deux positions

décrivant les types des échanges qui peuvent avoir lieu localement et avec le parent. Formellement nous avons

Expression Types	W	$::=$	$\mathsf{Amb}[E,F]$	ambient
		$\mid$	$\mathsf{Cap}[E]$	capability
		$\mid$	$W_1 \times \cdots \times W_n$	tuple
Exchange Types	E,F	$::=$	shh	no exchange
		$\mid$	W	exchange
Process Types	T	$::=$	$\mathsf{Proc}[E,F]$	composite exchange

La structure des types est à première vue semblable à celle des systèmes de types pour le calcul des ambients [CG99b, CGG99]. Toutefois leur interprétation est très différente.

$\mathsf{Amb}[E,F]$: ambients qui enferment processus de type $\mathsf{Proc}[E,F]$,

$\mathsf{Cap}[E]$: capabilités exercées à l'intérieur d'ambients ayant des échanges vers le parent de type E,

$\mathsf{Proc}[E,F]$: processus dont les échanges locaux et avec le parent ont type E et F respectivement.

Notons que les types capabilité sont formés par un constructeur unaire et se désintéressent complètement des échanges locaux. La raison pour cela est que grâce à l'absence de `open` nous avons que (i) exercer une capabilité dans un ambient, disons a, ne peut causer que le déplacement de a et (ii) la *safety* d'un déplacement peut être établie indépendamment des échanges locaux de l'ambient.

Pour ce qui concerne les types processus, quelques exemples peuvent aider à expliquer l'intuition des échanges:

$(x\!:\!W)\langle x\rangle$: $\mathsf{Proc}[W,\mathsf{shh}]$. W est échangé (lu et écrit) localement et il n'y a pas d'échange avec le parent.

$(x\!:\!W)^{\uparrow}\langle x\rangle^{n}$: $\mathsf{Proc}[\mathsf{shh},W]$. W est échangé avec le (précisément, lu du) parent et ensuite écrit dans l'ambient n. Il n'y a aucun échange local, d'où le type shh (qui dénote l'absence d'échange, en français chut) en première composante du type processus. Pour que ce typage soit dérivable il faut que n : $\mathsf{Amb}[W,E]$ pour un certain E.

$(x\!:\!W)^{\uparrow}(y\!:\!W')(\langle x\rangle^{n} \mid \langle y\rangle)$: $\mathsf{Proc}[W',W]$. W est échangé (lu) avec le parent et ensuite expédié à n, tandis que W' est échange (lu et écrit) localement. Ce typage demande que n : $\mathsf{Amb}[W,E]$ pour un certain E.

$(x\!:\!W)\langle x\rangle^{\uparrow}$: $\mathsf{Proc}[W,W]$. W est lu localement et écrit dans le parent.

Ces exemples nous donnent déjà un premier aperçu de la flexibilité des BA: comme les ambients mobiles, les BA sont des “endroits de conversation” mais à la différence de ces derniers ils permettent plus d'un “sujet” de conversation, car chaque ambient peut échanger des valeurs de types différents avec chacun de ses enfants.

Définition 4.4.1 [*Sous-typage*] Nous dénotons par $\leqslant$ le plus petit pré-ordre sur les types échange tel que $\mathsf{shh} \leqslant E$ pour tout type échange E. Cette relation est transposée aux types processus ainsi: $\mathsf{Proc}[\mathsf{shh},F] \leqslant \mathsf{Proc}[E,F]$. □

Le sous-typage pour les types processus est utilisé en conjonction avec une règle de subsumption, ce qui n'est pas le cas pour le sous-typage des types échange. L'idée est que l'absence d'échanges, dénotée par shh, est toujours compatible avec une situation où des échanges sont attendus. Il serait tentant d'étendre la relation de sous-typage pour les types processus aussi à la composante qui enregistre les échanges avec le parent. Toutefois, comme montré en détails dans [BCC01a], un tel choix ne serait pas correct.

Typage des Expressions

$$\text{(PROJECTION)}\ \frac{\Gamma(n) = W}{\Gamma \vdash n : W} \qquad \text{(TUPLE)}\ \frac{\Gamma \vdash M_i : W_i \quad \forall i \in 1..k}{\Gamma \vdash (M_1, ..., M_k) : W_1 \times ... \times W_k} \qquad \text{(PATH)}\ \frac{\Gamma \vdash M_1 : \mathsf{Cap}[F] \quad \Gamma \vdash M_2 : \mathsf{Cap}[F]}{\Gamma \vdash M_1.M_2 : \mathsf{Cap}[F]}$$

$$\text{(IN)}\ \frac{\Gamma \vdash M : \mathsf{Amb}[F, E] \quad F' \leqslant F}{\Gamma \vdash \mathtt{in}\ M : \mathsf{Cap}[F']} \qquad \text{(OUT)}\ \frac{\Gamma \vdash M : \mathsf{Amb}[E, F] \quad F' \leqslant F}{\Gamma \vdash \mathtt{out}\ M : \mathsf{Cap}[F']}$$

Les règles (PROJECTION), (TUPLE), et (PATH) sont standard. Les règles (IN) et (OUT) définissent les contraintes pour une mobilité sûre par rapport aux types et expliquent pourquoi les types capabilité ne contiennent qu'une composante. L'idée est la suivante: prenons une capabilité quelconque, par exemple $\mathtt{in}\ n$ où $n : \mathsf{Amb}[F, E]$, et supposons que cette capabilité est exercée à l'intérieur d'un ambient, disons, m. Si les échanges de m vers son père sont de type F', alors $\mathtt{in}\ n : \mathsf{Cap}[F']$. Pour que le déplacement de m dans n soit sûr par rapport aux types, il faut que le type des échanges locaux à n soit égal au type F' des échanges de m vers son père. En fait nous pouvons être moins restrictifs car si m n'a aucun échange avec son parent alors $F' = \mathsf{shh} \leqslant F$, et m peut tranquillement rentrer dans n. Un raisonnement similaire s'applique à la règle (OUT): les échanges de l'ambient sortant avec son parent doivent avoir un type $\leqslant$-compatible avec le type des échanges vers le parent de l'ambient d'où il sort.

Typage des Processus

$$\text{(DEAD)}\ \frac{}{\Gamma \vdash \mathbf{0} : T} \qquad \text{(NEW)}\ \frac{\Gamma, x : W \vdash P : T}{\Gamma \vdash (\boldsymbol{\nu} x{:}W)P : T} \qquad \text{(PARALLEL)}\ \frac{\Gamma \vdash P : T \quad \Gamma \vdash Q : T}{\Gamma \vdash P \mid Q : T}$$

$$\text{(PREFIX)}\ \frac{\Gamma \vdash M : \mathsf{Cap}[F] \quad \Gamma \vdash P : \mathsf{Proc}[E, F]}{\Gamma \vdash M.P : \mathsf{Proc}[E, F]} \qquad \text{(AMB)}\ \frac{\Gamma \vdash M : \mathsf{Amb}[E, F] \quad \Gamma \vdash P : \mathsf{Proc}[E, F]}{\Gamma \vdash M[P] : \mathsf{Proc}[F, G]}$$

$$\text{(SUBSUM PROC)}\ \frac{\Gamma \vdash P : T \quad T \leqslant T'}{\Gamma \vdash P : T'} \qquad \text{(REPLICATION)}\ \frac{\Gamma \vdash P : T}{\Gamma \vdash !P : T}$$

(DEAD), (NEW), (PARALLEL), (REPLICATION) et la règle de subsumption sont standard. Dans (PREFIX) le typage de la capabilité M assure via les règles (IN), (OUT), et (PATH) montrées auparavant que les échanges locaux de tout ambient traversé suite à l'utilisation de M est compatible avec les échanges vers le parent de l'ambient déplacé par M.

La règle (AMB) établit les conditions que P doit satisfaire pour pouvoir être enfermé dans M: les échanges de P doivent avoir les mêmes types E et F que ceux déclarés pour M. En fait P peut aussi être localement silencieux et le type dérivable par subsumption. Par contre la règle n'impose aucune contrainte sur le type des échanges avec le parent.

$$\text{(INPUT} \star\text{)}\ \frac{\Gamma, x{:}W \vdash P : \mathsf{Proc}[W, E]}{\Gamma \vdash (x{:}W)P : \mathsf{Proc}[W, E]} \qquad \text{(OUTPUT} \star\text{)}\ \frac{\Gamma \vdash M : W \quad \Gamma \vdash P : \mathsf{Proc}[W, E]}{\Gamma \vdash \langle M \rangle P : \mathsf{Proc}[W, E]}$$

$$\text{(INPUT } M\text{)}\quad \frac{\Gamma \vdash M : \mathsf{Amb}[W, E] \quad \Gamma, x{:}W \vdash P : T}{\Gamma \vdash (x{:}W)^M P : T}$$

$$\text{(OUTPUT } M\text{)}\quad \frac{\Gamma \vdash M : \mathsf{Amb}[W, E] \quad \Gamma \vdash N : W \quad \Gamma \vdash P : T}{\Gamma \vdash \langle N \rangle^M P : T}$$

$$\text{(INPUT } \uparrow\text{)}\quad \frac{\Gamma, x{:}W \vdash P : \mathsf{Proc}[E, W]}{\Gamma \vdash (x{:}W)^{\uparrow} P : \mathsf{Proc}[E, W]}$$

$$\text{(OUTPUT } \uparrow\text{)}\quad \frac{\Gamma \vdash M : W \quad \Gamma \vdash P : \mathsf{Proc}[E, W]}{\Gamma \vdash \langle M \rangle^{\uparrow} P : \mathsf{Proc}[E, W]}$$

Dans tous les cas de input/output le type de l'échange doit être compatible avec le type local des échanges. Notons que dans les échanges avec les enfants —règles (INPUT M) et (OUTPUT M)— aucune contrainte n'est imposée sur les types des échanges.

Ces règles de typage assurent que les communications dans et entre les ambients ne produisent jamais d'erreur de type, ce qui est une conséquence de la propriété de subject reduction satisfaite par le système

Théorème 4.4.2 (Subject Reduction) *Si* $\Gamma \vdash P : T$ *et* $P \rightarrow Q$ *alors* $\Gamma \vdash Q : T$.

4.5 Comparaison entre les boxed ambients et les ambients mobiles

Ayant défini le système de types nous pouvons considérer l'impact qu'il a sur la mobilité et les communications des ambients et le comparer avec celui des ambients mobiles.

Nous avons déjà remarqué que la correction d'une configuration ne dépend pas du type des échanges internes. Par contre les communications avec un parent imposent des restrictions à la mobilité puisque des ambients dont les communications vers le parent sont de type W ne peuvent traverser que des ambients avec des échanges locaux de type W. Toutefois il est facile de voir qu'une telle restriction est toujours moins sévère que celle imposée par le typage des ambients mobiles.

En fait si nous considérons le sous-système où tout échange avec un parent serait interdit (autrement dit où tout type ambient est de la forme $\mathsf{Amb}[E, \mathsf{shh}]$) nous pouvons noter que:

- *La mobilité des ambients boxed est aussi flexible que celle des ambients mobiles*. En fait il est facile de voir que grâce aux règles (IN) et (OUT) un ambient dont les communications avec le parent seraient inhibées peut se déplacer librement indépendemment de son type et du type de l'ambient de sa destination.
- *La communication des ambients boxed est plus flexible que celle des ambients mobiles*. Le fait que les communications d'un ambient vers son parent ne soient pas possibles ne signifie pas qu'un tel ambient ne communique pas: une communication avec le parent peut être établie par ce dernier. Plus généralement tout ambient peut utiliser une combinaison de mobilité et de communications vers ses enfants pour communiquer avec un autre ambient car il peut accéder aux canaux locaux des enfants et de tout ambient entrant: en plus, tous ces échanges peuvent être de types différents. En outre le codage des canaux en section 4.3 peut être utilisé pour effectuer des échanges locaux de types différents, car l'ambient $c[!(x{:}W)\langle x \rangle\,]$ peut être considéré comme un canal local de type W.

  Dans les ambients mobiles au contraire une communication entre un parent et un enfant demande l'ouverture de l'enfant (ou d'un ambient "messager" sortant de l'enfant). Par conséquence un parent ne peut communiquer avec ses enfants que s'ils ont tous le même type local.

Cela dit, il est clair que l'inhibition des échanges vers le parent ne constitue pas une solution car nous ne pouvons pas par exemple typer les ambients de transport $p[\cdots]$ utilisés dans le codage des

canaux du seal calcul (section 4.3). En réalité il est possible d'étendre le système de types précédent pour permettre une interaction plus flexible entre la mobilité et la communication vers le parent, comme nous le montrons dans la section suivante.

4.6 Moded Types

Nous avons déjà souligné que dans le système de types précédent un ambient qui effectue des échanges de type W vers le haut ne peut traverser que des ambients dont les échanges locaux sont aussi de type W. Les ambients de transport $p[\cdots]$ dans le codage de canaux du Seal Calcul en section 4.3 page 115 effectuent tous des échanges locaux, ce qui signifie qu'une version typée de ce codage forcerait tous les ambients participant à une communication à partager le même type pour les échanges locaux. Toutefois si nous regardons de près le comportement de ces ambients de transport nous nous rendons compte qu'il s'agirait d'une précaution inutile car ces ambients n'effectuent aucune communications vers le haut en dehors de l'ambient qui modélise le canal. C'est pourquoi ils ne peuvent pas interférer avec les communications locales des ambients qu'ils traversent. Grâce à cette observation nous pouvons modifier le système de types précédent afin d'éviter cette restriction superflue.

L'idée est assez simple quoique sa mise en oeuvre soit techniquement compliquée. Elle se base sur le fait que lors qu'ils sont silencieux vers le haut les ambients peuvent se déplacer librement indépendemment de tout type, tandis que lors qu'ils effectuent des échanges vers le haut ils doivent se trouver dans un ambient dont les échanges locaux sont de type approprié. L'idée est donc de distinguer de manière précise les phases où un ambient est silencieux vers le haut de celles où il effectue des échanges vers son parent et d'imposer des contraintes de types seulement pendant ces dernières. Les ambients pour lesquels il est possible de détecter un tel comportement (comme par exemple les ambients de transport du codage des canaux de Seal) seront typés par un nouveau constructeur de type: $\mathsf{Amb}^\circ[E,F]$. Plus précisément un ambient de type $\mathsf{Amb}^\circ[E,F]$ est un ambient qui effectue des échanges locaux de type E, se déplace silencieusement (par rapport au parent) jusqu'à arriver dans un ambient de type approprié où il effectuera des échanges vers le haut de type F. Un tel comportement est déterminé en examinant le processus contenu dans l'ambient et en vérifiant que pendant son exécution son type alternera différentes modalités. En particulier nous distinguons pour un type processus trois modalités différentes:

$\mathsf{Proc}[E,{}^\bullet W]$: processus silencieux vers le haut avec des échanges locaux de type E. Le type W signale qu'un processus de ce type peut s'exécuter de manière sure en parallèle avec un processus qui effectue des échanges vers le haut de type W

$\mathsf{Proc}[E,{}^\circ W]$: processus avec des échanges locaux de type E et des échanges vers le haut de type W. Les échanges vers le haut sont temporairement inactifs car le processus est en train de se déplacer.

$\mathsf{Proc}[E,{}^\vartriangle W]$: processus avec des échanges locaux de type E et qui après avoir effectué des échanges vers le haut de type W évoluera en un processus de type $\mathsf{Proc}[E,{}^\circ W]$ ou $\mathsf{Proc}[E,{}^\vartriangle W]$.

L'exposition détaillée de ce système de types est trop étendue pour pouvoir être incluse dans ce rapport, la principale difficulté technique étant la gestion de la composition parallèle de processus de modalités différentes et la vérification des types pendant les phases non silencieuses. Tous ces détails peuvent être trouvés dans [BCC01a].

Pour ce rapport il suffira de retenir que les type Amb° permettent de typer les ambients de transport du codage d'un canal c de type W en Seal ainsi:

$$
\begin{array}{lcl}
\langle\!\langle\, \overline{c}^m(x) \,\rangle\!\rangle_n & = & (\boldsymbol{\nu} p\colon \mathsf{Amb}^\circ[\mathsf{shh}, W])p[\mathtt{in}\ m.\mathtt{in}\ c.\langle x\rangle^\uparrow\,] \\
\langle\!\langle\, \overline{c}^\uparrow(x) \,\rangle\!\rangle_n & = & (\boldsymbol{\nu} p\colon \mathsf{Amb}^\circ[\mathsf{shh}, W])p[\mathtt{out}\ n.\mathtt{in}\ c.\langle x\rangle^\uparrow\,] \\
\langle\!\langle\, c^m(x\colon W).P \,\rangle\!\rangle_n & = & (\boldsymbol{\nu} p\colon \mathsf{Amb}^\circ[W, W])p[\mathtt{in}\ m.\mathtt{in}\ c.(x\colon W)^\uparrow \mathtt{out}\ c.\mathtt{out}\ m.\langle x\rangle\,] \mid (x\colon W)^p \langle\!\langle\, P \,\rangle\!\rangle_n \\
\langle\!\langle\, c^\uparrow(x\colon W).P \,\rangle\!\rangle_n & = & (\boldsymbol{\nu} p\colon \mathsf{Amb}^\circ[W, W])p[\mathtt{out}\ n.\mathtt{in}\ c.(x\colon W)^\uparrow \mathtt{out}\ c.\mathtt{in}\ n.\langle x\rangle\,] \mid (x\colon W)^p \langle\!\langle\, P \,\rangle\!\rangle_n
\end{array}
$$

ceci de manière indépendante des types de m et de n.

4.7 Communications Asynchrones

Dans [Car99a] Cardelli note qu'une computation mobile distribuée ne peut pas se baser uniquement sur des communications synchrones, ce qui est d'ailleurs confirmé par la difficulté d'implanter le synchronisme dans des premières implantations [BV02, FLS00].

Nous avons vu dans la section précédente qu'il est possible de typer des exemples plus significatifs en détectant de façon précise le comportement des ambients qui alternent des phases de communication vers le parent avec des phases pendant lesquelles l'ambient est silencieux vers le haut et peut donc se déplacer librement. Nous n'avons pas pu donner les détails mais à cette alternance de phases correspondent des types différents du processus qui est exécuté par l'ambient: tant que ce processus effectue des communication vers le haut il se trouve dans un état de mobilité plus contrainte, dénoté par un type $^\vartriangle$. Une fois toute communication vers le parent achevée, l'ambient rentre dans une phase silencieuse —dénoté par un type $^\bullet$— où il possède une totale liberté de mouvement.

Tout ceci est possible car toute communication est synchrone. Nous sommes donc capables de dire quand une communication sera terminée. Par contre, en présence de communications asynchrone des problèmes surgissent. Dans [Bou92] Boudol montre qu'il existe au moins deux manières équivalentes d'obtenir un calcul asynchrone à partir du π-calcul synchrone: (i) soit en imposant que tout processus de output ait comme continuation le processus $\mathbf{0}$ (ii) soit en ajoutant la règle de congruence structurale $\langle M\rangle^\eta P \equiv \langle M\rangle^\eta \mathbf{0} \mid P$. Aucune des deux solutions n'est satisfaisante en présence de moded types. Avec la première solution on s'interdit de pouvoir passer à une phase silencieuse après un output, car rien ne peut suivre un output $\langle M\rangle^\uparrow$ sinon $\mathbf{0}$. Et pour ce qui concerne la deuxième solution elle n'est pas correcte (sound) par rapport au typage car si P est un processus silencieux (type $^\bullet$) alors $\langle M\rangle^\uparrow P$ est un processus qui après un pas devient silencieux (et donc typable par $^\vartriangle$), mais il n'est pas possible de déduire la même chose pour $\langle M\rangle^\uparrow \mathbf{0} \mid P$.

Toutefois le problème n'est pas du à l'asynchronie en elle même, mais plutôt aux codages particuliers considérés. Reprenons le cas du processus silencieux P qui en tant que tel peut exécuter toute action de mouvement sans aucune contrainte. Considérons maintenant $\langle M\rangle^\uparrow \mathbf{0} \mid P$. Il est clair qu'avant d'effectuer les actions de mobilité de P (qui peuvent déplacer l'ambient courant n'importe où) il faut être sûr que M ait été délivré au parent (qui doit avoir le type approprié). Mais peu importe de savoir si une synchronisation entre $\langle M\rangle^\uparrow$ et une lecture dans le parent a eu lieu. La seule chose importante est d'être sûr que si jamais une telle synchronisation aura lieu, elle aura lieu dans le parent où l'ambient se trouvait avant d'exécuter P. Cette assurance peut être obtenue par les règles de réduction. Il existe au moins trois modifications possibles de la sémantique opérationnelle pour obtenir une version asynchrone des boxed ambients qui soit compatible avec les moded types (voir [BCC01a]). L'une d'elle est d'utiliser les règles suivantes

(*asynch output* $\star$)	$\langle M\rangle P$	$\rightarrow$	$\langle M\rangle \mid P$
(*asynch output* n)	$\langle M\rangle^n P \mid n[Q\,]$	$\rightarrow$	$P \mid n[\langle M\rangle \mid Q\,]$
(*asynch output* $\uparrow$)	$n[\langle M\rangle^\uparrow P \mid Q\,]$	$\rightarrow$	$\langle M\rangle \mid n[P \mid Q\,]$
(*asynch input* $\star$)	$(x)P \mid \langle M\rangle$	$\rightarrow$	$P\{x\colon = M\}$
(*asynch input* n)	$(x)^n P \mid n[\langle M\rangle \mid Q\,]$	$\rightarrow$	$P\{x\colon = M\} \mid n[Q\,]$
(*asynch input* $\uparrow$)	$\langle M\rangle \mid n[(x)^\uparrow P \mid Q\,]$	$\rightarrow$	$n[P\{x\colon = M\} \mid Q\,]$

selon lesquelles un output libère sa continuation après avoir été déposé à la place appropriée à une synchronisation locale.

Il faut toutefois noter qu'une telle implantation nous fait perdre la propriété de médiation typique des communications du Seal Calcul, et laisse la porte ouverte à la présence de "covert channels". Par exemple dans $b[a[(x\colon W)^{\uparrow}P] \mid c[\langle M\rangle^{\uparrow}Q]]$ les deux ambients a et c peuvent communiquer sans que b serve d'intermédiaire à cette communication ou qu'il s'en aperçoive. En fait une telle configuration se réduit en deux pas à $b[a[P\{x\colon = M\}] \mid c[Q]]$.

4.8 Sécurité

Dans les boxed ambients il est naturel de considérer que la ressource d'un ambient est son canal anonyme local. Nous avons donc une claire distinction entre les sujets (les ambients) et les objets (les canaux locaux). Nous pouvons en plus différencier deux types d'accès, en lecture et en écriture, ce qui induit la classification suivante

$m[\langle M\rangle^n P \mid n[Q] \mid R]$	m accède le canal situé dans n en "write mode"
$m[(x)^n P \mid n[Q] \mid R]$	m accède le canal situé dans n en "read mode"
$m[P \mid n[\langle M\rangle^{\uparrow}Q \mid R]]$	n accède le canal situé dans son parent en "write mode"
$m[R \mid n[(x)^{\uparrow}P \mid Q]]$	n accède le canal situé dans son parent en "read mode"

Nous pouvons donc affecter à chaque sujet et à chaque objet un niveau de sécurité dans un treillis et utiliser ces affectations pour définir des politiques de sécurité. Le système de types peut en outre être utilisé pour assurer statiquement le respect de telles politiques. Le traitement détaillé d'un tel système n'est pas du ressort de ce rapport et peut être consulté dans [BCC01a, BCC01b]. Nous voulons toutefois donner l'idée de comment cela peut être fait.

D'abord nous enrichissons les types par un niveau de sécurité σ choisi parmi les éléments d'un treillis, et par une modalité d'accès $\mathscr{A}$ choisie parmi $\{\mathsf{w}, \mathsf{r}, \mathsf{rw}, \mathsf{shh}\}$ (indiquant respectivement un accès en écriture, en lecture, en écriture et lecture, et aucun accès). Ainsi $\sigma\mathsf{Amb}[E, F^{\mathscr{A}}]$ typera les ambients de niveau σ qui renferment des processus dont les échanges locaux sont de type E et les échanges sont de type F et modalité $\mathscr{A}$.

Lors du typage des accès vers un parent les règles de typage déduiront les modalités appropriées:

(INPUT $\uparrow$)
$$\frac{\Gamma, x\colon W \vdash P : \sigma\mathsf{Proc}[F, W^{\mathscr{A}}] \quad \mathscr{A} \in \{\mathsf{r}, \mathsf{rw}\}}{\Gamma \vdash (x\colon W)^{\uparrow}P : \sigma\mathsf{Proc}[F, W^{\mathscr{A}}]}$$

(OUTPUT $\uparrow$)
$$\frac{\Gamma \vdash M : W \quad \Gamma \vdash P : \sigma\mathsf{Proc}[F, W^{\mathscr{A}}] \quad \mathscr{A} \in \{\mathsf{w}, \mathsf{rw}\}}{\Gamma \vdash \langle M\rangle^{\uparrow}P : \sigma\mathsf{Proc}[F, W^{\mathscr{A}}]}$$

Lors de l'accès à la ressource d'un fils le typage vérifie que la politique de sécurité est respectée. Ainsi dans le cas de sécurité militaire nous aurons les règles suivantes

(INPUT M)
$$\frac{\Gamma, x : W \vdash P : \sigma\mathsf{Proc}[E, {}^{\mu}F^{\mathscr{A}}] \quad \Gamma \vdash M : \rho\mathsf{Amb}^{?}[W, U^{\mathscr{B}}] \quad \rho \preceq \sigma}{\Gamma \vdash (x : W)^{M}P : \sigma\mathsf{Proc}[E, {}^{\mu}F^{\mathscr{A}}]}$$

(OUTPUT M)
$$\frac{\Gamma \vdash N : W \quad \Gamma \vdash P : \sigma\mathsf{Proc}[E, {}^{\mu}F^{\mathscr{A}}] \quad \Gamma \vdash M : \rho\mathsf{Amb}^{?}[W, U^{\mathscr{B}}] \quad \sigma \preceq \rho}{\Gamma \vdash \langle N\rangle^{M}P : \sigma\mathsf{Proc}[E, {}^{\mu}F^{\mathscr{A}}]}$$

Où $\mathsf{Amb}^{?}$ indique que la règle est valable pour Amb et $\mathsf{Amb}^{?}$et μ est n'importe quelle modalité. La vérification du respect des politiques de sécurité par les accès vers le haut sera effectuée lors du typage de l'ambient parent.

4.9 Exemples

Voyons un certain nombre d'exemples pris dans la littérature sur le sujet et montrons comment les gérer par les boxed ambients.

4.9.1 Wrappers

Comme solution pour la protection de ressources et le contrôle d'accès, Sewell et Vitek [SV00] proposent l'utilisation de *wrappers* pour isoler des programmes des éventuels attaquants. Ceci est fait dans une extension du π-calcul inspirée pour ce qui concerne la structuration et les communications par le Seal-calcul: le code peut être isolé dans une "boite" qui en limite les communications avec l'environnement qui l'entoure. L'exemple paradigmatique de leur travail est le suivant

$$(\nu a, b)\ \big(a[P] \mid !(x)^a\langle x\rangle^b \mid b[Q]\big)\ .$$

P et Q sont des processus arbitraires emboîtés dans des "boxes" (en fait des ambients ou seals) avec des noms privés a et b en parallèle avec un processus qui fait suivre les messages de a à b. Cette configuration de *wrapping* est intéressante lorsque P et Q sont des processus potentiellement nocifs, car les frontières des boites leur empêchent une communication directe, tandis que les restrictions assurent que les seules interactions possibles avec l'environnement passent par le processus $!(x)^a\langle x\rangle^b$. De cette manière le formalisme de [SV00] empêche (i) que Q fasse suivre des secrets à P et (ii) que P et Q puissent corrompre l'environnement. Ces propriétés bien sûr sont aussi valables quand on considère le même exemple comme une expression dans les boxed ambients. Mais en plus avec les boxed ambients d'autres choix sont aussi possibles. Par exemple pour imposer (i) nous pouvons utiliser la sécurité militaire et assurer une propriété plus générale: si nous affectons à a un niveau de sécurité strictement supérieur au niveau de b alors nous avons la certitude qu'aucun accès de Q à P ne pourra avoir lieu indépendemment du contexte dans lequel ils se trouvent. Pour imposer aussi (ou seulement) la propriété (ii) nous pouvons une fois de plus utiliser la sécurité militaire (mais aussi la commerciale) en affectant à l'environnement un niveau de sécurité incompatible avec les niveaux de a et de b. De cette manière les processus ne pourront jamais accéder et corrompre les ressources de l'environnement.

4.9.2 Firewalls

Examinons le protocole de franchissement de firewalls défini par [CG99b] et peaufiné par [LS00] et montrons comment le définir avec les Boxed Ambients. L'idée du protocole est de laisser un *Agent* traverser un *Firewall* par le biais d'une clef partagée k.

$$\mathit{Firewall} = (\nu f)f[k[\mathtt{out}\ f.\langle \mathtt{in}\ f\rangle^a] \mid \ldots] \qquad \mathit{Agent} = a[\mathtt{in}\ k.(x)\mathtt{out}\ k.x.Q]$$

$$\begin{array}{lll}(\nu k)(\mathit{Firewall} \mid \mathit{Agent}) & \to^* & (\nu k, f)f[\ldots] \mid k[\langle \mathtt{in}\ f\rangle^a \mid a[(x)\mathtt{out}\ k.x.Q]] \\ & \to^* & (\nu k, f)f[\ldots] \mid k[a[\mathtt{out}\ k.\mathtt{in}\ f.Q]] \\ & \to^* & (\nu f)f[\ldots \mid a[Q]]\end{array}$$

Le *Firewall*, dénoté par le nom secret f, envoie un ambient pilote k pour guider l'agent à l'intérieur. Le nom k est le password que l'agent a doit connaître pour pouvoir entrer dans le firewall.

Outre l'authentification des agents entrants, un firewall doit en général fournir d'autres garanties de sécurité. Par exemple l'administrateur du firewall peut vouloir assurer que les processus résidant dans le firewall puissent accéder aux ressources de l'agent mais pas le contraire. Ceci peut être obtenu par le biais d'une sécurité commerciale et l'affectation des niveaux de sécurité suivants: $f : \phi\mathsf{Amb}[E, F^{\mathscr{A}}]$ et $k : \kappa\mathsf{Amb}[\mathsf{shh}, \mathsf{shh}]$, où E et $F^{\mathscr{A}}$ sont des types appropriés et ϕ et κ sont des niveaux de sécurité tels que $\kappa \prec \phi$.

Pour voir les effets d'une telle affectation considérons un agent générique a (dont la définition peut différer de celle de *Agent*) qui veuille entrer dans les firewall et ait le type $\alpha\mathsf{Amb}[G, H^{\mathscr{B}}]$. Pour

traverser le firewall a doit se conformer au protocole et donc accepter l'accès en écriture de k. Avec la sécurité commerciale ceci est possible seulement si $\alpha \preceq \kappa$ ce qui par transitivité implique $\alpha \prec \phi$. En outre la politique commerciale interdit à un ambient de niveau inférieur (tel que a) contenu dans un ambient de niveau supérieure (tel f) d'effectuer des communications vers le haut donc $\alpha \prec \phi$ implique $\mathscr{B} = \mathsf{shh}$. En conclusion les types du firewall et de son pilote imposent à tout agent a voulant entrer dans le firewall de posséder un niveau de sécurité strictement inférieur que celui de f; ainsi la politique de sécurité assure qu'une fois entré l'agent ne pourra pas directement accéder aux ressources situées à l'intérieur du firewall.

4.9.3 Chevaux de Troie

Dans le chapitre III.2 nous avons défini un système de types pour détecter statiquement des chevaux de Troie. L'exemple motivant le travail était

$$a[\mathtt{in}\, c.P\,] \mid b[\mathtt{in}\, a.\mathtt{out}\, a.\mathtt{in}\, d.Q\,] \mid c[R \mid d[S\,]\,]$$

où d contenait des données confidentielles. La question était si c devait laisser entrer a. Et la sécurité était assurée par le système de types qui détectait tout ambient entrant dans c et pouvant accéder à d. Ici nous pouvons obtenir le même dégré de sécurité tout simplement en affectant à d un niveau de sécurité qui soit incomparable avec tout niveau de sécurité défini à l'extérieur de c. Il est intéressant de noter qu'indépendamment du fait que le nom d ait été communiqué vers l'extérieur et que c se trouve dans un contexte bien typé ou pas, la sécurité commerciale ne laissera accéder à d que des processus déjà en c. En fait contrairement à ce qui se passe dans les ambients mobiles, communiquer le nom d'un ambient ne correspond pas a donner la permission d'accéder à ses ressources.

Il faut noter qu'aucun des exemples présentés jusqu'à ici n'utilise des communications vers le haut, ce qui signifie que le système de types en section 4.4 ou même sa version simplifiée en section 4.5 suffisent. Nous présentons ci dessous un exemple qui demande l'utilisation des *moded types*.

4.9.4 Sécurité du langage distribué

Pour donner un exemple de l'expressivité des boxed ambients et de leurs politiques de sécurité nous donnons le codage du langage distribué jouet de [CGG00] dont la syntaxe est définie ci-dessus (pour le typage consulter [CGG00]).

Types	$W ::= \mathit{Node}$	type of nodes
	$\mid \mathit{Ch}(W)$	type of channels
Network	$\mathit{Net} ::= (\boldsymbol{\nu} x{:}\, W)\mathit{Net}$	restriction
	$\mid \mathit{Net} \mid \mathit{Net}$	network composition
	$\mid \mathit{node}\ n[\mathit{Cro}]$	node
Crowd	$\mathit{Cro} ::= (\boldsymbol{\nu} x{:}\, W)\mathit{Cro}$	restriction
	$\mid \mathit{Cro} \mid \mathit{Cro}$	crowd composition
	$\mid \mathtt{channel}\ c$	channel
	$\mid \mathtt{thread}\ [\mathit{Th}]$	thread
Threads	$\mathit{Th} ::= \mathsf{go}\ n.\mathit{Th}$	migration
	$\mid \overline{c}(x)$	output to a channel
	$\mid c(x).\mathit{Th}$	input from a channel
	$\mid \mathtt{fork}\ (\mathit{Cro}).\mathit{Th}$	fork a new crowd
	$\mid \mathtt{spawn}\ n[\mathit{Cro}].\mathit{Th}$	spawn a new node

Pour le codage nous n'avons pas besoin de l'entière puissance des boxed ambients. Des ambients sans communications vers le haut suffiront la plupart du temps. Dans ce cas nous écrirons $\sigma\mathsf{Amb}[E]$ pour $\sigma\mathsf{Amb}[E, \mathsf{shh}^{\mathscr{A}}]$ en omettant le niveau de sécurité σ quand il n'est pas important. Nous utilisons aussi *Synch* pour dénoter le n-uplet vide. Le codage est ainsi défini (où tous les types ont le même niveau de sécurité).

$$
\begin{array}{lcl}
\textit{Types}: & & \\
\langle\!\langle \textit{Node} \rangle\!\rangle & = & \mathsf{Amb}[\mathsf{shh}] \\
\langle\!\langle \textit{Ch}(W) \rangle\!\rangle & = & \mathsf{Amb}[\langle\!\langle W \rangle\!\rangle] \\[1ex]
\textit{Net}: & & \\
\langle\!\langle (\boldsymbol{\nu} x\colon \textit{Node})\textit{Net} \rangle\!\rangle_{\Gamma} & = & (\boldsymbol{\nu} x : \langle\!\langle \textit{Node} \rangle\!\rangle)\,\langle\!\langle \textit{Net} \rangle\!\rangle_{\Gamma} \\
\langle\!\langle (\boldsymbol{\nu} x\colon \textit{Ch}(W))\textit{Net} \rangle\!\rangle_{\Gamma} & = & (\boldsymbol{\nu} x : \langle\!\langle \textit{Ch}(W) \rangle\!\rangle)\,\langle\!\langle \textit{Net} \rangle\!\rangle_{\Gamma, x:W} \\
\langle\!\langle \textit{Net}_1 \mid \textit{Net}_2 \rangle\!\rangle_{\Gamma} & = & \langle\!\langle \textit{Net}_1 \rangle\!\rangle_{\Gamma} \mid \langle\!\langle \textit{Net}_2 \rangle\!\rangle_{\Gamma} \\
\langle\!\langle \textit{node}\ n[\textit{Cro}] \rangle\!\rangle_{\Gamma} & = & n[\,\langle\!\langle \textit{Cro} \rangle\!\rangle^{n}_{\Gamma}\,] \\[1ex]
\textit{Crowd}: & & \\
\langle\!\langle (\boldsymbol{\nu} x\colon \textit{Node})\textit{Cro} \rangle\!\rangle^{n}_{\Gamma} & = & (\boldsymbol{\nu} x\colon \langle\!\langle \textit{Node} \rangle\!\rangle)\,\langle\!\langle \textit{Cro} \rangle\!\rangle^{n}_{\Gamma} \\
\langle\!\langle (\boldsymbol{\nu} x\colon \textit{Ch}(W))\textit{Cro} \rangle\!\rangle^{n}_{\Gamma} & = & (\boldsymbol{\nu} x\colon \langle\!\langle \textit{Ch}(W) \rangle\!\rangle)\,\langle\!\langle \textit{Cro} \rangle\!\rangle^{n}_{\Gamma, x:W} \\
\langle\!\langle \textit{Cro}_1 \mid \textit{Cro}_2 \rangle\!\rangle^{n}_{\Gamma} & = & \langle\!\langle \textit{Cro}_1 \rangle\!\rangle^{n}_{\Gamma} \mid \langle\!\langle \textit{Cro}_2 \rangle\!\rangle^{n}_{\Gamma} \\
\langle\!\langle \mathtt{thread}\ [\textit{Th}] \rangle\!\rangle^{n}_{\Gamma} & = & (\boldsymbol{\nu} t\colon \mathsf{Amb}[\mathsf{shh}])\,t[\,\langle\!\langle \textit{Th} \rangle\!\rangle^{n,t}_{\Gamma}\,] \\
\langle\!\langle \mathtt{channel}\ c \rangle\!\rangle^{n}_{\Gamma} & = & c[\,!(x\colon \langle\!\langle \Gamma(c) \rangle\!\rangle)\langle x \rangle\,] \\[1ex]
\textit{Threads}: & & \\
\langle\!\langle \bar{c}(x) \rangle\!\rangle^{n,t}_{\Gamma} & = & (\boldsymbol{\nu} w\colon \mathsf{Amb}^{\circ}[\mathsf{shh}, \langle\!\langle \Gamma(c) \rangle\!\rangle^{\mathsf{w}}])\,w[\,\mathtt{out}\ t.\mathtt{in}\ c.\langle x \rangle^{\uparrow}\,] \\
\langle\!\langle c(x).\textit{Th} \rangle\!\rangle^{n,t}_{\Gamma} & = & (\boldsymbol{\nu} r\colon \mathsf{Amb}^{\circ}[\langle\!\langle \Gamma(c) \rangle\!\rangle, \langle\!\langle \Gamma(c) \rangle\!\rangle^{\mathsf{r}}])\ (x : \langle\!\langle \Gamma(c) \rangle\!\rangle)^{r}\,\langle\!\langle \textit{Th} \rangle\!\rangle^{n,t}_{\Gamma} \mid \\
 & & \quad r[\,\mathtt{out}\ t.\mathtt{in}\ c.(x : \langle\!\langle \Gamma(c) \rangle\!\rangle)^{\uparrow}.\mathtt{out}\ c.\mathtt{in}\ t.\langle x \rangle\,] \\
\langle\!\langle \mathtt{go}\ m.\textit{Th} \rangle\!\rangle^{n,t}_{\Gamma} & = & \mathtt{out}\ n.\mathtt{in}\ m.\,\langle\!\langle \textit{Th} \rangle\!\rangle^{n,t}_{\Gamma} \\
\langle\!\langle \mathtt{fork}\ (c).\textit{Th} \rangle\!\rangle^{n,t}_{\Gamma} & = & (\boldsymbol{\nu} s\colon \mathsf{Amb}[\textit{Synch}])\ ()^{s}\,\langle\!\langle \textit{Th} \rangle\!\rangle^{n,t}_{\Gamma} \\
 & & \qquad \mid\ c[\,\mathtt{out}\ t.(!((x\colon \langle\!\langle \Gamma(c) \rangle\!\rangle)\langle x \rangle) \mid s[\,\mathtt{out}\ c.\mathtt{in}\ t.\langle\rangle\,])\,] \\
\langle\!\langle \mathtt{fork}\ (\textit{Th}').\textit{Th} \rangle\!\rangle^{n,t}_{\Gamma} & = & (\boldsymbol{\nu} s\colon \mathsf{Amb}[\textit{Synch}])\ ()^{s}\,\langle\!\langle \textit{Th} \rangle\!\rangle^{n,t}_{\Gamma} \\
 & & \mid\ (\boldsymbol{\nu} t'\colon \mathsf{Amb}[\mathsf{shh}])\,t'[\,\mathtt{out}\ t.(\,\langle\!\langle \textit{Th}' \rangle\!\rangle^{n,t'}_{\Gamma} \mid s[\,\mathtt{out}\ t'.\mathtt{in}\ t.\langle\rangle\,])\,] \\
\langle\!\langle \mathtt{fork}\ (\textit{Cro}_1 \mid \textit{Cro}_2).\textit{Th} \rangle\!\rangle^{n,t}_{\Gamma} & = & \langle\!\langle \mathtt{fork}(\textit{Cro}_1).\mathtt{fork}(\textit{Cro}_2).\textit{Th} \rangle\!\rangle^{n,t}_{\Gamma} \\
\langle\!\langle \mathtt{fork}\ ((\boldsymbol{\nu} x\colon \textit{Ch}(W))\textit{Cro}).\textit{Th} \rangle\!\rangle^{n,t}_{\Gamma} & = & (\boldsymbol{\nu} x\colon \langle\!\langle \textit{Ch}(W) \rangle\!\rangle)\,\langle\!\langle \mathtt{fork}\ (\textit{Cro}).\textit{Th} \rangle\!\rangle^{n,t}_{\Gamma,,x:W} \\
\langle\!\langle \mathtt{fork}\ ((\boldsymbol{\nu} x\colon \textit{Node})\textit{Cro}).\textit{Th} \rangle\!\rangle^{n,t}_{\Gamma} & = & (\boldsymbol{\nu} x\colon \langle\!\langle \textit{Node} \rangle\!\rangle)\,\langle\!\langle \mathtt{fork}\ (\textit{Cro}).\textit{Th} \rangle\!\rangle^{n,t}_{\Gamma} \\
\langle\!\langle \mathtt{spawn}\ m[\textit{Cro}].\textit{Th} \rangle\!\rangle^{n,t}_{\Gamma} & = & (\boldsymbol{\nu} s\colon \mathsf{Amb}[\textit{Synch}])\ ()^{s}\,\langle\!\langle \textit{Th} \rangle\!\rangle^{n,t}_{\Gamma} \\
 & & \qquad \mid\ m[\,\mathtt{out}\ t.\mathtt{out}\ n.(\,\langle\!\langle \textit{Cro} \rangle\!\rangle^{m}_{\Gamma} \mid s[\,\mathtt{out}\ m.\mathtt{in}\ n.\mathtt{in}\ t.\langle\rangle\,])\,]
\end{array}
$$

Le codage est très simple. Le codage d'un network est paramétrique dans l'environnement de types Γ qui enregistre les types des valeurs transportées par les canaux (on en a besoin car les paramètres des inputs ne sont pas explicitement typés); le codage d'un crowd est paramétrique aussi dans le noeud courant n qui enferme le crowd; le codage des threads est paramétré aussi par le nom du thread courant. Les noeuds sont des ambients silencieux dont les sous-ambients modélisent des threads ou des canaux. La migration d'un thread est obtenue en sortant le noeud courant et en rentrant dans le noeud de destination. Il n'y a que trois points dans le codage qui ne sont pas totalement évidents. Le premier est le codage des forks et des spawns qui nécessitent un ambient de synchronisation s qui garde la continuation du thread. Le second est le codage des forks qui doit être donné par cas sur le crowd: n'ayant pas la capabilité `open` nous ne pouvons pas faire sortir un crowd entier et le relâcher dans le noeud; au lieu de cela nous faisons sortir chaque composante du crowd individuellement. Le troisième est le codage de canaux, lequel utilise des ambients de transport "frais" —appelés w pour writer et r pour reader— qui se déplacent dans le canal, se synchronisent et, dans le cas d'un lecteur, rapatrient le message. Pour typer les readers et les writers nous avons besoin des moded types car il s'agit d'ambients qui ne sont pas silencieux mais qui pour rejoindre leur destination sont amenés à traverser des ambients silencieux. C'est un simple exercice que de vérifier que ces ambients de

transport sont bien typés dans le système de moded types défini dans [BCC01a].

Ce codage parait bien plus simple que le codage défini pour les Ambients Mobiles dans [CGG00] car les canaux et leurs opérations sont codés plus facilement et leurs types sont bien plus proches des types du langage.

Considérons maintenant différentes politiques de sécurité.

Un premier exemple de politique de sécurité est d'enrichir les types des canaux et des threads par des niveaux de sécurité: les threads sont les sujets et les canaux les objets. Nous pouvons ainsi modifier notre codage (où $level(n)$ dénote le niveau de sécurité de l'ambient n):

$$
\begin{array}{lcl}
\langle\!\langle \sigma Ch(W) \rangle\!\rangle & = & \sigma\mathsf{Amb}[\langle\!\langle W \rangle\!\rangle] \\
\langle\!\langle \mathtt{thread}\sigma[Th] \rangle\!\rangle^{n}_{\Gamma} & = & (\boldsymbol{\nu} t\colon \sigma\mathsf{Amb}[\mathsf{shh}])\ldots \\
\langle\!\langle \bar{c}(x) \rangle\!\rangle^{n,t}_{\Gamma} & = & (\boldsymbol{\nu} w\colon level(t)\mathsf{Amb}^{\circ}[\mathsf{shh},\ \langle\!\langle \Gamma(c) \rangle\!\rangle^{\mathsf{w}}])\ldots \\
\langle\!\langle c(x).Th \rangle\!\rangle^{n,t}_{\Gamma} & = & (\boldsymbol{\nu} r\colon level(t)\mathsf{Amb}^{\circ}[\langle\!\langle \Gamma(c) \rangle\!\rangle,\ \langle\!\langle \Gamma(c) \rangle\!\rangle^{\mathsf{r}}])\ldots \\
\langle\!\langle \mathtt{fork}\ (Th').Th \rangle\!\rangle^{n,t}_{\Gamma} & = & (\boldsymbol{\nu} s\colon \mathsf{Amb}[Synch])(\boldsymbol{\nu} t'\colon level(t)\mathsf{Amb}[\mathsf{shh}])\ldots
\end{array}
$$

Il est fort intéressant de noter que, par le biais de la traduction ci-dessus, l'utilisation de la sécurité militaire ou commerciale dans le langage cible induit respectivement la sécurité militaire ou commerciale dans le langage distribué. En fait toute opération non compatible avec la politique de sécurité choisie résulterait en un reader ou en un writer mal typé, et comme tel il serait détecté statiquement.

Une politique différente pourrait être d'enrichir les types noeud par des niveaux de sécurité. Les threads auraient alors le niveau du noeud où ils ont été créés. Ceci est une situation que l'on rencontre fréquemment dans la pratique (d'habitude on la rencontre sous la forme d'une partition des noeuds en "trusted" et "distrusted", donc deux niveaux de sécurité). Elle pourrait être obtenue par une simple modification de la définition précédente:

$$
\begin{array}{lcl}
\langle\!\langle \sigma Node \rangle\!\rangle & = & \sigma\mathsf{Amb}[\mathsf{shh}] \\
\langle\!\langle \mathtt{thread}[Th] \rangle\!\rangle^{n}_{\Gamma} & = & (\boldsymbol{\nu} t\colon level(n)\mathsf{Amb}[\mathsf{shh}])\ldots
\end{array}
$$

Avec cette politique un thread peut se déplacer en tout noeud de n'importe quel niveau, même si une fois à destination il ne pourrait accéder qu'aux ressources permises par son niveau. Si nous voulons interdire aux threads de se déplacer dans des noeuds "incompatibles" (ce qui d'habitude est requis en deux situations: quand nous partitionnons les noeuds en trusted et distrusted et que nous voulons que les threads sensibles ne soient exécutés que sur des noeuds trusted, ou bien quand ce sont les threads qui sont partitionnés en trusted et distrusted et où donc nous voulons qu'un noeud sensible n'exécute que des threads trusted) il suffit de modifier la traduction de façon que les threads notifient leur arrivée au noeud de destination:

$$
\begin{array}{lcl}
\langle\!\langle \sigma Node \rangle\!\rangle & = & \sigma\mathsf{Amb}[Synch] \\
\langle\!\langle node\ n[Cro] \rangle\!\rangle_{\Gamma} & = & n[\langle\!\langle Cro \rangle\!\rangle^{n}_{\Gamma} \mid !()] \\
\langle\!\langle \mathtt{thread}[Th] \rangle\!\rangle^{n}_{\Gamma} & = & (\boldsymbol{\nu} t\colon level(n)\mathsf{Amb}[\mathsf{shh}, Synch^{\mathsf{w}}])\ldots \\
\langle\!\langle \mathtt{fork}\ (Th').Th \rangle\!\rangle^{n,t}_{\Gamma} & = & (\boldsymbol{\nu} s\colon \mathsf{Amb}[Synch])(\boldsymbol{\nu} t'\colon level(t)\mathsf{Amb}[\mathsf{shh}, Synch^{\mathsf{w}}])\ldots \\
\langle\!\langle \mathsf{go}\ m.Th \rangle\!\rangle^{n,t}_{\Gamma} & = & \mathtt{out}\ n.\mathtt{in}\ m.\langle\rangle^{\uparrow}\,\langle\!\langle Th \rangle\!\rangle^{n,t}_{\Gamma}
\end{array}
$$

Toute tentative de déplacer un thread dans un noeud incompatible fera échouer le typage du thread courant. En particulier l'utilisation de la politique commerciale dans les boxed ambients correspondra à un langage distribué qui assure un politique de protection des noeuds car tout noeud ne pourrait exécuter que des threads de niveau supérieur ou égal au sien (le noeud n'exécute que des thread provenant des noeuds en qui il a confiance), tandis que l'utilisation de la politique militaire induit une politique de protection des threads car tout thread ne s'exécutera que sur des noeuds de niveau supérieur ou égal au sien (les threads sensibles ne s'exécutent que sur des noeuds de confiance).

4.10 Conclusion et travaux connexes

Nous avons étudié la sécurité MAC pour les ambients mobiles et argumenté que le calcul n'est pas complètement adapté pour exprimer des concepts de sécurité. Comme solution nous avons présenté les boxed ambients, une variante des ambients mobiles inspirée des primitives de communication du Seal Calcul. Le nouveau calcul complète les constructions de mobilité des ambients mobiles par des mécanismes de protection des ressources que nous considérons plus efficaces et complets. En outre le calcul induit un typage qui, par rapport aux Ambients Mobiles, permet une plus grande flexibilité des communications, une mobilité moins restreinte, et jette une nouvelle lumière sur la relation entre synchronie et asynchronie.

Outre ses liens avec les ambients mobiles et les seals, liens longuement débattus, le calcul partage les mêmes motivations et est superficiellement semblable à Boxed-π de Sewell et Vitek [SV00]. Cependant le développement technique est complètement différent car nous ne fournissons pas de mécanismes pour définir des *wrappers* mais plutôt nous proposons de nouvelles constructions qui permettent de contrôler de façon plus aisée les communications entre ambients. En outre les communications dans notre modèle sont anonymes et effectuées en présence de mobilité, tandis que cette dernière est complètement absente de [SV00].

Dans [HR01] Hennessy et Riley discutent un système de types pour la protection des ressources dans le $D\pi$-calcul, une variation distribuée du π-calcul. Dans $D\pi$ les communications ont lieu sur des canaux nommés qui sont associés à des capabilités de read et write. Le système contrôle que les processus qui accèdent une ressource sont en possessions des capabilités requises. Dans notre approche au contraire un accès est considéré légal ou illégal selon les niveaux de sécurité relatifs du sujet et de l'objet. Une différence supplémentaire réside dans la topologie des locations qui dans le $D\pi$ ont une structure plate car à la différence des ambients elles ne peuvent pas être emboîtées. L'interaction entre la variation dynamique de la structure emboîtée causée par la mobilité et la liaison dynamique de la sémantique de $\uparrow$ rendent le contrôle des accès dans les boxed ambients plus compliqué.

Dans [HR00] les mêmes auteurs discutent le *security-π* une variante du π-calcul où les processus sont syntactiquement déclarés comme exécutés à un certain niveau de sécurité et dont le système de types assure qu'un processus de niveau inférieur ne peut jamais accéder à des ressources de niveau supérieur. Au contraire dans BA les niveaux de sécurité sont spécifiés par les types et représentent le niveau tant des ambients que des ressources associées aux ambients. En plus dans leur article Merro et Hennessy introduisent une définition opportune d'équivalence observationelle qu'ils utilisent pour vérifier la propriété de non-interférence. Il s'agit de propriétés que nous n'avons pas encore étudiées pour les BA.

Enfin notre système de type est comparable à d'autres systèmes de types développés pour les ambients mobiles. Dans [CG99b] les types garantissent l'absence de confusion de types pour les communications. Les systèmes de types de [CGG99] et [Zim00] fournissent un meilleur contrôle de l'utilisation de la mobilité. En plus l'introduction des *groupes* dans [CGG00] fourni des manières très expressives pour empêcher la propagation de noms. Le système de types pour le Safe Ambients, présenté dans [LS00] ajoute un contrôle plus fin sur les interactions entre ambients et permet d'éviter des *interférences graves*. Toutes ces approches sont indépendantes du problème d'accès aux ressources que nous avons étudié ici. Nous croyons que des disciplines de typage similaires, ainsi que l'introduction des groupes puissent être obtenus pour les BA et produire des résultats comparables.

Quatrième partie

Conclusion

Conclusion

Dans ce mémoire j'ai tenté de montrer l'esprit qui sous-tend toute ma recherche: j'ai toujours taché de construire un cadre formel pour des problèmes informatiques, et de réappliquer les résultats obtenus dans ce cadre à la pratique.

Il est d'usage de terminer avec quelques perspectives de recherche. Je pense avoir déjà esquissé les perspectives sur le court terme dans les différents chapitres de cette thèse. Pour ce qui concerne le moyen/long terme mes perspectives se concentrent sur deux axes. D'une part je m'investirai sur le projet de recherche européen MyThS que j'ai proposé en collaboration avec l'Université de Venise et celle de Sussex. Ce projet, qui va démarrer en 2002, porte sur l'utilisation des techniques de typage pour la sécurité dans les paradigmes avec mobilité; donc il s'insère à plein titre dans la continuation des thématiques exposées dans la dernière partie de ce mémoire. D'autre part j'ai l'intention de poursuivre en parallèle des recherches sur le typage et la sécurité des transformations de documents XML, recherches que j'ai démarrées trop récemment pour pouvoir les inclure dans ce rapport.

En marge de ces deux axes principaux j'aimerais pouvoir clore des travaux sur les sujets des deux premières parties de ce mémoire encore inachevés. Je pense en particulier aux multi-méthodes dans les langages à objets à prototypes et à l'utilisation de la surcharge dans les systèmes de modules.

Mais ce que je (me) souhaite le plus est d'avoir la même chance que dans le passé et de continuer à pourvoir collaborer avec des collègues et des amis si brillants et aimables qui rendent le travail léger et agréable.

Bibliographie

[ABL99] R. Amadio, G. Boudol, and C. Lhoussaine. The receptive distributed π-calculus. In *FST&TCS*, number 1738 in Lecture Notes in Computer Science, pages 304–315, 1999.

[AC96a] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

[AC96b] M.-V. Aponte and G. Castagna. Programmation modulaire avec surcharge et liaison tardive. In *Journées Francophones des Langages Applicatifs*, Val Morin, Québec, Canada, 1996. In French.

[AC96c] D. Aspinall and A. Compagnoni. Subtyping dependent types. In *11th Ann. Symp. on Logic in Computer Science*, pages 86–97, 1996.

[Asp95] D. Aspinall. Subtyping with singleton types. In *Proc. Computer Science Logic, CSL'94*, number 933 in Lecture Notes in Computer Science. Springer, 1995.

[Bar84] H.P. Barendregt. *The Lambda Calculus Its Syntax and Semantics*. North-Holland, 1984. Revised edition.

[BC96] J. Boyland and G. Castagna. Type-safe compilation of covariant specialization: a practical case. In Pierre Cointe, editor, *ECOOP '96, 10th European Conference on Object-Oriented Programming*, number 1098 in Lecture Notes in Computer Science, pages 3–25. Springer, 1996.

[BC97] J. Boyland and G. Castagna. Parasitic methods: Implementation of multi-methods for Java. *OOPSLA '97, 12th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications. SIGPLAN Notices*, 32(10):66–76, 1997.

[BC00] M. Bugliesi and G. Castagna. Mobile Objects. In *7th Workshop on Foundation of Object-Oriented Languages*, Boston, 2000. Electronic Proceedings.

[BC01] M. Bugliesi and G. Castagna. Secure safe ambients. In *Proc. of the 28th ACM Symposium on Principles of Programming Languages*, pages 222–235, London, 2001. ACM Press.

[BC02] M. Bugliesi and G. Castagna. Behavioral typing for Safe Ambients. *Computer Languages*, 2002. To appear.

[BCC+96] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

[BCC00] M. Bugliesi, G. Castagna, and S. Crafa. Typed Mobile Objects. In *Proceedings of CONCUR 2000 (11th. International Conference on Concurrency Theory)*, number 1877 in Lecture Notes in Computer Science, pages 504–520. Springer, 2000.

[BCC01a] M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *TACS 2001 (4th. International Symposium on Theoretical Aspects of Computer Science)*, number 2215 in Lecture Notes in Computer Science, pages 38–63, Sendai, Japan, 2001. Springer.

[BCC01b] M. Bugliesi, G. Castagna, and S. Crafa. Mobile Objects. In preparation, It includes the results presented in [BC00, BCC00, BCC01d], 2001.

[BCC01c] M. Bugliesi, G. Castagna, and S. Crafa. Reasoning about security in Mobile Ambients. In *CONCUR 2001 (12th. International Conference on Concurrency Theory)*, number 2154 in Lecture Notes in Computer Science, pages 102–120, Aahrus, Danemark, 2001. Springer.

[BCC01d] M. Bugliesi, G. Castagna, and S. Crafa. Subtyping and matching for Mobile Objects. In *ICTCS 2001 (7th. Italian Conference on Theoretical Computer Science)*, number 2202 in Lecture Notes in Computer Science, pages 235–255, Torino, Italy, 2001. Springer.

[BCDC83] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *J.Symbolic Logic*, 48:931–940, 1983.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Implementing an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann, 1992.

[BL95] V. Bono and L. Liquori. A Subtyping for the Fisher-Honsell-Mitchell Lambda Calculus of Objects. In L. Pacholsky and J. Tiuryn, editors, *Proceedings of International Conference of Computer Science Logic 1994*, volume 933 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 1995.

[Bou92] G. Boudol. Asynchrony and the π-calculus. Research Report 1702, INRIA, http://www-sop.inria.fr/mimosa/personnel/Gerard.Boudol.html, 1992.

[Bru94] B. Bruce, K. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 1(4):127–206, 1994.

[BV02] C. Bryce and J. Vitek. The JavaSeal mobile agent kernel. *Autonomous Agents and Multi-Agent Systems*, 2002. To appear.

[CAB+86] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mender, P. Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics in the NuPRL Proof Development System*. Prentice-Hall, 1986.

[Car88] L. 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, 1984.

[Car99a] L. Cardelli. *Abstractions for Mobile Computation*, volume 1603 of *Lecture Notes in Computer Science*, pages 51–94. Springer, 1999.

[Car99b] L. Cardelli. Abstractions for mobile computations. In *Secure Internet Programming*, number 1603 in Lecture Notes in Computer Science, pages 51–94. Springer, 1999.

[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. Extended abstract. The full version in *Formal Aspects of Computing*, Springer International.

[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. FST&TCS'93.

[Cas94] G. Castagna. *Surcharge, sous-typage et liaison tardive : fondements fonctionnels de la programmation orientée objets*. PhD thesis, Université Paris 7, January 1994. LIENS - Rapport de Recherche.

[Cas95a] G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.

[Cas95b] G. Castagna. A meta-language for typed object-oriented languages. *Theoretical Computer Science*, 151(2):297–352, November 1995.

[Cas96a] G. Castagna. Integration of parametric and "ad hoc" second order polymorphism in a calculus with subtyping. *Formal Aspects of Computing*, 8(3):247–293, 1996.

[Cas96b] Giuseppe Castagna. Le modèle fondé sur la surcharge : une visite guidée. *Technique et Science Informatiques*, 15(6):673–708, june 1996.

[Cas97a] G. Castagna. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science. Birkäuser, Boston, 1997. ISBN 3-7643-3905-5.

[Cas97b] G. Castagna. Unifying overloading and λ-abstractions: $\lambda^{\{\}}$. *Theoretical Computer Science*, 176(1-2):337–345, April 1997. Note.

[CC01] G. Castagna and G. Chen. Dependent types with subtyping and late-bound overloading. *Information and Computation*, 168(1):1–67, 2001.

[CD80] M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre-Dame Journal of Formal Logic*, 21(4):685–693, October 1980.

[CG98] L. Cardelli and A. Gordon. Mobile Ambients. In *Proceedings of POPL '98*. ACM Press, 1998.

[CG99a] L. Cardelli and A. Gordon. Equational properties for Mobile Ambients. In *Proceedings FoSSaCS '99*. Springer LNCS, 1999.

[CG99b] L. Cardelli and A. Gordon. Types for Mobile Ambients. In *Proceedings of POPL '99*, pages 79–92. ACM Press, 1999.

[CG00] L. Cardelli and A. Gordon. A commitment relation for the Ambient Calculus. Available at `http://research.microsoft.com/~adg/Publications/ambient-commitment.pdf`, October 2000.

[CGG99] L. Cardelli, G. Ghelli, and A. Gordon. Mobility types for Mobile Ambients. In *Proceedings of ICALP '99*, number 1644 in Lecture Notes in Computer Science, pages 230–239. Springer, 1999.

[CGG00] L. Cardelli, G. Ghelli, and A. D. Gordon. Ambient groups and mobility types. In *International Conference IFIP TCS*, number 1872 in Lecture Notes in Computer Science, pages 333–347. Springer, 2000.

[CGL92] 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, 1992. Extended and revised version in *Information and Computation* 117(1):115-135, 1995.

[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 Lecture Notes in Computer Science, pages 107–123, Utrecht, The Netherlands, March 1993. Springer.

[CGZ01] G. Castagna, G. Ghelli, and F. Zappa Nardelli. Typing mobility in the Seal Calculus. In *CONCUR 2001 (12th. International Conference on Concurrency Theory)*, number 2154 in Lecture Notes in Computer Science, pages 82–101, Aahrus, Danemark, 2001. Springer.

[Cha92] C. Chambers. Object-oriented multi-methods in Cecil. In *ECOOP'92*, number 615 in Lecture Notes in Computer Science. Springer, 1992.

[CL95] C. Chambers and G. T. Leavens. Type-checking and modules for multi-methods. *ACM Transactions on Programming Languages and Systems*, 17(6):805–843, November 1995.

[CM91] L. Cardelli and J.C. Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1(1):3–48, 1991.

[Cou97a] Judicaël Courant. An applicative module calculus. In *Theory and Practice of Software Development 97*, volume 1214 of *Lecture Notes in Computer Science*, pages 622 – 636, Lille, France, April 1997. Springer-Verlag.

[Cou97b] Judicaël Courant. A module calculus for pure type systems. In *Typed Lambda Calculus and Applications*, number 1210 in Lecture Notes in Computer Science, pages 112–128, 1997.

[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.

[CP96] A. Compagnoni and B.C. Pierce. Multiple inheritance via intersection types. *Mathematical Structures in Computer Science*, 6(5), 1996.

[CRR98] G. Castagna, R. Rousseau, and J.-C. Royer. Fiabilité des langages à objets: le typage est-il suffisant? *l'Objet*, 4(1), 1998.

[CV99] G. Castagna and J. Vitek. Commitment and confinement for the Seal calculus. Rapport de Recherche, Laboratoire d'Informatique, Ecole Normale Supérieure - Paris, 1999.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[CZ02] G. Castagna and F. Zappa Nardelli. The Seal Calculus revisited: contextual equivalences, bisimulation, and types. submitted, 2002.

[DCS00] M. Dezani-Ciancaglini and I. Salvo. Security types for Safe Mobile Ambients. In *Proceedings of ASIAN '00*, pages 215–236. Springer, 2000.

[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 Lecture Notes in Computer Science, pages 151–170, Paris, France, June 1987. Springer.

[DH89] R. Ducournau and M. Habib. La multiplicité de l'héritage dans les langages à objets. *Technique et Science Informatiques*, 8(1):41–62, 1989.

[DHH$^+$95] R. Ducournau, M. Habib, M. Huchard, M.-L. Mugnier, and A. Napoli. Le point sur l'héritage multiple. *Technique et Science Informatiques*, 14(3):309–345, 1995.

[DoD85] US Department of Defense. DoD trusted computer system evaluation criteria, (the orange book). DOD 5200.28-STD, 1985.

[Dyl92] Apple Computer Inc., Eastern Research and Technology. *Dylan: an object-oriented dynamic language*, April 1992.

[FHM94] K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[FLS00] C. Fournet, J-J. Levy, and A Shmitt. An asynchronous, distributed implementation of Mobile Ambients. In *International Conference IFIP TCS*, number 1872 in Lecture Notes in Computer Science. Springer, 2000.

[GH98] A. Gordon and P. D Hankin. A concurrent object calculus: reduction and typing. *In Proceedings HLCL '98, Elsevier ENTC*, 1998. Also Technical Report 457, University of Cambridge Computer Laboratory, February 1999.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Java series. Addison-Wesley, 1996.

[GM94] C.A. Gunter and J.C. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. The MIT Press, 1994.

[HHP93] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[HL94] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st Annual Symposium on Principles of Programming Languages*, pages 123–137, Portland, Oregon, January 1994. ACM Press.

[HR00] M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous π-calculus (extended abstract). In *Automata, Languages and Programming, 27th International Colloquium*, volume 1853 of *Lecture Notes in Computer Science*, pages 415–427. Springer, 2000.

[HR01] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Information and Computation*, 2001. To appear.

[LC96] L. Liquori and G. Castagna. A typed calculus of objects. In J. Jaffar and R. Yap, editors, *Proc. of 1996 Asian Computing Conference*, number 1179 in Lecture Notes in Computer Science, pages 129–141. Springer, 1996.

[Ler94] X. Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the 21st Symposium on Principles of Programming Languages*, pages 109–122. ACM Press, January 1994.

[LS00] F. Levi and D. Sangiorgi. Controlling interference in Ambients. In *POPL '00*, pages 352–364. ACM Press, 2000.

[LY97] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Java series. Addison-Wesley, 1997.

[Mey91] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1991.

[MH02] M. Merro and M. Hennessy. Bisimulation conguences in Safe Ambients. In *POPL '02*. ACM Press, 2002.

[MQ85] D.B. Mac Queen. Modules for standard ML. *Polymorphism*, 2 (2), 1985.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[Pfe93] F. Pfenning. Refinement types for logical frameworks. In *Informal Proceedings of the 1993 Workshop on Types for Proofs and Programs*, May 1993.

[Rém89] D. Rémy. Type-checking records and variants in a natural extension of ML. In *16th Ann. ACM Symp. on Principles of Programming Languages*, 1989.

[San01] D. Sangiorgi. Extensionality and intensionality of the ambient logic. In *Proc. of the 28th ACM Symposium on Principles of Programming Languages*, London, 2001. ACM Press.

[Str67] C. Strachey. Fundamental concepts in programming languages. Lecture notes for International Summer School in Computer Programming, Copenhagen, August 1967.

[SV00] P. Sewell and J. Vitek. Secure composition of untrusted code: Wrappers and causality types. In *13th IEEE Computer Security Foundations Workshop*, 2000.

[Tsu92] H. Tsuiki. *A record calculus with a merge operator*. PhD thesis, Faculty of Environmental Information, Keio University, November 1992.

[Tsu95] H. Tsuiki. A denotational model of overloading — solving a domain equation in a functor category —. Technical Report KSU/ICS-95-01, Institute of Computer Science, Kyoto Sangyo University, 1995.

[VC99a] J. Vitek and G. Castagna. Mobile computations and hostile hosts. In *Journ ées Francophones des Langages Applicatifs*, 1999.

[VC99b] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, number 1686 in Lecture Notes in Computer Science, pages 47–77. Springer, 1999.

[Wan87] M. Wand. Complete type inference for simple objects. In *2nd Ann. Symp. on Logic in Computer Science*, 1987.

[Zap00] F. Zappa Nardelli. Types for Seal Calculus. Master's thesis, Università degli Studi di Pisa, October 2000. Available at `ftp://ftp.di.ens.fr/pub/users/zappa/readings/mt.ps.gz`.

[Zim00] P. Zimmer. Subtyping and typing algorithms for mobile ambients. In *Proceedins of FoSSaCS '99*, volume 1784 of *LNCS*, pages 375–390. Springer, 2000.