Évaluation Partielle Dirigée par les types en Objective Caml

Vincent Balat

Rapport de stage

de deuxième année du magistère d'informatique de l'ENS Lyon,
effectué à BRICS, université d'Aarhus, Danemark
sous la direction d'Olivier Danvy

juillet-août 1997

Remerciements

Je tiens à apporter mes remerciements les plus chaleureux à Olivier Danvy pour son accueil et ses conseils.

Ma reconnaissance va également aux membres de l'équipe et invités qui m'ont aidé et ont contribué à rendre mon séjour au Danemark agréable, notamment à Zhe Yang, Paola Quaglia, Ulrich Kohlenbach, Daniel Damian.

Table des matières

Présentation du stage

Ce stage a été effectué à BRICS[1] , université d'Aarhus[2] au Danemark, sous la direction d'Olivier Danvy.

Nous avons étudié la possibilité d'associer l'évaluateur partiel dirigé par les types d'Olivier Danvy au principe de génération de code au moment de l'exécution pour le langage Objective Caml. Le résultat permet de trouver la forme normale longue d'une valeur close de Caml chargée en mémoire (donc en forme normale de tête faible), de générer le bytecode correspondant et de le charger directement à la place de l'ancien. Le code généré peut être de 2 à plusieurs milliers de fois plus rapide sur les exemples testés. Le principe a été adapté au langage de modules, ce qui permet de faire une optimisation sur les foncteurs.

Parallèlement à ce rapport, un article cosigné avec Olivier Danvy et reprenant ces travaux sera proposé incessamment à une conférence.

1. Basic Research In Computer Science,
   Centre of the Danish National Research Foundation.
2. Department of Computer Science
   University of Aarhus
   Building 540, Ny Munkegade, DK-8000 Aarhus C, Denmark

Chapitre 1

Introduction

1.1 L'évaluation partielle

L'évaluation partielle est une transformation de programmes visant à les spécialiser. Étant donné un programme et une partie de ses données, elle spécialise ce programme par rapport aux données fournies. Si l'évaluateur partiel, le programme source et le programme spécialisé terminent, l'application du programme résiduel au reste des données produira le même résultat que le programme initial appliqué à l'ensemble des données. Il s'agit donc d'une version effective du théorème S_n^m de Kleene.

Le travail d'un évaluateur partiel consiste donc à effectuer toutes les réductions possibles, pour construire la version résiduelle du programme. Il doit donc être capable de repérer les parties du programme, dites *statiques*, qui pourront être réduites, et de construire un nouveau terme, *complètement dynamique*, c'est-à-dire sans parties statiques.

1.2 Type-Directed Partial Evaluation (tdpe)

La méthode d'évaluation partielle dirigée par les types d'Olivier Danvy (*Type Directed Partial Evaluation*, [Danvy 96]) permet d'obtenir la forme normale complètement η-expansée, d'un λ-terme clos typé fortement normalisable, uniquement à partir des informations contenues dans son type.

La transformation, appelée *résidualisation*, s'effectue en deux étapes. Le type permet, dans une premier temps, de construire un terme β-η-équivalent dont les parties statiques sont marquées. Ensuite, toutes les parties marquées sont réduites.

L'algorithme est très simple, et contrairement aux méthodes de réduction « source à source », elle peut être réalisée effectivement dans un langage fonctionnel tel que OCaml ou Scheme à partir de valeurs compilées du langage.

Les parties statiques sont évaluées en tant qu'expressions du langage, sans phase supplémentaire d'interprétation ; les parties dynamiques construites serviront à former le programme résiduel. Elles peuvent à leur tour être chargées en mémoire et exécutées, avec un gain de vitesse dû à la normalisation.

1.3 Run-Time Code Generation (rtcg)

La *Run-Time Code Generation* est une technique qui consiste à générer du code compilé (bytecode) pendant l'exécution d'un programme, sans passer par l'intermédiaire d'un code source. Aucun appel au compilateur n'est réalisé, ce qui permet une très grande vitesse d'exécution.

Associée à un évaluateur partiel, cette technique permet de générer directement du code compilé qui peut être chargé en mémoire afin d'être prêt à être exécuté.

1.4 Une implantation en Objective Caml

Nous proposons ici un exemple d'implantation de Tdpe en Objective Caml utilisant la Rtcg. Le but n'est pas de construire un outil permettant de traiter de manière exhaustive tout le langage Caml, mais plutôt de créer un noyau assez puissant pour pouvoir traiter des exemples intéressants.

Le programme fonctionne à partir du système interactif du langage (toplevel) et permet de réaliser l'optimisation de valeurs définies précédemment, et de les charger directement en mémoire.

Le programme commence par calculer la forme normale, qui est ensuite compilée vers du byte-code, et chargée en mémoire à la place de la valeur initiale. Cette technique a nécessité une légère modification de l'implantation d'Objective Caml, afin d'autoriser l'accès aux fonctions internes de chargement de byte-code.

Le résultat, sur les exemples que nous avons pris, est de 2 à 16000 fois plus rapide que le programme initial.

À notre connaissance, le seul article se rapprochant de ce travail est celui de Sperber et Thiemann [ST 97], dans lequel un évaluateur partiel traditionnel est utilisé avec un générateur de code pour le langage Scheme 48.

1.5 Un exemple : les entiers de Church

Cette section présente un exemple d'utilisation de tdpe pour optimiser des calculs sur les entiers de Church. La figure 1.1 montre un module définissant le zéro ('cz'), le successeur d'un entier de Church ('cs'), l'addition de deux entiers de Church ('cadd'), ainsi que des fonctions de conversion d'entiers vers entiers de Church ('n2cn'), et vice-versa ('cn2n').

Définissons l'opération « ajouter 1000 » de la manière suivante :

```
# let cs1000 a = cadd (n2cn 1000) a;;
```

```
module ChurchNumbers =
  struct
    let cz s z = z

    let cs n s z = n s (s z)

    let cadd a b = a cs b

    let rec n2cn n = if n = 0 then cz else (cs (n2cn (n-1)))
    let cn2n n = n (fun i → i+1)

  end;;
```

FIG. 1.1 - *Entiers de Church en OCaml*

```
val cs1000 : (('a -> 'a) → 'b -> 'a) → ('a -> 'a) → 'b -> 'a = <fun>
```

L'application de 'cs$n$' à 'cz' nécessite $O(n)$ β-réductions qui auraient pu être réalisées statiquement auparavant.

```
# cn2n (cs1000 cz);;
- : int = 1000
```

La résidualisation de cette valeur s'effectue simplement en tapant la commande suivante :

```
# tdpe_2srnor "cs1000";;
- : unit = ()
```

La valeur 'cs1000' contient maintenant la forme optimisée. On peut percevoir nettement la différence en mesurant le temps d'exécution. L'application de 'cs1000' à 'cz' dure 46.10^{-4} ms au lieu de 22 ms. Le résultat est 4800 fois plus rapide. La résidualisation en elle-même dure entre 0.1 et 18 secondes[1] selon les versions utilisées (voir chapitre 6). Elle devient donc rentable à partir de 5 applications de 'cs1000' pour cet exemple.

1.6 Application aux modules

Le langage des modules de Objective Caml présente des similitudes avec un lambda-calcul très simple : un module peut être considéré comme un tuple de valeurs ; les foncteurs permettent de réaliser des abstractions. Ceci nous pousse à appliquer aux foncteurs le même principe d'optimisation.

Nous avons donc légèrement modifié le programme pour lui permettre de faire une optimisation de ce langage. L'optimisation se fait non seulement pour chaque valeur du module, mais aussi au niveau des applications de foncteurs.

1. 0.1 s sans `shift` et `reset` — voir chapitre 3

1.7 Organisation du rapport

Le chapitre 2 présente la technique d'évaluation partielle utilisée à partir d'un lambda-calcul à deux niveaux. L'implantation en Caml est détaillée dans les chapitres 3 et 4, ce dernier traitant de la Rtcg.

Le chapitre 5 décrit la version du programme adaptée au langage des modules.

Les différentes versions du programme ont été testées sur deux exemples d'interpréteurs pour des langages simples qui sont présentés dans le chapitre 6. On y trouvera également une analyse des résultats obtenus.

Chapitre 2

Type-Directed Partial Evaluation

L'évaluation partielle dirigée par les types [Danvy 96] permet de mettre un terme clos en forme normale complètement η-expansée. Le terme initial doit être typé et fortement normalisable.

2.1 Principe de l'évaluateur partiel

Le but d'un évaluateur partiel est de construire, à partir d'un λ-terme, un terme en forme normale, qui lui est β-η-équivalent. L'idée la plus simple pour réaliser une normalisation est de procéder à une analyse syntaxique du terme afin de repérer les β-redex, puis de réaliser effectivement les substitutions.

Cette approche est purement syntaxique et oblige donc à travailler sur le texte du programme pour produire à nouveau le texte d'un programme. Elle présente deux inconvénients : d'une part l'analyse et le travail sur le programme sont coûteux, d'autre part elle ne permet pas de réduire des valeurs déjà compilées.

L'idée proposée par Olivier Danvy consiste à utiliser les propriétés de normalisation du langage lui-même. Sur un terme possédant un type de base, l'évaluateur de Caml effectue les β-réductions conduisant à la forme normale. Pour un terme typé d'ordre supérieur, il n'effectue pas les réductions tant que les paramètres ne lui sont pas fournis. Tdpe commence donc par réaliser l'η-expansion complète du terme initial, en utilisant un *deuxième niveau* de λ-calcul, qui servira à représenter les parties qui figureront dans le résultat (parties dynamiques). Ceci permet d'appliquer complètement le terme initial, c'est-à-dire d'effectuer la réduction de toutes les parties statiques, en appliquant toutes les abstractions à des termes de type adéquat pour lesquels le type de base est toujours celui servant à représenter les termes dynamiques.

Cette technique oblige donc à ne considérer que des types polymorphes pour les types d'ordre 0, puisque dans le terme η-expansé, tous les types de base sont remplacés par le type des termes dynamiques. On est donc contraint, a priori, à effectuer une clôture par toutes les variables libres non polymorphes. On ne considérera dans cette partie que des termes clos.

2.2 Description de l'algorithme

La présentation de l'algorithme utilisera un λ-calcul à deux niveaux, dans lesquelles les abstractions et applications dynamiques seront soulignées, alors que les parties statiques seront surlignées. Le symbole @ sera utilisé pour l'application. Les types de base seront dénoté par la lettre b, la lettre t représentera un type quelconque.

L'algorithme classique d'η-expansion à un niveau est donné par la figure 2.1.

$$\begin{array}{rcl} |^{b}\ v & = & v \\ |^{t_1 \to t_2}\ v & = & \lambda x.\ |^{t_2}(v\ @\ |^{t_1} x) \text{ où } x \text{ est une variable neuve} \end{array}$$

FIG. 2.1 - *Algorithme d'η-expansion*

L'application de cet algorithme à un terme quelconque fortement normalisable, suivi d'une normalisation conduit à la forme normale longue du terme initial (c'est-à-dire à la forme normale complètement η-expansée). Il s'agit maintenant de placer correctement les termes statiques et dynamiques, afin que toutes les parties statiques puissent être évaluées pour obtenir un terme résiduel complètement dynamique en forme normale.

Définition 1 – *Un terme sera dit* statique *(resp.* dynamique*) si c'est une abstraction statique (resp. dynamique) ou une application statique (resp. dynamique) ou bien une variable*

– *Un terme est dit* complètement statique *(resp.* complètement dynamique*) si toutes ses abstractions et toutes ses applications sont statiques (resp. dynamiques).*

Une variable est donc à la fois statique et dynamique.

Par exemple, $\overline{\lambda} f.\underline{\lambda} x.\ f\ \overline{@}\ x$ est un terme statique ; $\overline{\lambda} f.\overline{\lambda} x.\ f\ \overline{@}\ x$ est complètement statique.

On veut modifier les règles de la figure 2.1 pour réaliser l'η-expansion en λ-calcul à deux niveaux d'un terme complètement statique, de telle sorte que la réduction des parties statiques soit toujours possible et conduise à un terme en forme normale complètement dynamique. Pour cela, on définit deux opérations : la *réification*, notée $\downarrow^t$ et la *réflexion*, notée $\uparrow^t$, comme le montre la figure 2.2. La réification construit une abstraction dynamique ; la réflexion une abstraction

$$
\begin{array}{lcl}
\text{réification :} & & \\
\downarrow^b v & = & v \\
\downarrow^{t_1 \to t_2} v & = & \underline{\lambda} x.\ \downarrow^{t_2} (v\ \overline{@}\ \uparrow^{t_1} x) \text{ où } x \text{ est une variable fraîche} \\
\downarrow^{t_1 \times t_2} e & = & \underline{\text{pair}}(\downarrow^{t_1} \overline{\text{fst}}\ v, \downarrow^{t_2} \overline{\text{snd}}\ v) \\
\text{réflexion :} & & \\
\uparrow^b e & = & e \\
\uparrow^{t_1 \to t_2} e & = & \overline{\lambda} v.\ \uparrow^{t_2} (e\ \underline{@}\ \downarrow^{t_1} v) \\
\uparrow^{t_1 \times t_2} e & = & \overline{\text{pair}}(\uparrow^{t_1} \underline{\text{fst}}\ v, \uparrow^{t_2} \underline{\text{snd}}\ v)
\end{array}
$$

FIG. 2.2 - *Évaluation Partielle Dirigée par les Types*

statique.

La résidualisation s'effectue en deux étapes : d'abord la réification du terme statique, puis la réduction de toutes les parties statiques.

Considérons par exemple le terme $\overline{\lambda} f.\overline{\lambda} x.\ (\overline{\lambda} y.\overline{\lambda} z.z)\ \overline{@}\ (f\ \overline{@}\ x)\ \overline{@}\ f$, de type $(b_1 \to b_2) \to b_1 \to b_1 \to b_2$. Après réification, on obtient :

$$\underline{\lambda} a.\underline{\lambda} b.\underline{\lambda} c.\ ((((\overline{\lambda}\mathbf{f}.\overline{\lambda}\mathbf{x}.\ (\overline{\lambda}\mathbf{y}.\overline{\lambda}\mathbf{z}.\mathbf{z})\ \overline{@}\ (\mathbf{f}\ \overline{@}\ \mathbf{x})\ \overline{@}\ \mathbf{f})\ \overline{@}\ (\overline{\lambda} d.\ a\ \underline{@}\ d))\ \overline{@}\ b)\ \overline{@}\ c)$$

puis après réduction de parties statiques :

$$\underline{\lambda} a.\underline{\lambda} b.\underline{\lambda} c.\ (a\ \underline{@}\ c)$$

La génération des variables neuves introduites par réification doit en fait se faire au moment de la réduction statique pour éviter des phénomènes de capture. Considérons l'exemple :

$$\overline{\lambda} f.\overline{\lambda} g.\ g\ \overline{@}\ (\overline{\lambda} a.\ g\ \overline{@}\ (\overline{\lambda} b.\ f\ \overline{@}\ a\ \overline{@}\ b))$$

de type $(b_1 \to b_1 \to b_2) \to ((b_1 \to b_2) \to b_2) \to b_2$. Sa résidualisation doit donner

$$\overline{\lambda} f.\overline{\lambda} g.\ g\ \overline{@}\ (\overline{\lambda} z_1.\ g\ \overline{@}\ (\overline{\lambda} z_2.\ f\ \overline{@}\ z_1\ \overline{@}\ z_2))$$

et non

$$\overline{\lambda} f.\overline{\lambda} g.\ g\ \overline{@}\ (\overline{\lambda} z.\ g\ \overline{@}\ (\overline{\lambda} z.\ f\ \overline{@}\ z\ \overline{@}\ z))$$

alors que z_1 et z_2 sont introduits par le même appel à la fonction de réification.

2.3 Explication de l'algorithme

La correction de l'algorithme a été montrée dans [Berger 93] et [BS 91].

Pour comprendre l'algorithme, on peut faire appel aux notions de types *covariants* et *contravariants*. Pour les définir, on associe une polarité aux types de la manière suivante :

$$\begin{array}{rcl} t & ::= & t^+ \\ t^+ & ::= & b^+ \mid t_1^+ \times t_2^+ \mid t_1^- \rightarrow t_2^+ \\ t^- & ::= & b^- \mid t_1^- \times t_2^- \mid t_1^+ \rightarrow t_2^- \end{array}$$

Une occurrence positive d'un type est dite *covariante*; une occurrence négative est dite *contravariante*.

L'algorithme est écrit de telle sorte que la fonction de réification est appelée sur les types covariants, alors que la réflexion a lieu pour les types contravariants. Pour un terme en forme normale, l'inférence d'un type $t_1 \rightarrow t_2$ covariant correspond au typage d'une abstraction, alors que le type inféré pour la partie gauche d'une application sera contravariant (voir figure 2.3). Cela explique que le λ introduit par la réification est dynamique — il apparaît dans la forme normale — alors que le λ introduit par la réflexion, devant disparaître à la réduction, est statique. De même, les applications statiques sont utilisées lorsque leur premier terme a un type covariant car elles pourront être réduites, sinon on utilise des applications dynamiques. Ainsi toutes les réductions statiques peuvent avoir lieu, et tous les termes restant font partie de la forme normale.

variables :

$$\frac{}{x : \tau, H \vdash x : \tau}$$

abstractions :

$$\frac{x : \sigma, H \vdash t : \tau}{H \vdash \lambda x.t : \sigma \rightarrow \tau}$$

applications :

$$\frac{H \vdash t : \sigma \rightarrow \tau \qquad H \vdash u : \sigma}{H \vdash (t \quad u) : \tau}$$

FIG. 2.3 - *Algorithme de typage*

Chapitre 3

Tdpe en Caml

L'évaluation partielle dirigée par les types va être utilisée pour optimiser des programmes Caml. Une telle implantation a déjà été réalisée en Scheme par Olivier Danvy. Cependant les caractéristiques de Caml interdisent d'utiliser exactement les mêmes méthodes de programmation. La réalisation en Caml est décrite dans la section 3.1. La section 3.2 précise les limitations imposées sur les types. Enfin, la section 3.3 montre comment il est possible d'éviter les duplications de code, et de traiter les fonctions avec effets de bords.

3.1 Programmation en Caml de l'algorithme de base

L'idée consiste à utiliser pour le niveau dynamique un type de données avec constructeurs, et pour le niveau statique le langage Caml lui-même. Ceci permet de faire effectuer automatiquement les réductions statiques au langage.

En Scheme, on peut programmer cela de manière très simple, comme le montre la figure 3.1 (voir [Danvy 96]). (`gensym!` est une fonction générant des variables neuves).

Ce programme ne peut pas être transposé en Caml car il n'est pas correctement typé, par exemple v peut être inféré de type « λ-expression dynamique » et ne peut donc pas être appliqué dans la deuxième ligne de `reify`.

Deux solutions différentes ont été utilisées pour résoudre ce problème : la première est une technique introduite par Andrzej Filinski (pas encore publiée) qui permet de construire facilement la fonction d'η-expansion associée à un type ; la seconde consiste à créer dans un premier temps un terme à deux niveaux en utilisant un type de données adéquat, et ne réaliser la réduction (c'est-à-dire la compilation de l'expression Caml) qu'ensuite.

Cette dernière méthode présente a priori l'inconvénient de ne pas être réalisable en une seule opération : on doit d'abord construire l'expression à deux

```
(define residualize
  (lambda (v t)
    (letrec ([reify
              (lambda (t v)
                (case-record t
                  [(Base)
                   v]
                  [(Func t1 t2)
                   (let ([x1 (gensym!)])
                     `(lambda (,x1)
                        ,(reify t2 (v (reflect t1 x1)))))]))]
             [reflect
              (lambda (t e)
                (case-record t
                  [(Base)
                   e]
                  [(Func t1 t2)
                   (lambda (v1)
                     (reflect t2 `(,e ,(reify t1 v1))))]))])
      (begin
        (reset-gensym!)
        (reify (tdpe_parse-type t) v)))))
```

FIG. 3.1 - Tdpe *en* Scheme

niveaux et l'afficher sous forme d'un programme Caml, ensuite exécuter ce programme pour produire le résultat. Les techniques développées pour la « Run-Time Code Generation » ont permis d'éviter ce problème et de faire la résidualisation en une seule étape.

La figure 3.2 montre un exemple de résidualisation. Le résultat est affiché sous la forme d'un programme Caml grâce à la directive `#install_printer` de OCaml, ce qui permet de se servir très facilement du résultat. Le chargement direct en mémoire fait l'objet du chapitre 4.

```
# let foo = (fun f g → g ((fun h a → h (fun b → f a b)) g));;
val foo : ('a -> 'a → 'b) -> (('a → 'b) -> 'b) → 'b = <fun>
# tdpe_2_rnor_nf "foo";;
- : NormalForms.computation =
(fun v6 v7 → (v7 (fun v8 → (v7 (fun v9 → (v6 v8 v9))))))
```

FIG. 3.2 - *Résidualisation d'un exemple*

3.2 Prise en compte des types non polymorphes

On a vu que la résidualisation d'une fonction grâce à Tdpe consistait à appliquer la fonction à des paramètres formels dont le type est imposé. Cela oblige donc à considérer uniquement des termes pour lesquels tous les types d'ordre 0 sont polymorphes.

Par exemple, dans l'exemple suivant, l'opérateur `-` impose un type `int` pour le 'x'; il n'est pas possible d'appliquer `-` à une λ-expression.

```
# let opp x = - x;;
val opp : int → int = <fun>
# tdpe_2srnor "opp";;
Uncaught exception: Failure("reflect: not possible on type int")
```

On peut résoudre le problème en effectuant une clôture par `-` :

```
# let opp moins x = moins x;;
val opp : ('a -> 'b) → 'a -> 'b = <fun>
# tdpe_2srnor "opp";;
- : unit = ()
```

La clôture par des fonctions polymorphes sans effets de bords n'est pas nécessaire. Une fonction effectuant des effets de bord doit être abstraite, sinon les effets de bord ont lieu au moment de la résidualisation.

En fait, on autorise des types de base simple (`int`, `float`, `bool`, `unit`) lorsqu'ils apparaissent du côté *covariant*, c'est-à-dire lorsqu'ils sont traités par la fonction 'reify'. Il suffit d'ajouter dans le type des formes normales la possibilité de mettre des valeurs de ces types, et de faire une simple traduction dans la fonction 'reify'. Ceci permet l'utilisation de constantes de ces types dans les programmes. Par exemple, les types `int`, `'a → int` et `(float→ 'a) → int` sont autorisés. Ce qui permet de faire :

```
# let a = 3;;
val a : int = 3
# tdpe_2srnor "a";;
- : unit = ()
# let succ (+) x = x + 1;;
val succ : ('a -> int -> 'b) → 'a -> 'b = <fun>
# tdpe_2srnor "succ";;
- : unit = ()
# succ (+) 21;;
- : int = 22
```

Enfin, il convient d'être attentif aux types de la forme `'_a`[1]. Si une fonction dont le type contient un sous-type de cette forme est appliquée avant la résidualisation, le type ne sera plus suffisamment polymorphe. Les versions de Tdpe

1. Cette notation est utilisée par Caml pour les types qui ne sont pas encore déterminés. Ils seront déterminés définitivement par la prochaine utilisation de la valeur

utilisant des termes à deux niveaux présentent l'avantage de ne pas spécialiser ces types. Ce problème provient très souvent de variable polymorphiques faibles. On les évite en faisant une simple η-expansion, comme dans l'exemple suivant, avec les entiers de Church (voir chapitre 1) :

```
# let cs1000 = cadd (n2cn 1000);;
val cs1000 : (('_a -> '_a) → '_b -> '_a) → ('_a -> '_a) → '_b -> '_a =
  <fun>
# let cs1000 a = cadd (n2cn 1000) a;;
val cs1000 : (('a -> 'a) → 'b -> 'a) → ('a -> 'a) → 'b -> 'a = <fun>
```

3.3 Améliorations de l'algorithme

Lorsqu'un programme contient des fonctions statiques dont les paramètres sont répétés non linéairement, par exemple à l'aide de constructions 'let ... in ...', la résidualisation peut entraîner des duplications de codes, ainsi que des modifications de l'ordre d'exécution des effets de bord. Considérons par exemple le fonction suivante :

```
# let dupl f g x = let y = f x in g y y;;
val dupl : ('a -> 'b) → ('b -> 'b → 'c) -> 'a → 'c = <fun>
```

Avec une version de base de Tdpe, on obtient :

```
# tdpe_2_rnor_nf "dupl";;
- : NormalForms.computation = (fun v6 v7 v8 → (v7 (v6 v8) (v6 v8)))
```

On peut résoudre ce problème en programmant avec des continuations (transformation CPS). Dans cet exemple, cela imposerait de définir la fonction de la manière suivante :

```
# let dupl2 f g x k = f x (fun v → g v v k);;
```

L'utilisation des opérateurs de contrôle `shift` et `reset` décrits dans [DF 90] et [DF 92] permet de résoudre le problème en style direct par introduction de 'let' :

```
# let dupl f g x = let y = f x in g y y;;
val dupl : ('a -> 'b) → ('b -> 'b → 'c) -> 'a → 'c = <fun>
# tdpe_2srnor_nf "dupl";;
- : NormalForms.computation =
(fun v9 v10 v11 →
 let v12 = v9 v11 in
 let v13 = v10 v12 in
 let v14 = v13 v12 in
 v14)
```

`reset` délimite un contexte, que `shift` peut ensuite abstraire. Leur fonctionnement ne sera pas décrit ici. Mentionnons juste qu'ils permettent l'introduction de 'let'.
Par exemple, 1 + reset (**fun** () → 10 + (shift (**fun** k → (k 2) + (k 3))))
est équivalent à 1 + **let** k = **fun** v → 10 + v **in** (k 2) + (k 3)

`shift` (en Caml), réalise en fait une copie de la pile. Il a été implanté par Olivier Danvy grâce à un callcc pour OCaml, écrit par Xavier Leroy.

L'utilisation de `shift` et `reset` diminue considérablement les performances de Tdpe en Caml, mais permet de ne pas imposer de contraintes au programmeur (écriture avec continuations). Des comparaisons des deux méthodes figurent dans le chapitre 6.

Chapitre 4

La Run-Time Code Generation en OCaml

Associer les concepts d'évaluation partielle et de *Run-Time Code Generation* permet d'avoir un outil beaucoup plus puissant :

- Tout le travail s'effectue en une seule passe ;
- La compilation est très simplifiée.

4.1 Chargement en mémoire au moment de l'exécution

Le résultat de l'évaluation partielle est chargé directement en mémoire au moment de l'exécution. Tout le travail se fait donc en une seule passe ; l'utilisateur n'a pas à compiler lui même le résultat pour le mettre en mémoire. Dans notre version de Tdpe, la version optimisée remplace directement le programme initial.

En appliquant deux fois cette technique, on évite aussi la passe supplémentaire nécessaire pour la version de Tdpe qui construit des λ-termes à deux niveaux (voir page 13). Elle devient donc aussi facilement utilisable que la version basée sur la méthode de Filinski.

Certains langages disposent d'une fonction d'évaluation capable d'exécuter une phrase du langage passée en paramètre sous forme de chaîne de caractères. Une telle fonction — qui n'existe pas en Ocaml — aurait permis de réaliser ceci facilement. On aurait pu également faire un appel au compilateur sur le texte du résultat fourni par l'évaluateur partiel.

Comparée à ces deux idées, la technique utilisée est beaucoup plus efficace, car elle permet de supprimer des étapes de compilation.

4.2 Génération de bytecode

Le bytecode n'est pas généré par un appel au compilateur sur le texte d'un programme. Le résultat de l'évaluation partielle utilise dans un premier temps une structure de données Caml très simple (*NormalForms.computation*) adaptée à la représentation des λ-termes en forme normale. Un générateur de bytecode a été écrit spécialement pour cette structure, ce qui évite d'appeler celui utilisé par Caml qui est beaucoup plus compliqué.

On évite également les phases successives d'analyse du texte, de vérification des types, etc. par lesquelles on serait passé si l'on avait produit le texte d'un programme.

L'espression à deux niveaux construite utilise le type *Lambda.lambda* dont Caml se sert pour représenter les expressions du langage pour le niveau statique. Une fois compilée, cette expression a le type *NormalForms.computation*.

La méthode utilisée impose donc d'avoir accès aux fonctions de chargement de bytecode d'OCaml, ce qui a nécessité de légères modifications de l'implantation de OCaml :

- Des points d'accès aux fonctions utiles ont été ajouté dans le module Topdirs (qui est un des modules chargés par défaut au toplevel) ;
- Certaines interfaces (`.cmi`) définissant des types utilisés par le compilateur doivent être rendue accessibles en étant copiées dans le répertoire contenant les bibliothèques d'OCaml.

Les principales fonctions dont on a besoin sont :

- des fonctions utilisées pour les différentes étapes de la compilation ;
- des fonctions nécessaires au chargement du code en mémoire
- une fonction retournant l'environnement courant, pour pouvoir avoir les informations nécessaires sur les valeurs chargées en mémoire ;
- des fonctions permettant de retrouver des objets dans cet environnement.

De plus, on peut accéder directement au type inféré par Caml pour la fonction à optimiser, ce qui évite d'avoir à le passer en paramètre à Tdpe. L'utilisation de Tdpe est donc rendue très simple. L'exemple suivant montre la définition d'une fonction simple, et son exécution avant et après l'appel à Tdpe. Des mesures d'efficacité sont réalisées dans le chapitre 6.

```
# let app2 g a = (fun f x y → f x y) g a a;;
val app2 : ('a -> 'a → 'b) -> 'a → 'b = <fun>
# app2 (+) 2;;
- : int = 4
# tdpe_2srnor "app2";;
- : unit = ()
# app2 (+) 2;;     (* version résidualisée *)
- : int = 4
```

Chapitre 5

Extension au langage de modules

Objective Caml ajoute à Caml un langage spécifique pour la gestion des modules. Optimiser des modules grâce à Tdpe ne signifie pas seulement optimiser chaque valeur contenue dans un module ; on peut profiter de la similitude entre ce langage et le λ-calcul pour effectuer une optimisation sur les applications de foncteurs.

Cette extension est rendue facile par le fait que les modules sont stockés en mémoire comme des tuples de valeurs, et les foncteurs comme des fonctions sur des tuples. Il suffit donc de construire le type correspondant à un module ou un foncteur à partir de sa signature, et d'appeler Tdpe avec ce type.

La figure 5.1 présente un exemple d'utilisation du langage des modules, écrit par Olivier Danvy, pour lequel une optimisation par Tdpe est intéressante. Il s'agit d'un module qui applique un foncteur plusieurs fois ; on utilise Tdpe avec le foncteur partiellement appliqué.

```
# module Base = struct let f a = a let g a = a end;;
  module Base : sig val f : 'a -> 'a val g : 'a -> 'a end
# module App5
           = functor (F : functor(B : sig val f : 'a -> 'a val g : 'a
-> 'a end)
                          → sig val f : 'a -> 'a val g : 'a -> 'a end)
             → functor(B : sig val f : 'a -> 'a val g : 'a -> 'a end)
                → F (F (F (F (F (B)))));;
  module App5 :
    functor
      (F : functor(B : sig val f : 'a -> 'a val g : 'a -> 'a end) →
             sig val f : 'a -> 'a val g : 'a -> 'a end) →
      functor(B : sig val f : 'a -> 'a val g : 'a -> 'a end) →
        sig val f : 'a -> 'a val g : 'a -> 'a end
# module DoStg
           = functor (B : sig val f : 'a -> 'a val g : 'a -> 'a end)
             → struct let f = B.f
                      let g x = f (B.g x)
                end;;
  module DoStg :
    functor(B : sig val f : 'a -> 'a val g : 'a -> 'a end) →
      sig val f : 'a -> 'a val g : 'a -> 'a end
# module Part5 = App5 (DoStg);;
  module Part5 :
    functor(B : sig val f : 'a -> 'a val g : 'a -> 'a end) →
      sig val f : 'a -> 'a val g : 'a -> 'a end
  # module Res5'' = Part5 (Base);;
  module Res5'' : sig val f : 'a -> 'a val g : 'a -> 'a end
  # tdpe_mod "Part5";;
  - : unit = ()
  # module Res5''' = Part5 (Base);;
  module Res5''' : sig val f : 'a → 'a val g : 'a → 'a end
```

FIG. 5.1 - *Exemple d'optimisation de modules*

Chapitre 6

Résultats

Tdpe a été testé sur deux exemples d'interpréteurs pour des langages impératifs non-triviaux : Tiny et Medium, comme cela avait été fait dans [DV 96] pour Medium avec une version en Scheme de Tdpe. Tiny a été programmé entièrement en CPS, et permet donc d'utiliser les versions de Tdpe sans `shift` et `reset`. Il sera comparé avec Medium, programmé pour OCaml en style direct et modulaire par Olivier Danvy.

6.1 Les langages

La syntaxe de Tiny est résumée ci-dessous :

$$
\begin{array}{rcl}
\langle pgm\rangle & ::= & \langle cmd\rangle \\
\langle decl\rangle & ::= & \mathbf{var}\ \langle ide\rangle\mathbf{:}\ \langle btype\rangle\mathbf{=}\ \langle exp\rangle \\
\langle cmd\rangle & ::= & \langle one\_cmd\rangle\mathbf{;}\ \langle cmd\rangle | \langle one\_cmd\rangle \\
\langle one\_cmd\rangle & ::= & \mathbf{skip} \\
 & | & \mathbf{write}\ \langle exp\rangle \\
 & | & \mathbf{read}\ \langle ide\rangle \\
 & | & \langle ide\rangle\mathbf{:=}\ \langle exp\rangle \\
 & | & \mathbf{if}\ \langle exp\rangle\mathbf{then}\ \langle cmd\rangle\mathbf{else}\ \langle one\_cmd\rangle \\
 & | & \mathbf{while}\ \langle exp\rangle\mathbf{do}\ \langle one\_cmd\rangle \\
 & | & \mathbf{block}\ \mathbf{(}\ \langle list\_decl\rangle\mathbf{)}\ \mathbf{in}\ \langle cmd\rangle\mathbf{end} \\
 & | & \mathbf{block}\ \langle cmd\rangle\mathbf{end} \\
\langle list\_decl\rangle & ::= & \langle decl\rangle\mathbf{,}\ \langle list\_decl\rangle \\
 & | & \langle decl\rangle \\
\langle exp\rangle & ::= & \langle exp\rangle\langle op\rangle\langle exp\rangle \\
 & | & \mathbf{(}\ \langle exp\rangle\mathbf{)} \\
 & | & \langle lit\rangle
\end{array}
$$

$$
\begin{array}{rcl}
 & \mid & \langle ide \rangle \\
\langle lit \rangle & ::= & \langle entier \rangle | \langle flottant \rangle | \ \textbf{true} \mid \textbf{false} \\
\langle op \rangle & ::= & + | * | - \mid < \mid = \mid \textbf{and} \mid \textbf{or} \\
\langle btype \rangle & ::= & \textbf{bool} \mid \textbf{int} \mid \textbf{real}
\end{array}
$$

La syntaxe de Medium ressemble beaucoup mais permet en plus la déclaration de procédures et le sous-typage.

6.2 Spécialisation de l'interpréteur

Les interpréteurs ont été écrit en abstrayant toutes les fonctions réalisant des effets de bord ou pas suffisamment polymorphes (accès à la mémoire, opérations,...). Pour exécuter le programme, il suffit d'appliquer la fonction d'interprétation au programme et à toutes ces fonctions.

En appliquant l'interpréteur uniquement au programme et en réalisant une évaluation partielle grâce à Tdpe, on construit une version spécialisée de l'interpréteur pour le programme. Toute la couche d'interprétation est supprimée par la résidualisation ; on obtient donc une version compilée du programme.

Tdpe associé à un interpréteur réalise donc le travail d'un compilateur. L'écriture de ce compilateur se résume uniquement à l'écriture de l'interpréteur, basée sur la définition sémantique du langage.

Une partie du code de l'interpréteur est donnée par la figure 6.1. Son application au programme Tiny de la figure 6.2 puis sa résidualisation sous forme d'un programme Caml donne le résultat présenté sur la figure 6.3

6.3 Tests de performances

Le tableau de la figure 6.3 montre les mesures effectuées sur un programme Tiny de 1000 lignes avec trois différentes versions de Tdpe. La première utilise `shift` et `reset`, alors que la deuxième ne l'utilise pas. Cette dernière se révèle 3.6 fois plus rapide à la compilation, et produit un code plus efficace (1.5 fois plus rapide), car les applications ne sont pas nommées.

La troisième colonne montre les résultats avec une version de Tdpe qui n'utilise pas le type de données des formes normales ($NormalForms.Computation$), mais le type des lambda-expressions utilisé par Caml. Le code est produit par le générateur de code d'OCaml. La résidualisation se révèle beaucoup plus lente dans ce cas, et le code généré beaucoup moins efficace.

Avec Medium, l'utilisation de `shift` et `reset` est obligatoire, car il est écrit en style direct. La figure 6.3 montre que ce type de programmation est cependant beaucoup mieux adapté à Tdpe, probablement parce que Caml n'est pas optimisé pour la programmation en CPS. Le programme Medium testé comportait 10000 lignes. La résidualisation est rapide, et le résultat efficace. On voit donc que sur cet exemple non trivial, écrit modulairement, Tdpe donne des résultats très intéressants.

```
let interpret p k add sub mul equal lt
    et ou read_int read_real write fix is_that_true
    quote_int quote_real coerce_real
    lookup_int lookup_real lookup_bool
    update_int update_real update_bool init_store =

  let rec inter_pgm p k s = match p with
    Pgm c →
      inter_cmd
        c
        (fun i → raise (Error ("undeclared identifier "^i)))
        (0,0,0)
        k
        s

  and inter_decl d offset r k s = match d with ...

  and inter_cmd c r offset k s = match c with
    Skip → k s
  | Write e →
      inter_exp e r (fun v → write v s k) s
  | Aff (i, e) →
      let off,typ = (r i)
      in inter_exp
        e
        r
        (fun v → (match typ with
           TInt → update_int off v s k
         | TReal → update_real off (coerce_real v) s k
         | TBool → update_bool off v s k))
        s
  | ...

  and inter_exp e r k s = match e with ...

  ...

  in inter_pgm p k init_store;;
```

Fig. 6.1 - *Forme générale de l'interpréteur* Tiny *(en CPS) en* Caml

```
(* Factorielle : *)
block(var res : int = 1,
      var val : int = 2,
      var aux : int = 3)
in
  read val;
  aux := 1;
  while 0 < val do
    block
      aux := aux * val;
      val := val - 1
    end;
  res := aux;
  write res
end
```

FIG. 6.2 - *Factorielle en* Tiny

```
fun k add sub mul equal lt et ou read_int read_real write fix
    is_that_true quote_int quote_real coerce_real lookup_int lookup_real
    lookup_bool update_int update_real update_bool init_store →
 (update_int 2 (quote_int 3) init_store
 (fun v0 → (update_int 1 (quote_int 2) init_store
   (fun v1 → (update_int 0 (quote_int 1) init_store
     (fun v2 → (read_int
       (fun i3 → (update_int 1 i3 init_store
         (fun v4 → (update_int 2 (quote_int 1) v4
           (fun v5 → (fix
             (fun v6 v7 v8 → (lookup_int 1 v7
               (fun v9 → (lt (quote_int 0) v9
                 (fun x10 → (is_that_true x10
                   (fun v11 v12 → (lookup_int 2 v11
                     (fun v13 → (lookup_int 1 v11
                       (fun v14 → (mul v13 v14
                         (fun x15 → (update_int 2 x15 v11
                           (fun v16 → (lookup_int 1 v16
                             (fun v17 → (sub v17 (quote_int 1)
                               (fun x18 → (update_int 1 x18 v16
                                 (fun v19 → (v6 v19
                                   (fun v20 → (v12 v20)))))))))))))
))))
                   (fun v21 v22 → (v22 v21)) v7
                   (fun v23 → (v8 v23)))))))) v5
             (fun v24 → (lookup_int 2 v24
               (fun v25 → (update_int 0 v25 v24
                 (fun v26 → (lookup_int 0 v26
                   (fun v27 → (write v27 v26
                     (fun v28 → (k v28))))))))))))))))))))))
```

FIG. 6.3 - *Factorielle après résidualisation. (version de* Tdpe *sans* `shift` *et* `reset`*)*

	Tdpe avec `shift` et `reset`	Tdpe sans `shift` et `reset`	Tdpe sans `shift` et `reset` avec type `Lambda.lambda`
temps de résidualisation	72 min 17 s	19 min 50 s	64 min 27 s
mémoire utilisée	28 M	4.9 M	7 M
temps d'exécution après résidualisation	0.070 s	0.044 s	0.067 s

FIG. 6.4 - Tdpe *sur un programme* Tiny *de 1000 lignes. (Temps d'exécution avant résidualisation : 0.097s)*

	Tdpe avec `shift` et `reset`
temps de résidualisation	45 s
mémoire utilisée	25.8 M
temps d'exécution après résidualisation	0.16 s

FIG. 6.5 - Tdpe *sur un programme* Medium *de 10000 lignes. (Temps d'exécution avant résidualisation : 0.42s)*

Chapitre 7

Conclusion

Les résultats décrit dans le chapitre précédent montrent que Tdpe permet de faire une optimisation très intéressante des programmes Caml. Le temps de résidualisation est très variable selon les programmes, mais dans la plupart des cas, son coût est amorti après seulement quelques utilisations. La Rtcg permet de rendre l'utilisation de Tdpe très simple.

La principale limitation de Tdpe est probablement l'obligation de travailler sur des termes clos. Mais l'introduction pour la première fois de la possibilité d'utiliser des modules permet de le faire de manière très naturelle.

En plus de la programmation de Tdpe en Caml avec Rtcg, ce stage m'a permis de comprendre le principe de fonctionnement de la machine virtuelle d'OCaml, et a été l'occasion de réfléchir au concept d'évaluation partielle. J'ai notamment proposé une nouvelle explication de l'algorithme de Tdpe dans le cadre du λ-calcul pur simplement typé (voir chapitre 2). Les parties concernant l'introduction des constantes, l'insertion des 'let' et les modules n'ont pas encore été prouvées.

Le travail effectué pendant le stage sera poursuivi durant l'année, avec notamment la réalisation d'une distribution de Tdpe pour OCaml avec un manuel et la proposition d'un article.

Annexe A

Installation et Utilisation de Tdpe pour OCaml

A.1 Installation

Tdpe fonctionne avec une version de Objective Caml (1.05) légèrement modifiée pour la Rtcg. Elle nécessite donc la recompilation du langage, après remplacement de certains fichiers.

Pour l'explication de la procédure d'installation, on suppose que le fichier `rtcg.tar` a été désarchivé dans un répertoire `~/tdpe`, qui contiendra aussi la nouvelle version d'OCaml. Le répertoire `rtcg` contient quatre sous-répertoires :

- `ocaml_modif` contient les modifications de la distributions d'OCaml ;
- `callcc` contient le module `callcc` de Xavier Leroy, nécessaire pour `shift` et `reset` ;
- `tdpe` contient les fichiers de Tdpe ;
- `examples` contient les exemples mentionnés dans ce rapport.

A.1.1 Recompilation d'OCaml

Pour recompiler OCaml :

- Remplacer les fichiers
  - `toplevel/topdirs.ml`
  - `toplevel/topdirs.mli`
  - `typing/ident.mli`

  de la distribution de OCaml 1.05 par les fichiers correspondants du répertoire `ocaml_modif`.

- Recompiler OCaml en suivant les règles habituelles, c'est-à-dire :
```
./configure -libdir ~/tdpe/lib -bindir ~/tdpe/bin -mandir ~/tdpe/man
make world
```
et éventuellement :
```
make bootstrap
make opt
```
puis : `make install`
(voir le fichier `INSTALL` d'OCaml pour plus d'informations)
- Avant de taper `make clean`, copier les fichiers suivant dans le répertoire `lib` :
```
bytecomp/instruct.cmi
bytecomp/lambda.cmi
typing/ident.cmi
typing/path.cmi
typing/types.cmi
parsing/longident.cmi
parsing/asttypes.cmi
parsing/parsetree.cmi
```
- Taper `make clean`

A.1.2 Installation de Callcc

Dans le répertoire `callcc`, modifier le fichier `Makefile` pour préciser le répertoire de la distribution d'OCaml, puis taper `make`.

Ensuite, créer un « toplevel » avec `callcc`. Pour cela, se placer dans le répertoire `bin` d'OCaml, vérifier que la version par défaut d'`ocamlc` est la nouvelle version (sinon modifier par exemple le fichier `ocamlmktop`), puis taper :
```
./ocamlmktop -custom -o ~/tdpe/rtcg/ocaml_tdpe ~/tdpe/rtcg/callcc/callcc.o
~/tdpe/rtcg/callcc/callcc.cmo
```

A.1.3 Installation de Tiny

À partir du répertoire `rtcg/examples/tiny`, modifier le fichier `Makefile` pour préciser le répertoire du compilateur modifié, puis taper `make`.

A.2 Utilisation

Pour utiliser Tdpe, lancer `ocaml_tdpe` à partir du répertoire `rtcg`. Pour charger Tdpe, taper :
```
#use ``load.ml''
```
qui charge notamment :

- `pretty_nor.ml`, pretty-printer pour le type `NormalForms.Computation`. Il est utilisé par défaut pour afficher ce type. Pour l'enlever, taper
`#remove_printer Pretty_nor.print_computation;;`.

– `tdpe_2_rlam.ml`, tdpe sans `shift-reset`, avec rtcg a partir du type `Lambda.lambda`

> **module** *Tdpe_2_rlam* :
> **sig**
> **val** tdpe_2_rlam_lam : string → *Lambda*.lambda
> **val** tdpe_2_rlam : string → unit
> **end**

exemple : tdpe_2_rlam "paf";; résidualise la fonction `paf` et charge le résultat en mémoire. tdpe_2_rlam_lam ne charge pas en mémoire mais affiche le résultat.

– `tdpe_2_rnor.ml`, tdpe sans `shift-reset`, avec rtcg à partir du type `NormalForms`.

> **module** *Tdpe_2_rnor* :
> **sig**
> **val** tdpe_2_rnor_nf : string → *NormalForms*.computation
> **val** tdpe_2_rnor : string → unit
> **end**

– `tdpe_2srnor.ml`, tdpe avec `shift-reset`, avec rtcg à partir du type `NormalForms`.

> **module** *Tdpe_2srnor* :
> **sig**
> **val** tdpe_2srnor_nf : string → *NormalForms*.computation
> **val** tdpe_2srnor : string → unit
> **end**

– `tdpe_mod.ml`, identique au précédent, avec en plus le gestion des modules. Il travaille indifféremment sur des modules (foncteurs) ou des valeurs.

> **module** *Tdpe_2srnor* :
> **sig**
> **val** tdpe_mod_nf : string → *NormalForms*.computation
> **val** tdpe_mod : string → unit
> **end**

Les 4 modules de tdpe sont ouverts par `load.ml`. `load_mod.ml` ne charge que `tdpe_mod`.

A.3 Tiny

Chargement de Tiny :

```
#cd "examples/tiny";;
```

```
#use "tiny.ml";;
```

Chargement et exécution d'un programme:

```
let prog = load "fact.tiny";;

let p = interpret prog;;

run p 5;; (* 5 est la taille de la memoire *)

tdpe_mod "p";;

run p 5;;
```

Bibliographie

[Danvy 96] Olivier DANVY, *Type-Directed Partial Evaluation*, in Proceedings of POPL 96, the 1996 ACM Symposium on Principles of Programming Languages

[DF 90] Olivier DANVY et Andrzej FILINSKI, *Abstracting Control*, in Mitchell Wand, editor, Proceedings of the 1990 ACM Conference on Lisp and Functionnal Programming, pages 151-160, Nice, France, Juin 1990. ACM Press

[DF 92] Olivier DANVY et Andrzej FILINSKI, *Representing Control, a study of the CPS transformation*, in Mathematical Structures in Computer Science, 2(4):361-391, Décembre 1992

[BS 91] Ulrich BERGER et Helmut SCHWICHTENBERG, *An inverse of the Evaluation Functional for Typed λ-calculus*, in Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science, pages 203-211, Amsterdam, Pays-Bas, Juin 1991. IEEE Computer Society Press

[Berger 93] Ulrich BERGER, *Program Extraction from Normalization Proofs*, in Typed Lambda Calculi and Applications, éditeurs M. Bezem et J.F. Groote, Lectures Notes in Computer Science, numéro 664, pages 91-106, TLCA, Utrecht, Pays-Bas, Mars 1993.

[ST 97] Michael SPERBER et Peter THIEMANN *Two for the price of one; Composing Partial Evaluation and Compilation*, in Proceedings of the 1997 ACM Conference on Programming Language Design and Implementation, pages 215-225, Las Vegas, USA.

[DV 96] Olivier DANVY et René VESTERGAARD, *Semantics-Based Compiling: A Case Study in Type-Directed Partial Evaluation*, technical report, BRICS-RS-96-13, Computer Science Department, Aarhus University, Danemark, à paraître dans PLILP'96

[OCAML] Xavier Leroy, *The Objective Caml system, release 1.05, documentation and user's manual*, Institut National de la Recherche en Informatique et Automatique, 1997