THÈSE DE DOCTORAT

présentée à

L'UNIVERSITÉ PARIS 7

Spécialité : Informatique

par

M. Denis BECHET

Sujet de thèse :

Les valeurs alternatives et la notion d'événement dans l'évaluation partielle

Soutenue le 19 octobre 1995 devant la commission d'examen composée de

MM.	Didier GALMICHE	Président
	Charles CONSEL	Rapporteurs
	Yves LAFONT	
	Guy COUSINEAU	Examinateurs
	Chris HANKIN	

Remerciments

Je remercie Didier Galmiche, Maître de Conférence habilité à diriger des recherches à l'Université de Nancy I, d'avoir accepté d'être président de mon jury et de me m'avoir soutenu et encouragé depuis mon arrivé à Nancy.

Je remercie Charles Consel, Professeur à l'université de Rennes, d'avoir bien voulu rapporter cette thèse et de m'avoir poussé à réaliser le logiciel LaMix.

Je voudrais remercier particuliérement Yves Lafont, Chercheur au CNRS à Marseille, qui a suivi mon travail depuis mon stage de D.E.A. à l'École Normale Supérieure et qui a bien voulu accepté le pénible travail de rapporteur.

Je remercie, de la même manière, Guy Cousineau, Professeur à Paris 7, mon directeur de thèse, pour tout le temps qu'il a consacré à mon travail, particulièrement vers la fin de ma thèse.

Je remercie vigoureusement Chris Hankin, Professeur à l'Impérial College of London. qui a bien voulu venir spécialement en France pour participer à mon jury.

Enfin, je remercie toutes les personnes qui m'ont aidé, soutenu, encouragé tout au long de ces longues années qui ont précédé ma soutenance.

Résumé

L'évaluation partielle est une technique de transformation de programmes basée sur une évaluation d'un programme lorsqu'un ou plusieurs arguments sont fixés et d'autres inconnus. Ainsi la fonction calculant la puissance n-ième d'un nombre peut être évaluée partiellement lorsque n est fixé. Le résultat de cette évaluation partielle est un programme *résiduel* définissant une fonction équivalente à la fonction originale mais *spécialisée* pour ces arguments constants. Ainsi, le résultat de la spécialisation de la fonction puissance lorsque n est fixé à 3 doit se comporter comme une fonction calculant le cube d'un nombre. Le but de ce mécanisme consiste à tenir compte des valeurs fixées pour créer une version plus efficace que la fonction originale appelée avec ces constantes.

Le second principe, spécifique à l'évaluation partielle par rapport à d'autres mécanismes de transformation de programmes, précise que cette opération doit s'apparenter à une évaluation du programme original. Certaines opérations ne pourront pas être calculées puisque certains arguments restent inconnus, tandis que d'autres pourront être *réduites*.

Dans ce travail, nous nous sommes intéressés à l'extension du domaine utilisé par les évaluateurs partiels pour décrire les différentes valeurs que peuvent prendre une expression. Nous parlerons de *valeurs alternatives*.

En effet, les évaluateurs partiels contemporains formalisent l'ensemble des valeurs pouvant être retournées par une expression du programme spécialisé comme étant soit complètement calculable (une valeur constante), soit complètement inconnu, soit partiellement connu (comme une paire dont seule la première composante est calculable). Rien ne permet de décrire des valeurs provenant de plusieurs sources comme la valeur retournée par une structure de contrôle dont la valeur testée n'est pas connue lors de la spécialisation.

Le domaine des valeurs alternatives décrit des ensembles de valeurs par des grammaires hors-contextes dont les symboles sont des *clones* de constructeurs ou de constantes et les règles, des instances de la déclaration du type associé à ces constructeurs. Ceci permet, par exemple, de décrire des ensembles de valeurs pouvant représenter plusieurs constantes, des listes de longueur inconnue mais dont les éléments sont des constantes...

L'intérêt des valeurs alternatives est de permettre d'éliminer complètement des niveaux d'interprétation dans le processus d'évaluation partielle. En effet, une application courante de la spécialisation de programme consiste à spécialiser un évaluateur lorsque l'expression à évaluer est connue mais pas ses paramètres. Dans les langages fortement typés, ces évaluateurs nécessitent un codage des expressions et des valeurs manipulées. Usuellement, ces programmes utilisent un type universel pour coder les valeurs manipulées et gèrent des structures de données représentant des environnements. Dans ce contexte, les valeurs alternatives permettent de simplifier ces structures en éliminant les parties déterminées lors de la spécialisation. Ce mécanisme débouche sur une *spécialisation de types* où un type du programme original est remplacé par un ou plusieurs types simplifiés. Ainsi, sur un évaluateur, le type universel est transformé en instances plus simples et les structures représentant des environnements sont remplacées (dans le meilleur des cas) en produits. Du point de vue de l'efficacité, ces transformations permettent de gagner à la fois sur la création de ces valeurs et sur les tests auquels elles sont soumises.

L'introduction de ce type de valeurs pose problème si nous désirons utiliser les mêmes mécanismes que ceux des évaluateurs partiels existants. Nous avons donc cherché et trouvé un mécanisme original basé sur une notion d'*événement* qui semble bien adapté aux valeurs alternatives. Ce mécanisme diffère des mécanismes habituels en s'intéressant plus à ce qui a

permis d'activer telle expression du programme spécialisé, plutôt que de comparer les arguments de cette expression avec les autres expressions similaires. Le mécanisme d'identification de versions spécialisées d'un même objet du programme original repose davantage sur une notion dynamique (comment sommes-nous arrivé ici ?) que sur une notion statique (quelles sont les valeurs des arguments des versions d'un objet ?).

Ceci nous a amené à comprendre de manière plus profonde les transformations utilisées par les évaluateurs partiels et à nous intéresser à une transformation élémentaire appelée *clonage*. Elle consiste, à partir d'un objet initial, à le dupliquer et à lui assigner un index.

La conjonction du clonage, des valeurs alternatives et des événements semble bien adaptée à la spécialisation de programmes qui utilisent la notion d'environnement et de type universel. Le résultat ressemble, d'après les essais effectués, à une véritable compilation sur ce genre de programmes bien que certaines améliorations soient nécessaires pour atteindre cet objectif.

Nous avons l'objectif d'offrir un évaluateur partiel pour un langage fonctionnel fortement typé du type de ML. Dans ce travail, nous nous sommes contentés d'un langage du premier ordre (appelé CL) avec définition de type somme. Une mise en œuvre *LaMix* a permis de tester ces notions sur quelques exemples.

D'autre part, la genèse de ce travail provenant d'une tentative d'appliquer les principes de l'évaluation partielle aux réseaux d'interaction, la seconde partie de ce travail reprend les résultats obtenus avec ce langage. En effet, la notion de clonage de constructeur et de structure de contrôle ressemble au mécanisme d'abréviation dans les réseaux d'interaction. La notion d'événement introduite dans la première partie trouve aussi son analogue dans les réseaux d'interaction avec l'application d'une règle d'interaction.

Nous avons l'intention de reprendre le travail sur l'évaluation partielle des réseaux d'interaction en introduisant les concepts de valeurs alternatives et d'événements puisqu'ils semblent bien s'adapter aux réseaux. Cela permettra de rendre complètement automatique le mécanisme d'introduction d'abréviation dans les réseaux d'interaction.

La troisième partie est consacrée à LaMix, le programme d'évaluation partielle qui implante les idées abordées dans la première partie.

Table des matières

Première partie

Les valeurs alternatives et la notion d'événement dans l'évaluation partielle

Chapitre 1

Introduction

1.1 Principes de la transformations de programmes

La programmation semble de plus en plus confrontée à un dilemme fondamental entre efficacité et clarté[Kot80] :

- D'une part, la complexité des programmes nécessaires aux applications contemporaines de l'informatique ne cesse de croître. Le génie logiciel tente de réduire cette complexité par une modularisation des programmes. Il s'attache à diviser les problèmes en tâches indépendantes, plus facilement solvables. Le concept important porte sur la clarté des programmes. Un programme *bien écrit* sera plus facilement modifiable. Il sera plus facilement vérifié. Lisibilité, modularité, correction, tous ces critères tendent à produire des programmes dont la principale qualité n'est pas leur efficacité mais d'offrir une maquette simple à créer, à comprendre et à modifier.

- D'autre part, le programmeur peut s'intéresser au meilleur moyen pour résoudre un problème particulier. Le critère porte alors sur l'efficacité du programme. Deux programmes peuvent calculer la même fonction. Pourtant, l'un peut s'exécuter plus rapidement que le second.

Ces deux jugements sur la qualité des programmes sont souvent contradictoires. Prenons par exemple le problème du calcul du n-ième terme de la suite de Fibonnacci :

$$\begin{aligned} U_0 &= 1 \\ U_1 &= 1 \\ U_n &= U_{n-1} + U_{n-2} \quad n \geq 2 \end{aligned}$$

Nous pouvons directement traduire cette définition mathématique en un programme fonctionnel en utilisant une fonction récursive :

```
let rec fib n = match n with
          0 -> 1
        | 1 -> 1
        | _ -> fib(n-1) + fib(n-2) ;;
```

La fonction `fib` est facile à comprendre une fois que le programmeur a une petite notion de programmation fonctionnelle et du dialecte ML utilisé (CAML [WL93, Ler93]). La correspondance entre la mise en œuvre et la définition de la fonction de Fibonnacci est immédiate.

Toutefois, ce programme n'est pas satisfaisant du point de vue de l'efficacité. En effet, la complexité du calcul est exponentielle par rapport à son argument. Cette fonction calcule bien les termes de la suite mais en un temps inacceptable. Un programmeur écrira plutôt le programme suivant :

```
let rec fib_bis u_n u_p_n i = match i with
         0 ->  u_n
       | _ ->  fib_bis (u_n+u_p_n) u_n (i-1) ;;
let fib n = match n with
         0 -> 1
       | _ -> fib_bis 1 1 (n-1) ;;
```

Cette solution pour calculer les nombres de Fibonnacci est beaucoup plus satisfaisante que la première version car sa complexité est linéaire. Pourtant, nous avons perdu la clarté et la simplicité de la version précédente. Cet exemple illustre le dilemme existant entre programmes clairs et programmes efficaces.

Ce problème général forme la base des principes de la transformation automatique de programmes et de la compilation [BD77, Hog81, Fea86, BEJ87, JGS93]. La tendance actuelle pour la genèse des programmes tend à privilégier la clarté et les facilités de vérification plutôt que l'efficacité. Cette tendance est particulièrement sensible lors de la réalisation des premières maquettes d'un logiciel, avec des programmes fortement modulaires. En effet, il est toujours possible de transformer un programme peu efficace en une version plus rapide une fois obtenue une version qui fonctionne correctement. Le but des mécanismes de transformation de programmes et de la compilation consiste à rendre automatique et surtout *sûr* cette phase d'optimisation pour que le programme original et le programme transformé soient équivalents.

Ainsi, les deux versions du calcul de la suite de Fibonnacci sont équivalentes et nous pouvons donner les différents mécanismes qui nous permettent de transformer la première version en la seconde. Ils reposent principalement sur les propriétés algébriques de l'addition sur les entiers et sur la notion de fonctions.

1.1.1 L'évaluation partielle

L'*évaluation partielle* appartient à l'ensemble des techniques de transformations de programmes. Elle a déjà une longue histoire [LR64, Fut71, BD77]. Récemment, le groupe de DIKU, Copenhague, relança l'intérêt de cette technique [JSS85, JGS93] en divisant ce mécanisme en deux phases, une d'analyse de programme et la suivante de spécialisation (évaluateurs partielles dits *hors-ligne*). Les mises en œuvres sur ce sujet sont nombreuses [Bon90, Bon91b, Con88, Con90b, Lau89, Lau91a, WCRS91].

Son principe de base semble clair et simple bien que ses extensions puissent conduire à des mécanismes relativement complexes.

- Considérons un programme P définissant une fonction f_P de plusieurs arguments $x_1, \ldots, x_n$. Cette fonction peut servir, dans le cadre d'une programmation modulaire, à calculer d'autres fonctions dans lesquelles un certain nombre des arguments $x_1, \ldots, x_n$ sont fixés. Ainsi, à partir de la fonction d'addition, nous pouvons définir la fonction successeur en fixant un des deux paramètres à 1.

- Le but de l'évaluation partielle (nommée aussi *spécialisation de programmes*) consiste, à partir de la définition de la fonction f_P du programme P et des arguments fixés, à trouver une définition équivalente et plus efficace de f_P en tenant compte de la valeur des paramètres constants. Le terme de *spécialisation* dénote bien ce mécanisme en proclamant que la fonction

ainsi obtenue résulte de la transformation d'un algorithme général (celui du programme P) en un algorithme valable uniquement sous certaines conditions (certains paramètres ont une valeur particulière). Le terme d'évaluation partielle dénote aussi que les transformations appliquées sont issues d'un processus se rapprochant d'une évaluation classique mais restreinte aux opérations qui peuvent être calculées en ne connaissant que les valeurs fixées.

Un exemple simple de ce processus provient de la spécialisation de la fonction calculant une puissance d'un nombre lorsque l'exposant est fixé. Considérons la fonction `power` suivante :

```
let rec power x n = match n with
     0 -> 1
   | _ -> x * (power x (n-1)) ;;
```

L'expression `(power x n)` calcule x^n pour $n \geq 0$. Nous pouvons supposer que cette fonction est utilisée par un programme avec `n` fixé à 3. La fonction `cube` calcule le cube de son argument en utilisant la fonction `power` :

```
let cube x = power x 3 ;;
```

Spécialisation de fonctions

La fonction `cube` appelle toujours `power` avec 3 comme deuxième argument. Nous pouvons spécialiser cette fonction pour cette valeur et obtenir la fonction `power_3` :

```
let power_3 x n = x * (power x 2) ;;
```

Cette définition induit la spécialisation de `power` avec 2 comme exposant qui, elle même, induit la spécialisation pour 1 puis 0. Le programme *résiduel* est :

```
let power_0 x n = 1 ;;
let power_1 x n = x * (power_0 x 0) ;;
let power_2 x n = x * (power_1 x 1) ;;
let power_3 x n = x * (power_2 x 2) ;;
let cube x = power_3 x 3 ;;
```

Ainsi, nous avons créé quatre versions spécialisées de `power` en calculant les expressions qui peuvent l'être par la seule donnée de la valeur 3 et en réduisant les branches des structures de contrôle dont la valeur testée est connue. Cette phase montre clairement le principe de l'évaluation partielle.

Élimination des arguments constants

Mais, nous pouvons aller plus loin dans les transformations du programme original. En effet, le paramètre `n` des versions spécialisées de `power` n'est jamais utilisé dans le corps de ces fonctions. Nous pouvons l'éliminer :

```
let power_0 x = 1 ;;
let power_1 x = x * (power_0 x) ;;
let power_2 x = x * (power_1 x) ;;
let power_3 x = x * (power_2 x) ;;
let cube x = power_3 x ;;
```

Dépliage de fonctions

De plus, nous pouvons *déplier* ces fonctions en remplaçant l'appel à une fonction spécialisée par le corps de cette fonction. Une version où les fonctions sont dépliées est :

```
let cube x = x * (x * (x * 1)) ;;
```

Nous voyons que cette version de `cube` est plus efficace que la version utilisant `power` car elle ne contient que les opérations sur `x`, ne fait plus référence à `n` et ne contient pas d'appel. Ainsi, grâce aux mécanismes de spécialisation de fonctions, d'évaluation des expressions calculables, d'élimination de paramètres inutilisés et de dépliage de fonctions, nous avons obtenu un programme plus efficace et strictement équivalent à la fonction d'origine. Une version un peu plus intelligente de ce mécanisme aurait pu s'apercevoir que la multiplication par 1 est inutile et aurait donné le programme :

```
let cube x = x * (x * x) ;;
```

Limite de l'évaluation partielle

Cet exemple montre l'intérêt de l'évaluation partielle comme processus de transformation d'un programme en un programme plus efficace. Toutefois, cette technique a des limites qui proviennent des transformations de programme qui peuvent être considérées comme appartenant à l'évaluation partielle. Ainsi, dans notre premier exemple sur la fonction de Fibonnacci, nous devions utiliser les propriétés algébriques de l'addition pour passer de la version simple à la version efficace ce qui sort du cadre de l'évaluation partielle. En fait, ce problème n'est pas tant lié aux transformations de programme pouvant s'appliquer et mener à un *bon* programme que le moyen de découvrir de manière automatique quelles sont les transformations qui déboucheront sur un programme efficace. L'évaluation partielle s'attache plus à résoudre ce dernier problème que d'offrir un ensemble vaste de transformations. Cette *incompétence* relative est toutefois compensée par l'automatisation accrue de cette technique, sa simplicité et sa rapidité qui lui permet de s'appliquer à des programmes réalistes et de taille importante.

De plus, l'évaluation partielle repose sur la transformation d'un programme écrit dans un certain langage en un autre programme écrit dans le même langage. Elle diffère donc de la compilation où le langage objet est différent du langage source. Ainsi, certaines optimisations résultant de la compilation d'un programme dans un langage objet ne peuvent pas s'appliquer car elles ne correspondent à aucune construction du langage source.

Supercompilation et déforestation

D'autres approches transformationnelles basés sur les mêmes transformations existent comme la *déforestation* [Wad88, FW88], la *supercompilation* [Tur79, Tur86] et le *calcul partiel généralisé* [FN88, FNT91, Tak91].

La *déforestation* essaye d'éliminer les structures de données intermédiaires, réduisant ainsi le nombre de phases de calcul. Ainsi, si nous considérons l'expression `append (append x y) z` où `append` est la fonction concaténant deux listes, nous pouvons constater que l'expression `append x y` construit une liste intermédiaire qui est détruite par le second `append`. La déforestation consiste ici à fusionner les deux fonctions `append` pour ne faire qu'une récurrence sur `x` puis sur `y`.

La *supercompilation* est une technique puissante combinant l'évaluation partielle et la déforestation. Elle repose sur deux principes. L'un, la *conduite* (*driving*), propage les informations sur les expressions et l'autre, la *généralisation* (*generalization*), tente de *plier* les expressions pour retrouver de nouvelles définitions de fonctions. L'article [GK93] exprime le concept de supercompilation dans un langage *à la Lisp*.

Le *calcul partiel généralisé*, d'abord appliqué à la programmation logique puis à la programmation fonctionnelle, est aussi puissant que la supercompilation.

Une comparaison de ces techniques est présentée dans [SGJ94]. Dans [GJ94b] et [GJ94a], les auteurs démontrent que ces transformations peuvent être obtenues par spécialisation d'un interpréteur.

1.1.2 Transformations de programme

Comme dans toute approche transformationnelle, la spécialisation de programme repose sur un certain nombre de transformations de programmes. Nous pouvons discerner les transformations suivantes :

- Remplacer une expression dont la valeur est calculable par sa valeur. Ainsi, dans la fonction `power_3`, `n` vaut 3 et l'expression `(n-1)` peut être remplacée par 2. Ce principe découle du terme *évaluation partielle* puisque ces expressions sont calculées.

- Remplacer l'appel à une fonction par un appel à une fonction spécialisée lorsque la valeur de certains arguments est connue. Dans l'exemple du cube, l'appel à `power` dans `cube` est remplacé par un appel à `power_3` spécialisant la fonction puissance pour un exposant égal à 3. Ce principe suit la notion de *spécialisation de fonctions* puisque nous remplaçons l'utilisation d'une fonction générale par une fonctions dont le contexte est restreint.

- Éliminer une structure de contrôle dont la valeur testée est connue par la branche activée. Ainsi, l'expression :

```
match n with
       0 -> 1
     | _ -> x * (power x (n-1)) ;;
```

  de `power_3` est remplacée par `power x (n-1)` puisque `n` vaut 3. Ici, nous pouvons parler de *spécialisation de structure de contrôle*.

- Éliminer les expressions, les paramètres et les variables inutilisés. Ainsi, dans le programme du cube, le paramètre `n` est éliminable.

- Remplacer l'appel à une fonction par le corps de cette fonction. Dans l'exemple du cube, les appels aux fonctions `power_3` ... `power_0` ont disparu de la version finale de `cube`. Ce processus s'appelle *dépliage d'appels de fonctions* [BD77].

Ces transformations de programmes forment le noyau initial de l'évaluation partielle. D'autres transformations ont été ajoutées, petit à petit, pour résoudre des problèmes posés par des programmes qu'une version précédente ne pouvait pas appréhender.

Elles proviennent, par exemple, de la notion de *valeurs partiellement connues*. Ainsi, une paire dont seule la première composante peut être connue lors de la phase de spécialisation ne peut pas être traitée par les évaluateurs partiels basés sur le noyau initial de transformations. Le terme de *structures partiellement connues* [Mog88, Bon88] fut inventé pour désigner ces valeurs dont une partie seulement est calculable.

D'autres transformations sont apparues pour traiter les fonctions d'ordre supérieur et ajouter la notion de fermeture aux objets spécialisables [Bon90, Con90a, Gom90, Hen91, NN88].

Enfin, certaines transformations ont été divisées en transformations plus élémentaires pour résoudre certains problèmes de duplication de code. Ainsi, le dépliage d'appels de fonctions a été

divisé en un mécanisme transformant un appel en une définition de variables locales puis une éventuelle simplification de cette définition de variables [Bon90]. Ainsi, avec la spécialisation de `cube`, nous pouvons déplier la fonction `power_3` de `cube` dans le programme :

```
let power_3 x n = x * (power_2 x 2) ;;
let cube x = power_3 x 3 ;;
```

et obtenir :

```
let cube x = let x = x and n = 3 in x * (power_2 x 2) ;;
```

Les paramètres de `power_3` donnent la définition de deux variables locales. Une phase ultérieure simplifie cette structure en s'apercevant que `n` est inutilisée et que la variable locale `x` est dépliable ce qui donne le résultat équivalent au dépliage direct de `power_3`.

Une autre transformation générale consistant à transformer le programme dans le style par continuations [Plo75] permet aussi de résoudre des problèmes de perte d'information [Dan91, Jør91, CD91, Bon91b, Jør92]. En effet, elle transforme un programme de telle manière que le seul résultat renvoyé par les fonctions est le résultat final. Ainsi, les pertes d'informations sur une valeur ne peuvent résulter du fait qu'elle est renvoyée par une fonction spécialisée ou une structure de contrôle dont la valeur de test ne peut pas être déterminée lors de la phase d'évaluation partielle. Cette propriété est très importante dans les évaluateurs partiels actuels car ces systèmes ne savent pas traiter ce type de constructions résiduelles.

D'autres transformations, plus anecdotiques, résolvent des problèmes pouvant se produire avec certains programmes. Ainsi, sur le programme :

```
let f x y = (if x then 1 else 2) + y ;;
```

Lorsque `x` est inconnu mais que `y` est constant, 3 par exemple, la valeur testée par la conditionnelle est inconnue et nous ne pouvons pas spécialiser correctement cette fonction. L'utilisation de la commutation sur les structures de contrôle (les réductions commutatives [GLT89]) qui permet de rentrer l'opération d'addition à l'intérieur du `if then else` résout ce problème [CD91]. En effet, ce programme est équivalent au programme :

```
let f x y = if x then 1+y else 2+y ;;
```

Les additions sont calculables et nous obtenons le programme :

```
let f x y = if x then 4 else 5 ;;
```

1.2 Motivations de ce travail

Dans ce travail, nous avons désiré résoudre un certain nombre de problèmes de généralisation de valeurs (perte d'information) lors de l'évaluation (partielle) des expressions. Cela nous a conduit à introduire la notion de *valeurs alternatives*. Cette notion nous a amené à trouver un mécanisme d'évaluation partielle, basé sur les notions de *clonage* et d'*événement*. Il permet de gérer ces valeurs qui, à notre avis, ne peuvent pas être appréhendées par les mécanismes actuels d'évaluation partielle.

Ce travail s'inscrit aussi dans l'optique d'une meilleure compréhension des transformations de programmes utilisées par l'évaluation partielle. Il autorise une décomposition de ce mécanisme en mécanismes plus élémentaires dont la résolution est indépendante des autres. Ainsi, nous distinguons les phases de *clonage*, de *fusion/généralisation* et de *spécialisation*.

1.2.1 Valeurs alternatives

Les évaluateurs partiels ont progressivement amélioré la description des valeurs associées aux expressions d'un programme à spécialiser.

Valeurs statiques/dynamiques

Ainsi, les premières versions ne pouvaient distinguer que les expressions entièrement calculables lors de l'évaluation partielle, des autres expressions [JSS85]. Dans le premier cas, la valeur associée à l'expression est naturellement la valeur constante résultant de la réduction de cette expression. Dans le second cas, l'expression est laissée *résiduelle*, et la valeur associée est considérée comme totalement inconnue. Une valeur est dite *statique* si sa valeur est calculable et *dynamique* dans le cas contraire. Les expressions calculables sont dénommées *réductibles*, les autres, *résiduelles*. Dans cette première version, nous avons une séparation nette entre une partie calculable sur laquelle l'évaluateur partiel travaille et une partie résiduelle qui est laissée tel quel. Les programmes bien adaptés à ce type d'évaluateurs partiels doivent séparer clairement la partie réductible de la partie résiduelle.

Valeurs partiellement connues

Dans une seconde phase, les évaluateurs pouvaient introduire des *valeurs partiellement connues* [Mog88, Bon88] comme une paire dont seule une des composantes est calculable ou bien une liste dont la longueur est calculable mais pas les éléments. Cette extension permet de traiter des types de programmes où la distinction entre partie réductible et partie résiduelle n'est pas aussi nette qu'avec le premier type d'évaluateurs partiels. De plus, les mécanismes pour arriver au résultat ne forment qu'une extension des mécanismes des premiers évaluateurs.

Autres valeurs

À la suite de cette génération d'évaluateurs partiels (ou de manière indépendante), d'autres notions et d'autres transformations de programmes ont été introduites pour répondre à des problèmes soulevés par des types de programmes mal traités par ces évaluateurs (passage au style par continuations [CD91], spécialisation de constructeurs [Mog93]) ou pour étendre le mécanisme à d'autres constructions (fonctions d'ordre supérieure, polymorphisme) [Bon90, Con90a, Gom90, Hen91, NN88].

Toutes ces améliorations ont pour but de traiter une plus grande classe de programmes. Elles tendent à offrir une meilleur séparation entre la partie statique et la partie dynamique. En particulier, elles tentent à faire disparaître les points où des valeurs doivent être généralisées car provenant de plusieurs sources. Cette remarque s'applique notamment aux structures de contrôle devant rester résiduelles lorsque la valeur testée ne peut pas être connue lors de l'évaluation partielle. Dans ce cas, la valeur associée au résultat de cette conditionnelle peut provenir du résultat de toutes les branches résiduelles. Les évaluateurs partiels existants décident généralement que cette valeur est totalement inconnue. Une autre source de perte d'informations provient des fonctions dont le résultat n'est pas complètement connu (elles ne peuvent pas être complètement calculées) et ne pouvant pas être dépliées.

Valeurs alternatives

Dans ce travail, nous avons voulu résoudre ce problème en introduisant un nouveau type de valeurs appelées *valeurs alternatives*. En effet, tous ces problèmes arrivent lorsqu'une valeur peut

provenir de plusieurs expressions comme des différentes branches d'une conditionnelle. Nous devons trouver une approximation de cette valeur à partir des valeurs des différentes sources. Avec les valeurs partiellement connues, cette description ne peut être que la partie commune à toutes les valeurs ce qui, en pratique, se traduit par un manque complet d'information (sauf, par exemple, dans [Bec94] ou dans les développements de [Lau91a]).

Considérons la fonction `f` suivante :

```
let f x y = (if x then 1 else 2) + y ;;
```

Lorsque `x` est inconnu, nous ne pouvons connaître la branche de la conditionnelle qui va être activée. Par conséquent, la valeur retournée par cette expression peut être 1 ou 2. Or, le domaine des valeurs partiellement connues ne peut décrire qu'une valeur puisse être égale à 1 ou à 2 et le résultat de cette conditionnelle doit être généralisé.

Les valeurs alternatives permettent de remédier à ce problème. Ici, la description de la valeur retournée par la conditionnelle serait une alternative entre les deux valeurs constantes 1 et 2. D'une manière plus précise, les valeurs alternatives décrivent des ensembles de valeurs par des grammaires hors-contexte où les symboles sont des *clones de constructeurs* (ou de constantes) et les règles, des spécialisations des déclarations des types associés à ces constructeurs. Un élément $\top$ est ajouté pour décrire toute les valeurs possibles d'un type donné.

Ainsi, ces valeurs peuvent décrire des valeurs complètement calculables (grammaire décrivant un seul élément, sans $\top$) ou complètement inconnues ($\top$) comme pour les premiers évaluateurs partiels. Mais, aussi des valeurs partiellement connues (grammaire décrivant un élément, avec $\top$ apparaissant comme terminal). Les valeurs alternatives peuvent aussi décrire des ensembles de valeurs plus originaux comme un ensemble fini de constantes, une liste de longueur inconnue mais dont les éléments sont connus, une liste de longueur inconnue mais se terminant toujours par une liste connue...

Élimination de niveaux de codage dans les interpréteurs

L'intérêt des valeurs alternatives apparaît lorsque le processus d'évaluation partielle s'applique à des évaluateurs (des interpréteurs), que l'expression évaluée (le programme) est connue mais pas ses arguments. En effet, une méthode simple et usuelle d'écrire ces programmes consiste à utiliser un type universel pour représenter les valeurs manipulées et à gérer des structures représentant des environnements. Cette méthode nécessite un codage des valeurs et des tests dynamiques pour vérifier que la représentation correspond à une valeur valide pour une opération [Lau91b]. Par exemple, lorsqu'un évaluateur doit exécuter une addition, il vérifie que les arguments représentent des nombres. L'introduction des valeurs alternatives va permettre d'éliminer (ou de simplifier) ce codage et les tests sur ces valeurs.

Prenons, par exemple, un interpréteur manipulant des valeurs atomiques (entiers, symboles, chaînes de caractères, booléens) ou des paires de valeurs (comme les "cons" de Lisp); les environnements correspondant à des listes d'association liants des variables à leur valeur. Cet interpréteur utilise un type universel pour les valeurs et un type pour les environnements similaires à :

```
type Value = Int of int
           | Symbol of string
           | String of string
           | Bool of bool
           | Cons of Value * Value
           | Nil ;;
type Env   = Empty
```

```
            | Bind of string * Value * Env ;;
```

Le code exécutant une addition pourrait s'écrire :

```
let make_add = fun
      (Int i1) (Int i2) -> Int(i1+i2)
    |    _        _     -> error"Type mismatch" ;;
```

Cette fonction teste si les deux valeurs correspondent à des entiers puis renvoie une représentation de leur somme.

Alors, si, au cours de l'évaluation partielle, nous pouvons déterminer que les paramètres de `make_add` sont associés à une valeur alternatives construites avec le seul constructeur `Int`, nous pouvons, d'une part transformer le code qui produit ces valeurs en éliminant le constructeur d'entier et d'autre part éliminer les tests sur `v1` et `v2`. De même, le résultat de cette fonction pourrait aussi renvoyer directement la somme des deux entiers plutôt la somme avec le constructeur. Ainsi, cette fonction pourrait se transformer en la définition suivante :

```
let make_add = fun v1 v2 -> v1+v2 ;;
```

Le mécanisme qui autorise cette transformation se base sur l'élimination de constructeur [Lau91a]. De notre point de vue, cette opération, dans le contexte des langages fortement typés, consiste à remplacer les types qui ne possèdent qu'un seul constructeur par le type de l'argument de ce constructeur. Ainsi, sur l'exemple ci-dessus, la descrition associée aux paramètres de `make_add` débouche sur une spécialisation du type `Value` composée d'un seul constructeur :

```
type Value_0 = Int_0 of int ;;
```

Ce type est isomorphe au type `int`, ce qui justifie la transformation de `Value_0` en `int` et l'élimination du constructeur `Int_0`.

Le concept de valeurs alternatives entraine la notion de type spécialisé puis la simplification de ces types en éliminant les constructeurs inutiles et les composantes constantes des produits.

Dans ce cadre, la notion de valeurs alternatives est une étape dans le mécanisme de spécialisation de type.

1.2.2 Notion d'événement

Une fois ce type de valeurs introduit, nous devons trouver les transformations de programme qui peuvent s'appliquer à ces valeurs et trouver aussi un mécanisme de spécialisation de programme qui peut les appréhender. Comme nous le verrons dans le chapitre sur le *clonage de fonctions*, nous n'avons pas trouvé un mécanisme similaire à ceux des évaluateurs partiels existants. Aussi, le mécanisme présenté dans ce travail repose-t-il sur une notion originale d'*événement* qui, nous le pensons, est bien adaptée à ce type de valeurs et qui ouvre la voie à une approche différente de l'évaluation partielle.

Cette notion d'événement repose sur la gestion des constructeurs qui ont été utilisés pour *activer* un point du programme et s'assimile à une sorte d'historique. Alors que les mécanismes actuels de spécialisation repose sur la comparaison des valeurs associées aux arguments d'un point du programme en cours de spécialisation. Nous sommes donc passés d'une vision statique (les valeurs) à une vision plus dynamique (consommation de valeurs). Ainsi, dans les approches traditionnelles, une fonction spécialisée est identifiée avec une autre lorsque les valeurs de leurs paramètres sont identiques. Dans ce travail, le mécanisme d'identification est plus complexe car il repose sur une analyse des événements qui ont conduit à activer ces deux fonctions.

1.2.3 Clonage

Cette approche a rendu possible l'utilisation des valeurs alternatives. De plus, cette notion nous a permis de comprendre de manière plus fine les mécanismes de spécialisation de programme en offrant la possibilité de découper le problème initial en sous-problèmes relativement indépendants. Cela nous a amené à nous concentrer sur la notion de *clonage* présentée longuement dans les chapitres suivants et qui consiste, grossièrement, à dupliquer la définition des fonctions et, plus généralement, des objets du programme original. Le clonage peut être vu comme une transformation de programme très élémentaire pouvant apparaître comme une partie des transformations impliquées dans l'évaluation partielle. Par conséquent, cette notion nous a permis de comprendre plus intimement le mécanisme de spécialisation en offrant un objectif plus simple que l'obtention direct d'un programme spécialisé plus efficace que le programme original. Cette phase a pu être rendu indépendante des autres problèmes et nous semble être la phase cruciale dans le processus d'évaluation partielle.

1.2.4 Évaluation en et hors ligne

Ces dernières années ont semblé privilégier une méthode de spécialisation dite *hors ligne* [JSS85, JGS93]. Ce principe consiste à diviser le mécanisme d'évaluation partielle en deux phases. Dans la première, le programme est analysé puis *annoté* en sachant quels sont les paramètres connus (mais pas leur valeur). Cette phase repose sur une *analyse des temps de liaison* [Bon91a, Con90a, NN88, Gom90, Hen91]. Puis, une phase de *spécialisation*, dirigée par les annotations du programme et la valeur des paramètres connus, transforme rapidement le programme source. Dans ce cadre, l'évaluateur partiel tente d'utiliser le moins possible les valeurs calculées pour prendre ses décisions. A l'inverse, la méthode *en ligne* utilise activement les valeurs calculées [CK92, CD93].

Les avantages de la méthode hors ligne sont d'offrir un algorithme plus rapide de spécialisation, de diviser les problèmes en deux et surtout d'essayer de se rapprocher du concept d'*auto-application*. Ceci est connu sous le nom des trois projections de Futamura [Fut71]. Futamura réalisa que l'évaluation partielle pouvait servir à dériver des compilateurs à partir d'interpréteurs et mieux encore à la réalisation de générateurs de compilateurs [Tur79]. L'idée consiste simplement à appliquer un évaluateur partiel à lui-même.

Ceci s'est traduit par l'abandon des méthodes en ligne car les versions hors lignes requièrent un évaluateur partiel moins complexe (débarrassé de nombreux choix par la phase d'analyse) et, par conséquent, répondant mieux au critère de l'auto-application [JSS85].

Les idées présentées dans ce travail semblent mal s'adapter aux méthodes hors lignes. En effet, nombres de décisions importantes (comme la notion d'événement) reposent sur les valeurs calculées (et notamment celles des paramètres connus). Notre but, ici, est d'introduire un nouveau mécanisme d'évaluation partielle plutôt que d'améliorer les techniques actuelles.

1.2.5 Supercompilation et déforestation

En comparaison avec les techniques de déforestation [Wad88, FW88], de *supercompilation* [Tur79, Tur86] et de *calcul partiel généralisé* [FN88, FNT91, Tak91], le mécanisme présenté dans cette thèse semble moins puissant sur certains points (au moins d'un point de vue théorique) mais original en ce qui concerne les valeurs alternatives.

Par rapport à la déforestation qui consiste à fusionner des phases de calcul pour éliminer des résultats intermédiaires, les trois concepts de valeurs alternatives, d'événement et de clonage ne peuvent pas, en général, éliminer la construction de valeurs intermédiaires. Toutefois, par rapport à une évaluation classique, le mécanisme d'évaluation partielle essaye de simplifier les résultats

intermédiaires (par exemple, en éliminant des constructeurs inutiles ou en simplifiant des produits). En particulier, sur les exemples d'évaluation partielle d'interpréteurs cela débouche sur la définition de nouveaux types pour représenter les valeurs manipulées par ces programmes.

La grande différence entre le clonage et la supercompilation provient de la non-propagation des informations que l'on peut déduire sur les valeurs des variables d'un programme à partir du contexte où elles sont utilisées. Ainsi, sur une structure de contrôle telle que :

```
match x with
     h::t -> hd(x)
   |  []  -> x
```

où `x` n'est pas connu, la supercompilation pourrait déduire que dans `h::t -> hd(x)`, `hd(x)` est égale à `h` et que dans `[] -> x`, `x` est la liste vide. Cela n'est pas réalisé dans ce travail.

La technique de *calcul partiel généralisé* est similaire, en substance, à la supercompilation et introduit les mêmes critiques par rapport à ce travail.

Nous pensons ajouter des mécanismes pouvant résoudre ces problèmes.

1.2.6 Autres domaines étendus

L'évaluateur partiel REDFUN-2 [Har77] utilise un domaine de valeurs un peu similaire aux valeurs alternatives. Avec ce système, la valeur associée à une variable du programme spécialisé peut être :

- soit `NOBIN`, un symbole spécifiant que la valeur est complètement inconnue,
- soit la liste de constantes que peut prendre cette variable,
- soit la liste de constantes que la variable ne peut pas avoir,
- soit un descripteur de type de données (atome, liste, entier...).

Par rapport aux valeurs alternatives, ce domaine ne décrit pas les valeurs que peuvent prendre une expression comme une structure arborescente d'alternatives (comme le domaine des valeurs alternatives) mais comme une alternative entre plusieurs constantes. Ainsi, REDFUN-2 peut savoir qu'une variable peut prendre 1,2 ou 3 comme valeur, ou que sa valeur est un entier. Par contre, il n'a pas de d'équivalent des valeurs partiellement connues et encore moins, un moyen de décrire qu'une expression est, par exemple, une liste de longueur paire.

1.3 Classe des programmes spécialisés

Ce travail s'inscrit dans la continuation d'un travail sur l'évaluation partielle des réseaux d'interaction [Bec92]. Les concepts résultent de l'action réciproque entre la logique linéaire [Gir87] et les réseaux d'interaction [Laf90] et le monde de l'évaluation partielle. Notre objectif était d'appliquer l'évaluation partielle aux réseaux. Cela nous a conduit à définir des mécanismes de transformation originaux. Ce travail résulte du retour de ces concepts à la programmation fonctionnelle ou applicative plus familière de la communauté s'intéressant à la spécialisation de programmes. Notre objectif consiste à créer un évaluateur partiel pour un langage fonctionnel fortement typé dans le style du langage ML.

Nous avons notamment le désir de nous intéresser aux interpréteurs et à leur spécialisation automatique lorsque le programme interprété est connu lors de l'évaluation partielle mais pas ses données. Cette classe de programmes a fortement influencé la littérature sur l'évaluation partielle car

ces programmes semblent bien adaptés aux mécanismes d'évaluation partielle. Dans cette optique, ce travail tente de trouver des réponses à des lacunes existantes en ce domaine. En particulier, nous aimerions que l'évaluation partielle d'un interpréteur, lorsque le programme interprété est connu, ressemble à une véritable compilation du programme interprété. Cet objectif ne peut être achevé qu'en introduisant la notion de spécialisation de types de données et des environnements car les interpréteurs utilisent généralement un type de données universel (pour les entiers, les listes, les symboles...) et travaillent avec des environnements liant un symbole du programme interprété à une valeur. Nous désirons qu'un évaluateur partiel traite correctement ces deux concepts. Le problème des environnements a reçu un certain nombre de réponses dans les évaluateurs partiels existants qui ne sont pas toujours satisfaisantes. Le premier problème (spécialisation de types de données) reste encore presque vierge de toute approche satisfaisante. Dans ce travail, nous avons proposé un mécanisme répondant à ces deux problèmes.

1.4 Plan de la thèse

- Le chapitre 2 présente le langage CL sur lequel nous travaillons.

- Comme le noyau de ce travail concerne les notions de clonage, de valeurs alternatives et d'événement, deux chapitres sont consacrés à leur description.

  Le chapitre 3 aborde la notion de clonage et la confronte aux évaluateurs partiels contemporains. Puis, nous présentons la notion de valeurs alternatives et justifions leur utilisation comme une extension des valeurs utilisées par les évaluateurs partiels existants (les valeurs partiellement connues). Ceci nous amène à montrer l'inadéquation des mécanismes de spécialisation pour ce type de valeurs et justifie la recherche d'un nouveau processus pour les traiter. La fin de ce premier chapitre présente brièvement ce nouveau mécanisme qui est basé sur une notion d'*événement*.

  Dans le chapitre 4, nous présentons en détail la notion d'événement et le mécanisme de clonage utilisé. Ce chapitre forme l'essentiel de ce travail.

- Le chapitre 5 se préoccupe de la correction du programme cloné par rapport au programme original et présente un mécanisme de *fusion/généralisation* permettant d'obtenir un programme correct, équivalent au programme original. Ce mécanisme débouche sur le concept de *type spécialisable* en rassemblant les clones de constructeur en classes et en familles créant de nouveaux types. Ce sont ces types qui seront simplifiés par la phase suivante en transformamt les types ne comportant qu'un constructeur.

- Le chapitre 6 donne toute la justification de l'approche utilisée en présentant les transformations de programmes qui permettent d'obtenir un programme plus efficace que le programme original. Ces transformations se basent sur la spécialisation de types et notamment sur la transformation d'un type somme en type produit et sur la transformation des expressions démontrées constantes en une référence vers une constante. Les autres transformations concernent le problème de *dépliage de fonction* qui remplace un appel à une fonction par le corps de cette fonction, la simplification de certaines constructions (déclaration de variables locales) et l'élimination de paramètres ou de variables inutiles. Toutes ces transformations peuvent être vues comme des optimisations locales presque indépendantes du processus de clonage.

  Le chapitre 7 conclut en présentant les extensions possibles. Notre objectif est d'offrir un évaluateur partiel pour un langage fonctionnel du type de ML. Cet objectif semble assez

facilement réalisable dans l'état actuel de nos recherches (fonctions d'ordre supérieur, polymorphisme et types de données mutables, programmation avec effets de bord) par l'utilisation conjointe des valeurs alternatives et de la notion d'événement. Cette voie semble assez prometteuse.

- La seconde partie reprend le travail effectué au début de la thèse. Elle est consacrée à l'évaluation partielle dans les réseaux d'interaction. Ce travail est indépendant de la première partie mais nous avons désiré l'inclure dans ce travail car beaucoup d'idées soutenant les concepts de clonage, de valeurs alternatives et d'événement proviennent de la confrontation entre l'évaluation partielle et les réseaux d'interaction. De plus, nous souhaitons continuer le travail entrepris sur les réseaux en introduisant les mécanismes d'évaluation partielle présentés dans la première partie.

- Enfin, les annexes décrivent une implantation des idées de la première partie nommée LaMix. L'annexe A décrit le langage LaML reconnu par LaMix. L'annexe B donne une brève description de l'installation de LaMix. Enfin, l'annexe C rapporte les essais effectués avec LaMix sur différents programmes (notamment ceux apparaissant comme exemples dans la première partie).

Chapitre 2

CL : un petit langage fonctionnel

L'évaluation partielle est un algorithme de transformations de programme. Ce chapitre décrit le langage reconnu par notre spécialiseur. Il n'a pas la prétention d'être un langage très expressif, ni très commode à utiliser mais permet de concentrer les problèmes sur un petit ensemble de constructions. Nous pouvons le considérer comme un langage intermédiaire entre un langage plus *réaliste* et notre évaluateur partiel.

La syntaxe et la sémantique du langage seront présentés. La section suivante abordera la notion de programme étiqueté qui sera utilisée dans les chapitres suivants. Enfin, un interpréteur pour TP, un petit langage impératif, écrit en CL, sera introduit. Ce programme ainsi que des fonctions sur la concaténation de listes illustreront la spécialisation de programmes dans les chapitres suivants.

2.1 Syntaxe du langage CL

Cette section décrit la syntaxe abstraite et la sémantique du langage CL qui servira de support au mécanisme de clonage présenté dans ce travail.

CL est un petit langage applicatif avec appel par valeur et liaisons statiques, avec définition de types concrets dans le style des langages ML [WL93, Ler93]. Toutefois, il est restreint au premier ordre (les arguments d'une fonction ne peuvent être une fonctionnelle), sans polymorphisme et avec des restrictions sur les motifs des structures de contrôle.

Un programme CL est une liste de déclaration de fonctions globales et de déclaration de type de données.

Soient les classes de symboles suivantes :

$\mathcal{T}$	Symboles de types
$\mathcal{F}$	Symboles de fonctions
$\mathcal{X}$	Symboles de variables locales et de paramètres
$\mathcal{C}$	Symboles de constructeurs
$\mathcal{O}$	Symboles de primitives

La syntaxe abstraite des programmes est décrite par la grammaire suivante :

p	$\rightarrow$	$d_1, \ldots, d_n$	
d	$\rightarrow$	`def` $f\ x_1\ \cdots\ x_n$ `=` e	$f \in \mathcal{F}, x_k \in \mathcal{X}, arity(f) = n$
	$\mid$	`type` t `=` $c_1(T)$ `|` $\cdots$ `|` $c_n(T)$	$t \in \mathcal{T}, c_k \in \mathcal{C}$
T	$\rightarrow$	t	$t \in \mathcal{T}$
	$\mid$	$T * \cdots * T$	
e	$\rightarrow$	`var` x	$x \in \mathcal{X}$
	$\mid$	`cons` $c(e)$	$c \in \mathcal{C}$
	$\mid$	`oper` $o\ e_1\ \cdots\ e_n$	$o \in \mathcal{O}, n = arity(o)$
	$\mid$	`call` $f\ e_1\ \cdots\ e_n$	$f \in \mathcal{F}, n = arity(f)$
	$\mid$	`let` $(x_1, e_1), \ldots, (x_n, e_n)$ `in` e	$x_k \in \mathcal{X}$
	$\mid$	`match` e `with` $c_1(x_1)$ `->` e_1 `|` $\cdots$ `|` $c_n(x_n)$ `->` e_n	$c_k \in \mathcal{C}, x_k \in \mathcal{X}$
	$\mid$	$\pi^n_m(e)$	$1 \leq m \leq n$
	$\mid$	$(e_1, \ldots, e_n)$	$n \geq 0$

La fonction *arity* renvoie l'arité associée au symbole correspondant. Un programme P est une liste de définitions de fonctions. Chaque fonction se caractérise par une liste de paramètres et une expression. Les expressions sont formées avec les constructions :

- `var` x : accès à la variable locale ou au paramètre x.
- `cons` $c(e)$: application du constructeur c. L'arité des constructeurs est toujours un (l'argument d'un constructeur constant est supposé être le produit vide noté ()).
- `oper` $o\ e_1\ \cdots\ e_n$: application d'une primitive. Les arguments sont présentés sous la forme curryfiée.
- `call` $f\ e_1\ \cdots\ e_n$: appel à la fonction f. Les arguments sont présentés sous la forme curryfiée.
- `let` $(x_1, e_1), \ldots, (x_n, e_n)$ `in` e : déclarations locales. Cette construction évalue $e_1, \ldots, e_n$ puis évalue e dans un environnement où les variables $x_1, \ldots, x_n$ sont liées aux valeurs calculées pour $e_1, \ldots, e_n$.
- `match` e `with` $c_1(x_1)$ `->` e_1 `\|` $\cdots$ `\|` $c_n(x_n)$ `->` e_n : structure de contrôle. Cette construction évalue e puis, si le résultat correspond au constructeur c_k, évalue la branche k avec un environnement où la variable x_k est liée à l'argument du constructeur c_k. Si aucun motif ne correspond à la valeur calculée pour e, le programme s'arrête en produisant une erreur de filtrage.
- $\pi^n_m(e)$: projection sur la m-ième composante d'un produit de n éléments.
- $(e_1, \ldots, e_n)$: produit de n valeurs.

Nous supposons que les programmes sont bien typés et que leur exécution ne peut produire aucune erreur de type. Cela suppose que cette propriété ait été vérifiée lors d'une phase antérieure.

2.2 Sémantique de CL

La sémantique dénotationnelle de ce langage est standard. Trois domaines sont nécessaires pour décrire la sémantique d'un programme. Ils correspondent aux valeurs $v \in Val$, aux environnements

de variables locales et de paramètres $\rho \in Xenv$ et aux environnements de fonctions $\phi \in Fenv$:

$$\begin{array}{lcl} Val & = & \texttt{C}\ (C \times Val) \mid \texttt{P}\ (Val \times \cdots \times Val) \\ Xenv & = & \mathcal{X} \to Val \\ Fenv & = & \mathcal{F} \to (Val \to \cdots \to Val \to Val) \end{array}$$

Trois fonctions permettent de définir la sémantique d'un programme CL.

- La fonction $(\phi, e, \rho) \mapsto E\,[\![e]\!]^{\phi}_{\rho}$ donne une signification à une expression e dans les environnements ρ et ϕ.
- La fonction $P \mapsto D\,[\![P]\!]$ construit l'environnement de fonctions associé au programme P.
- Pour les opérateurs prédéfinis, la fonction $o \mapsto O\,[\![o]\!]$ renvoie la fonction sémantique associée à la primitive o.

$$D\ :\ Prog \to Fenv$$

$$D\left[\!\!\left[\begin{array}{l} \texttt{def}\ f^1\ x^1_1\ \cdots\ x^1_{a^1}\ \texttt{=}\ e^1 \\ \qquad\qquad \cdots \\ \texttt{def}\ f^n\ x^n_1\ \cdots\ x^n_{a^n}\ \texttt{=}\ e^n \end{array}\right]\!\!\right] = fix(\lambda\phi.\left[\begin{array}{l} f^1 \mapsto \lambda v_1 \ldots v_{a^1}.E\,[\![e^1]\!]^{\phi}_{[x^1_1 \mapsto v_1;\cdots;x^1_{a^1} \mapsto v_{a^1}]} \\ \qquad\qquad \ldots \\ f^n \mapsto \lambda v_1 \ldots v_{a^n}.E\,[\![e^n]\!]^{\phi}_{[x^n_1 \mapsto v_1;\cdots;x^n_{a^n} \mapsto v_{a^n}]} \end{array}\right])$$

$$\begin{array}{lcl} E\ :\ Fenv \to Expr \to Xenv \to Val & & \\ E\,[\![\texttt{var}\ x]\!]^{\phi}_{\rho} & = & \rho(x) \\ E\,[\![\texttt{cons}\ c(e)]\!]^{\phi}_{\rho} & = & \texttt{C}(c, E\,[\![e]\!]^{\phi}_{\rho}) \\ E\,[\![\texttt{oper}\ o\ e_1\ \cdots\ e_n]\!]^{\phi}_{\rho} & = & O\,[\![o]\!]\ E\,[\![e_1]\!]^{\phi}_{\rho}\ \cdots\ E\,[\![e_n]\!]^{\phi}_{\rho} \\ E\,[\![\texttt{call}\ f\ e_1\ \cdots\ e_n]\!]^{\phi}_{\rho} & = & \phi(f)\ E\,[\![e_1]\!]^{\phi}_{\rho}\ \cdots\ E\,[\![e_n]\!]^{\phi}_{\rho} \\ E\,[\![\texttt{let}\ (x_1, e_1), \ldots, (x_n, e_n)\ \texttt{in}\ e]\!]^{\phi}_{\rho} & = & E\,[\![e]\!]^{\phi}_{[x_1 \mapsto E[\![e_1]\!]^{\phi}_{\rho};\cdots;x_n \mapsto E[\![e_n]\!]^{\phi}_{\rho}]\rho} \\ E\left[\!\!\left[\begin{array}{l} \texttt{match}\ e\ \texttt{with} \\ \quad c_1(x_1)\texttt{->}\ e_1 \mid \cdots \mid c_n(x_n)\texttt{->}\ e_n \end{array}\right]\!\!\right]^{\phi}_{\rho} & = & \begin{array}{l} match\ E\,[\![e]\!]^{\phi}_{\rho}\ with \\ \quad \texttt{C}\ (c_k, v) \to E\,[\![e_k]\!]^{\phi}_{[x_k \mapsto v]\rho} \end{array} \\ E\,[\![\pi^n_m(e)]\!]^{\phi}_{\rho} & = & \begin{array}{l} match\ E\,[\![e]\!]^{\phi}_{\rho}\ with \\ \quad \texttt{P}\ (v_1, \ldots, v_n) \to v_m \end{array} \\ E\,[\![(e_1, \ldots, e_n)]\!]^{\phi}_{\rho} & = & \texttt{P}\ (E\,[\![e_1]\!]^{\phi}_{\rho}, \ldots, E\,[\![e_n]\!]^{\phi}_{\rho}) \end{array}$$

2.3 Programmes étiquetés

Pour distinguer une expression, une variable, des autres expressions et variables, nous aurons besoin d'attacher, à chaque expression, un identificateur unique appelé *étiquette*. Une opération d'*étiquetage* permet de transformer un programme CL, P en un programme P^{ℓ} où chaque expression (et sous-expression) porte une étiquette unique.

Soient les classes de symboles suivantes :

$\mathcal{T}$	Symboles de types
$\mathcal{L}$	Symboles d'étiquettes
$\mathcal{F}$	Symboles de fonctions
$\mathcal{X}$	Symboles de variables locales et de paramètres
$\mathcal{C}$	Symboles de constructeurs
$\mathcal{O}$	Symboles de primitives

La grammaire des programmes étiquetés est dérivée directement de la grammaire du langage CL :

p	$\rightarrow$	$d_1, \ldots, d_n$	
d	$\rightarrow$	`def` $f\ x_1\ \cdots\ x_n$ `=` e	$f \in \mathcal{F}, x_k \in \mathcal{X}, arity(f) = n$
	$\mid$	`type` t `=` $c_1(T)$ `|` $\cdots$ `|` $c_n(T)$	$t \in \mathcal{T}, c_k \in \mathcal{C}$
T	$\rightarrow$	t	$t \in \mathcal{T}$
	$\mid$	$T * \cdots * T$	
e	$\rightarrow$	ℓ : `var` x	$x \in \mathcal{X}$
	$\mid$	ℓ : `cons` $c(e)$	$c \in \mathcal{C}$
	$\mid$	ℓ : `oper` $o\ e_1\ \cdots\ e_n$	$o \in \mathcal{O}, n = arity(o)$
	$\mid$	ℓ : `call` $f\ e_1\ \ldots\ e_n$	$f \in \mathcal{F}, n = arity(f)$
	$\mid$	ℓ : `let` $(x_1, e_1), \ldots, (x_n, e_n)$ `in` e	$x_k \in \mathcal{X}$
	$\mid$	ℓ : `match` e `with` $c_1(x_1)$ `->` e_1 `|` $\cdots$ `|` $c_n(x_n)$ `->` e_n	$c_k \in \mathcal{C}, x_k \in \mathcal{X}$
	$\mid$	$\ell : \pi_m^n(e)$	$1 \leq m \leq n$
	$\mid$	$\ell : (e_1, \ldots, e_n)$	$n \geq 0$

Dans la suite de ce travail, nous supposerons que les programmes sont étiquetés et nous écrirons P à la place de P^ℓ.

Nous écrirons $\mathcal{E}$, la classe des expressions étiquetées d'un programme. La fonction $label : \mathcal{E} \rightarrow \mathcal{L}$ permet de retrouver l'étiquette d'une expression étiquetée.

2.4 Interpréteur TP

Notre exemple principal de programme CL porte sur un interpréteur d'un petit langage impératif appelé TP (tiny Pascal) qui est une extension du petit langage utilisé dans [Lau91a]. La différence principale porte sur la possibilité de définir des procédures. La syntaxe d'un programme TP est la suivante :

$Prog$	=	`program` $name$ `;` $Decl_1$ `;` $\cdots Decl_n$ `;` $Command$
$Decl$	=	`procedure` p `;` $Command$
$Command$	=	`begin` $Command_1$ `;` $\cdots$ `;` $Command_n$ `end`
	$\mid$	`if` $Expr$ `then` $Command$ `else` $Command$
	$\mid$	x `:=` $Expr$
	$\mid$	p
	$\mid$	`while` $Expr$ `do` $Command$
	$\mid$	`read` x
	$\mid$	`write`$(Expr)$
$Expr$	=	x
	=	0
	$\mid$	`succ`$(Expr)$
	$\mid$	`pred`$(Expr)$

Un programme TP est composé d'une liste de déclaration de procédures suivie d'une commande. Une commande est un bloc (une suite de commandes entourées des mots-clés `begin` et `end`), une instruction conditionnelle `if...then...else...`, une boucle `while...do...`, une affectation de variable, un appel à une procédure, la lecture d'une variable `read` ou bien l'écriture de la valeur d'une expression `write`. Les variables sont globales. Pour simplifier, nous supposerons que les variables ne peuvent prendre que des valeurs entières (positives) construites avec la constante 0 et la fonction successeur `succ`. La valeur faux est identifiée avec 0 et toutes les autres valeurs correspondent à la valeur vrai. La fonction `pred` calcule le prédécesseur d'un nombre. Une erreur se produit lorsque cette fonction est appliquée à 0. Un programme TP lit un ou plusieurs nombres avec l'instruction `read` et écrit le ou les résultats avec `write`.

2.4.1 Interprétation standard d'un programme TP

L'évaluation d'un programme TP demande de définir la liste des valeurs que va lire ce programme avec l'instruction `read`. Le résultat de cette évaluation est la liste des valeurs écrites avec l'instruction `write`. Le programme CAML suivant interpréteur un programme TP :

```
type Ident == string ;;
type Nat = S of Nat | Z ;;          (* the type of a value *)
type Expr = Zero                    (* the type of an expression *)
          | Var of Ident
          | Succ of Expr
          | Pred of Expr ;;
type Com = Block of Com list        (* the type of a command *)
         | If of Expr * Com * Com
         | While of Expr * Com
         | Let of Ident * Expr
         | Proc of Ident
         | Read of Ident
         | Write of Expr ;;
type Decl == Ident * Com ;;         (* the type of a procedure *)
type Prog == Ident * Decl list * Com ;;
                                    (* the type of a program *)
type Env == (Ident * Nat) list * Nat list * Nat list ;;
                                    (* the type of an environment *)

             (* functions on environments *)
let get_proc_com f decls = assoc f decl ;;
let make_empty_env reads = ([],reads,[]) ;;
let get_read_val (vars,reads,writes) = hd reads
and pop_read_vals (vars,reads,writes) = (vars,tl reads,writes)
and push_write_val v (vars,reads,writes) = (vars,reads,v::writes)
and get_writes (vars,reads,writes) = writes ;;
let update_var v n (vars,reads,writes) = (upd vars,reads,writes)
  where rec upd vars = match vars with
         (v2,n2)::vars -> if v2 = v then (v,n)::vars else (v2,n2)::upd vars
       |  []           -> [(v,n)] ;;
let get_var v (vars,reads,writes) = assoc v vars ;;
```

```
              (* evaluation of expressions *)
let rec run_e expr env = match expr with
         Zero    -> Z
       | Var(v)  -> get_var v env
       | Succ(e) -> S(run_e e env)
       | Pred(e) -> (match run_e e env with S n -> n) ;;

              (* boolean test on an expression *)
let test expr env = match run_e expr env with
         Z -> false
       | S _ -> true ;;

              (* exec: Prog -> Read list -> Write list *)
let exec (name,decls,com) r = get_writes(run_c com (make_empty_env r))
  where rec
      run_c com env = match com with
         Block(coms) -> run_cs coms env
       | If(e,c1,c2) -> if test e env then run_c c1 env
                                      else run_c c2 env
       | While(e,c)  -> if test e env then run_c com (run_c c env)
                                      else env
       | Let(v,e)    -> update_var v (run_e e env) env
       | Proc(p)     -> run_c (get_proc_com p decls) env
       | Read(v)     -> update_var v (get_read_val env) (pop_read_vals env)
       | Write(e)     -> push_write_val (run_e e env) env
  and run_cs coms env = match coms with
         c::coms -> run_cs coms (run_c c env)
       | []      -> env ;;
```

La fonction `exec` prend comme arguments un programme et une liste d'entiers (pour les instructions `read`). Elle retourne une liste d'entiers (fournis par les instructions `write`). La fonction `run_c` évalue une commande dans un environnement donné et retourne un environnement modifié. La fonction `run_cs` évalue une suite de commandes (un bloc). La fonction `run_e` calcule une expression dans l'environnement courant tandis que `test` renvoie la valeur booléenne correspondant à une expression. Les autres fonctions gèrent l'environnement courant et permettent de retrouver la définition associée à une procédure.

2.4.2 Traduction en CL

Le langage reconnu par l'évaluateur partiel étant CL, cet interpréteur doit être traduit dans ce langage. Ce programme CAML n'est pas directement traduisible en CL pour plusieurs raisons. D'une part, les variables et les noms de procédures de TP sont des chaînes de caractères et ce type n'est pas accepté par CL. Nous pouvons remplacer ces identificateurs par des entiers (le type `Nat` par exemple). Un second problème est posé par des définitions de fonctions locales. Ici, il suffit de les redéfinir de manière globale en ajoutant comme paramètre supplémentaire la variable locale `decls` de la fonction `exec`. Cette opération qui transforme des définitions de fonctions locales en fonctions globale est connue sous le nom de lambda-lifting [Joh85]. De plus, lorsque l'interpréteur calcule le prédécesseur de zéro, le programme ML produit une erreur de filtrage. CL ne possède pas de construction équivalente. Nous avons choisi, dans ce cas, de faire entrer l'interpréteur dans une

boucle infinie (une fonction `error` s'appelant elle-même).

Une traduction possible donne le programme CL suivant dont nous n'avons conservé que la partie concernant l'évaluation des commandes :

```
def run_c com decls env = match com with
        Block(a) -> call run_cs a decls env
      | If(a) -> match call test π¹₃(a) env with
                      true(_) -> call run_c π²₃(a) decls env
                    | false(_) -> call run_c π³₃(a) decls env
      | While(a) -> match  test π²₁(a) env with
                      true(_) -> call run_c com decls (call run_c π²₂(a) decls env)
                    | false(_) -> env
      | Let(a) -> call update_var π²₁(a) (call run_e π²₂(a) env) env
      | Proc(p) -> call run_c (call get_proc_com p decls) decls env
      | Read(v) -> call update_var v (call get_read_val env) (call pop_read_vals env)
      | Write e -> call push_write_val (call run_e e env) env
def run_cs coms decls env = match coms with
        Cons(a) -> call run_cs π²₂(a) decls (call run_c π²₁(a) decls env)
      | Nil(_) -> env
def exec prog reads =
        call get_writes(call run_c π³₃(prog) π³₂(prog) (call make_empty_env π²₂(reads)))
```

2.4.3 Interpréteur TP avec continuations

L'interpréteur TP de la section précédente évalue une commande dans un environnement et retourne l'environnement modifié. Nous pouvons utiliser une version dans le style des continuations [Plo75, AJ89]. L'idée consiste à transformer la fonction `run_c` pour qu'elle prenne une liste de commandes au lieu d'une seule commande. Cette liste de commandes représente la liste des commandes qu'il reste à exécuter avant de terminer le programme. Les commandes sont, en quelque sorte, aplaties.

Voici une version de `exec` dérivée d'une sémantique par continuations :

```
let exec (name,decls,com) r = run_cs [com] (make_empty_env r)
  where rec
      run_cs coms env = match coms with
         [] -> get_writes env
       | com::cont -> (match com with
              Block(coms) -> run_cs (coms@cont) env
            | If(e,c1,c2) -> if test e env
                             then run_cs (c1::cont) env
                             else run_cs (c2::cont) env
            | While(e,c)  -> if test e env
                             then run_cs (c::coms)
                             else run_cs cont env
            | Let(v,e)    -> run_cs cont (update_var v (run_e e env) env)
            | Proc(p)     -> run_cs ((get_proc_com p decls)::cont) env
            | Read(v)     -> run_cs cont (update_var v (get_read_val env)
                                                       (pop_read_vals env))
            | Write(e)    -> run_cs cont (push_write_val (run_e e env) env) ;;
```

Ce programme ML donne le programme CL suivant :

```
def run_c com coms decls env = match com with
        Block(a) -> call run_cs (call append a coms) decls env
      | If(a) -> match call test π¹₃(a) env with
                        true(_) -> call run_c π²₃(a) coms decls env
                      | false(_) -> call run_c π³₃(a) coms decls env
      | While(a) -> match  test π²₁(a) env with
                        true(_) -> call run_c a (cons Cons(com,coms)) decls env
                      | false(_) -> call run_cs coms decls env
      | Let(a) -> call run_cs coms decls (call update_var π²₁(a) (call run_e π²₂(a) env) env)
      | Proc(p) -> call run_c (call get_proc_com p decls) coms decls env
      | Read(v) -> call run_cs coms decls (call update_var v (call get_read_val env)
                                                    (call pop_read_vals env))
      | Write e -> call run_cs coms decls (call push_write_val (call run_e e env) env)
def run_cs coms decls env = match coms with
        Cons(a) -> call run_c π²₁(a) π²₂(a) decls env
      | Nil(_) -> call get_writes env
def exec prog reads =
        call run_c π³₃(prog) (cons Nil()) π³₂(prog) (call make_empty_env π²₂(reads))
```

Dans cette version, la liste des commandes qui restent à exécuter pour terminer le programme est explicitement donnée en argument de la fonction run_c. Avec cette version, les environnements ne sont jamais retournés par une fonction (sauf, bien sûr, celles modifiant un environnement).

Chapitre 3

Clonage de fonctions

Ce chapitre présente la notion de *clonage* dans les mécanismes d'évaluation partielle. Bien que ce concept ne soit pas utilisé par les réalisations actuelles, nous pensons qu'elle forme une partie essentielle dans la spécialisation de fonctions. De plus, il sera nécessaire à la définition du mécanisme de spécialisation présenté dans ce travail. En fait, cette notion apparaît implicitement dans les transformations de programmes utilisées par les évaluateurs partiels. Elle nous permettra de définir de manière plus générale la notion d'identification d'objets spécialisés lors de l'évaluation partielle. Nous montrerons que cette identification repose sur l'égalité des parties connues des valeurs des paramètres des objets spécialisés dans les évaluateurs partiels contemporains.

Puis, après avoir introduit la notion de *valeurs alternatives* comme une extension des domaines utilisés par les évaluateur partiels, nous montrerons que ce mécanisme d'identification n'est pas adapté à ce type de valeurs. Ceci nous conduira à rechercher un moyen différent d'identification à travers la notion d'*événement*. Ici, le clonage, par son caractère général, autorise un nouveau mécanisme d'identification qui sera abordé sur quelques exemples simples.

La comparaison d'événements, contrairement à l'égalité de valeurs, définie un concept proche de l'analyse de la consommation des objets utilisés pour *activer* telle expression plutôt que telle autre. Elle ne s'attache pas aux valeurs calculées pour telle variable du programme mais se fonde sur la comparaison entre les ressources qui ont été consommées pour créer une valeur particulière.

La mise en relation entre clonage, valeurs alternatives et événement sera présentée dans le chapitre suivant.

3.1 Le clonage de fonctions

Les mécanismes de transformation de programmes utilisés par les évaluateurs partiels polyvariants [Bul84] reposent en partie sur une opération de *clonage* des fonctions composant un programme. Ce principe est à opposer à l'évaluation partielle monovariante qui produit qu'une seule version spécialisée par fonction initiale. En fait, les transformations effectivement utilisées par ces programmes ne distinguent pas cette opération de clonage. Elle apparaît de manière implicite, composée avec d'autres notions.

Dans un premier temps, le clonage sera définie, puis, nous analyserons les mécanismes de spécialisation existants en se focalisant sur cette unique transformation. Cette analyse pourrait sembler artificielle puisque ce concept n'apparaît pas dans la littérature sur l'évaluation partielle. Toute-

fois, le clonage aborde les problèmes de la spécialisation de programmes sous un angle original et permettra de trouver des solutions originales en liaison avec les notions de *valeurs alternatives* et d'*événement* ce qui justifie cette analyse.

Globalement, le clonage consiste, à partir d'un certain nombre de définitions de fonctions :

- à copier ces définitions,
- à ajouter des index à chaque duplicata,
- à déterminer les branches utiles des structures de contrôle.

Par exemple, considérons la fonction `append` concaténant deux listes et la fonction `f` appelant `append` avec pour premier argument la liste `[2;1]` :

```
let rec append x y = match x with
          h :: t -> h :: append t y
        |   []   -> y ;;
let f z = append [2;1] z ;;
```

La fonction `append` peut être spécialisée par rapport à son premier argument, ce qui produit trois versions de cette fonction :

```
let append0 x y = match x with [] -> y ;;
let append1 x y = match x with h :: t -> h :: append0 t y ;;
let append2 x y = match x with h :: t -> h :: append1 t y ;;
let f z = append2 [2;1] z ;;
```

La fonction `append2` correspond à la spécialisation de `append` lorsque `x` est la liste `[2;1]`. De même, `append1` spécialise `append` lorsque `x` vaut `[1]` et `append0` lorsque `x` est la liste vide. Chaque version est identifiée par un index et, dans le corps de ces fonctions, un appel à la fonction `append` est remplacé par un appel à un de ses clones. De plus, les structures de contrôle ont été spécialisées et seules les branches utiles apparaissent dans l'alternative : pour `append1` et `append2`, `x` correspond à une liste non vide et pour `append0`, `x` est toujours la liste vide.

La spécialisation continue ensuite en appliquant des transformations comme le dépliage de fonctions, le remplacement des expressions constantes par leur valeur, l'élimination des paramètres et des variables devenus inutiles, l'élimination des structures de contrôle ne possédant qu'une seule branche valide... La version finale de la spécialisation de `f` donne le programme suivant :

```
let f y = 2::1::y ;;
```

Dans ce chapitre, nous laisserons de coté ces transformations ultérieures pour nous concentrer sur cette première étape de clonage.

3.2 Définition du clonage de fonctions

Le clonage repose sur deux mécanismes. Le premier consiste à ajouter des index aux définitions de fonctions et aux appels apparaissant dans le corps de ces fonctions. Le second élimine les branches des structures de contrôle qui ne peuvent jamais être activées. Ainsi, à partir d'un programme P, nous obtenons un nouveau programme dont la syntaxe est dérivée du programme initial. Des index ont été ajoutés aux fonctions et les structures de contrôle ont été simplifiées. Nous appellerons *index de spécialisation* les index qui permettent de différencier un clone d'un autre.

Soit $\mathcal{I}$ l'ensemble de ces index. Le programme obtenu par clonage sur la langage CL est décrit par la grammaire suivante :

p	$\rightarrow$	$d_1, \ldots, d_n$	
d	$\rightarrow$	`def` $f^i\ x_1\ \cdots\ x_n$ `=` e	$f \in \mathcal{F}, x_k \in \mathcal{X}, arity(f) = n, i \in \mathcal{I}$
	$\mid$	`type` t `=` $c_1(T)$ `|` $\cdots$ `|` $c_n(T)$	$t \in \mathcal{T}, c_k \in \mathcal{C}$
T	$\rightarrow$	t	$t \in \mathcal{T}$
	$\mid$	$T * \cdots * T$	
e	$\rightarrow$	`var` x	$x \in \mathcal{X}$
	$\mid$	`cons` $c(e)$	$c \in \mathcal{C}$
	$\mid$	`oper` $o\ e_1\ \cdots\ e_n$	$o \in \mathcal{O}, n = arity(o)$
	$\mid$	`call` $f^i\ e_1\ \cdots\ e_n$	$f \in \mathcal{F}, n = arity(f), i \in \mathcal{I}$
	$\mid$	`let` $(x_1, e_1), \ldots, (x_n, e_n)$ `in` e	$x_k \in \mathcal{X}$
	$\mid$	`match` e `with` $c_1(x_1)$`->` e_1 `|` $\cdots$ `|` $c_n(x_n)$`->` e_n	$c_k \in \mathcal{C}, x_k \in \mathcal{X}$
	$\mid$	$\pi^n_m(e)$	$1 \leq m \leq n$
	$\mid$	$(e_1, \ldots, e_n)$	$n \geq 0$

Avec la nouvelle classe :

$\mathcal{I}$ Ensemble d'index

Les structures de contrôle ne comportent qu'une partie des branches de la structure de contrôle initiale, celles qui peuvent correspondre à la valeur testée par cette conditionnelle. Ainsi, dans l'exemple de `append`, les structures de contrôles des clones `append2`, `append1` et `append0` ne comportaient qu'une seule branche.

3.3 Clonage dans les évaluateurs partiels existants

Notre but étant d'isoler l'opération de clonage dans les mécanismes de spécialisation, la présentation des programmes d'évaluation partielle sera incomplète puisque les transformations de programmes utilisés seront restreintes au seul clonage. Nous nous efforcerons de réduire le processus de spécialisation à la production d'un programme cloné à partir d'un programme source. Cela consistera à décrire ce que représentent les index de spécialisation, quelle est la manière de les calculer et comment trouver les branches utiles des structures de contrôle du programme cloné.

3.3.1 Les évaluateurs partiels contemporains

Pour obtenir un programme cloné à partir du programme initial, les évaluateurs partiels issus du projet Mix [JSS85] se basent sur un algorithme en deux phases.

- Une première phase appelée *analyse des temps de liaison* (*binding time analysis* ou *BTA*) [Bon91a, Con90a, NN88, Gom90, Hen91] se propose de déterminer pour chaque variable et chaque expression apparaissant dans le programme initial, si leur valeur peut être calculée lors de la seconde phase d'évaluation partielle. Nous parlerons alors de variables ou d'expressions *statiques*, dans le cas contraire, de variables ou d'expressions *dynamiques*.

  Le résultat de cette première phase est constitué du programme initial auquel sont ajoutées des informations appelées *annotations* et permettant de diriger la seconde phase de l'évaluateur partiel.

– La seconde phase, dite phase de *spécialisation*, consiste à réduire les expressions statiques, à engendrer des versions spécialisées des fonctions et à éliminer les paramètres et les variables statiques (car leur valeur est connue).

  Comme nous nous intéressons uniquement au concept de clonage, nous supposerons que cette phase produit simplement des versions spécialisées d'une fonction (des clones) et associe une valeur à chaque expression statique du programme cloné.

Domaine des valeurs statiques/dynamiques

Les premiers évaluateurs partiels [JSS85] ne pouvaient distinguer qu'entre les variables ou les paramètres dont la valeur est soit calculable, soit complètement inconnue. La phase d'analyse repose sur une interprétation abstraite dont le domaine se compose de deux éléments : S pour statique et D pour dynamique (d'autres auteurs utilisent K pour connu (known) et U pour inconnu (unknown)). Après cette première phase, les variables et les expressions annotées par S sont entièrement calculables. Les autres expressions, annotées par D, ne retournent pas de résultat lors de la phase de spécialisation. Elles sont *résiduelles*.

La phase de spécialisation proprement dite utilise deux mécanismes. Un mécanisme de mémorisation (*memoization*), couplé à une liste des appels en suspens (*a pending list*), gère les appels aux fonctions résiduelles (celles dont le corps est annoté par D). Le second mécanisme calcule les expressions statiques et se comporte globalement comme un interpréteur.

Le mécanisme de mémorisation enregistre les appels de fonctions qui ont déjà été rencontrées. La liste des appels en suspens mémorise les appels que l'évaluateur partiel doit encore spécialiser. Au cours de l'évaluation partielle d'une expression, si un appel à une fonction résiduelle est rencontré, le spécialiseur commence par regarder si cet appel n'a pas déjà été rencontré. Dans ce cas, il remplace l'appel à cette fonction par l'appel à sa version spécialisée. Dans le cas contraire, il ajoute cet appel à la liste des appels en suspens et à la liste des appels déjà rencontrés. Le mécanisme de spécialisation s'arrête lorsque la liste des appels en suspens est vide.

Pour faire le lien entre ce mécanisme de spécialisation et le concept de clonage, il suffit de remarquer que les index de spécialisation sont constitués par la liste des valeurs des arguments statiques. Le clone d'une fonction se distingue des autres clones par la valeur donné aux paramètres statiques de cette fonction. Le mécanisme d'évaluation des expressions reposent sur le domaine suivant :

$$\begin{aligned} \mathcal{V} &= \top \mid \mathtt{S}\ (\mathcal{V}_S) \\ \mathcal{V}_S &= \mathtt{C}\ (C \times \mathcal{V}_S) \mid \mathtt{P}\ (\mathcal{V}_S \times ... \times \mathcal{V}_S) \end{aligned}$$

La valeur associée à une expression, une variable, un paramètre ou une fonction peut être, soit $\top$ signifiant que cette valeur est complètement inconnue et correspondant à la valeur D de la phase d'analyse des temps de liaison, soit une valeur appartenant au domaine fourni par la sémantique dénotationnelle $S(v)$. Les index de spécialisation sont constitués par la liste des valeurs de $\mathcal{V}$ associées aux paramètres d'une fonction. Les structures de contrôle dont la valeur testée est dynamique reste résiduelle (toutes les branches initiales sont conservées), tandis que les structures réductibles ne conservent que la branche correspondant à la valeur testée (qui ne peut pas être $\top$ si la phase d'analyse des temps de liaison est correcte).

Pour comprendre ce mécanisme sur un exemple, reprenons le programme du début de ce chapitre :

```
let rec append x y = match x with
          h :: t -> h :: append t y
        |   []   -> y ;;
let f z = append [2;1] z ;;
```

La phase d'analyse annote chaque expression comme statique ou dynamique. Nous noterons `S:...` une expression statique et `D:...` une expression dynamique. Le programme annoté est le suivant :

```
let rec append S:x D:y = D:match S:x with
          h :: t -> S:h D:(::) D:append S:t D:y
        |   []   -> D:y ;;
let f D:z = D:append S:[2;1] D:z ;;
```

La notation `D:(::)` signifie que le constructeur de liste `::` est résiduel.
Les annotations découlent des faits suivants :

- Le paramètre `z` de `f` est annoté dynamique car `f` est la fonction à spécialiser.
- Le premier argument de l'appel à `append` dans `f` est constant. Il est donc annoté statique. Par contre, le second argument provient de la variable `z` dont la valeur est supposée inconnue. Il est annoté dynamique.
- Cette annotation se généralise aux paramètres de `append`. En effet, le premier paramètre de cette fonction est lié soit au premier argument de l'appel à `append` de `f` (qui est statique), soit au premier argument de l'appel récursif à `append`. Or, ce dernier (la variable `t`) est statique car sa valeur provient du premier paramètre de `append`. Quand au second paramètre, il est naturellement dynamique car l'un des arguments correspondant est dynamique (celui de `f`).
- Le résultat de `append` est annoté dynamique car sa valeur vient du résultat d'une structure de contrôle dont une des branches a pour résultat une valeur dynamique (celle du cas de base). Le résultat de `f` est dynamique car il provient de celui de `append`.

L'analyse des temps de liaison a été étudiée dans le cadre de l'interprétation abstraite [Bon91a, Con90a, NN88] et dans celui de l'inférence de type [Gom90, Hen91].

La méthode basée sur une interprétation abstraite consiste, à partir d'un programme dans lequel les variables et les expressions sont toutes préalablement annotées par `S:.` (sauf les paramètres de la fonction à spécialiser), à généraliser ces annotations (en les transformant en `D:.`) si nécessaire. Ainsi, sur l'exemple ci-dessus :

- Le processus débute avec un programme annoté de cette manière :

```
let rec append S:x S:y = S:match S:x with
          h :: t -> S:h S:(::) S:append S:t S:y
        |   []   -> S:y ;;
let f D:z = S:append S:[2;1] S:z ;;
```

  Seul le paramètre `z` est annoté par `D:`.

- Puis, ce paramètre dynamique se propage dans `f` ce qui rend dynamique le second argument de l'appel à `append`. Cela impose que le second paramètre de cette fonction soit dynamique :

```
let rec append S:x D:y = S:match S:x with
          h :: t -> S:h S:(::) S:append S:t S:y
        |   []   -> S:y ;;
let f D:z = S:append S:[2;1] D:z ;;
```

- Puisque `y` est dynamique, le second argument de l'appel récursif à `append` est dynamique (cela ne change pas l'annotation du second paramètre de `append` car il est déjà dynamique). De même, la branche du cas de base de `append` devient dynamique. Cette seconde condition impose que le résultat de la structure de contrôle de `append` soit dynamique (car une de ses branches à un résultat dynamique). Le programme annoté devient :

```
let rec append S:x D:y = D:match S:x with
          h :: t -> S:h S:(::) S:append S:t D:y
        |   []   -> D:y ;;
let f D:z = S:append S:[2;1] D:z ;;
```

- Maintenant que le résultat de `append` est dynamique, le résultat de `f` doit l'être de même que le constructeur de liste `(::)` et le résultat du cas général de `append`. Nous obtenons le résultat final :

```
let rec append S:x D:y = D:match S:x with
          h :: t -> S:h D:(::) D:append S:t D:y
        |   []   -> D:y ;;
let f D:z = D:append S:[2;1] D:z ;;
```

- Ces annotations sont stables et n'engendrent pas de nouvelle généralisation.

La phase de spécialisation proprement dite va utiliser ce programme annoté pour produire trois versions de `append`. La phase de clonage effectue les opérations suivantes :

- Évaluation de `(f` $\top$`)` : ajout de `(append S([2;1])` $\top$`)` à la liste des appels en suspens. La valeur associée à `(f` $\top$`)` est $\top$.
- Évaluation de `(append S([2;1])` $\top$`)` : ajout de `(append S([1])` $\top$`)` à la liste des appels en suspens. La valeur associée à `(append S([2;1])` $\top$`)` est $\top$.
- Évaluation de `(append S([1])` $\top$`)`: ajout de `(append S([])` $\top$`)` à la liste des appels en suspens. La valeur associée à `(append S([1])` $\top$`)` est $\top$.
- Évaluation de `(append S([])` $\top$`)` : La valeur associée à `(append S([])` $\top$`)` est $\top$.

A ce moment là, la liste des appels en suspens est vide et l'évaluateur partiel s'arrête sur le programme :

```
let rec append_I0 x y = match x with [] -> y
and     append_I1 x y = match x with h :: t -> h :: append_I0 t y
and     append_I2 x y = match x with h :: t -> h :: append_I1 t y ;;
let f z = append_I2 [2;1] z ;;
```

Les index de spécialisation sont :

```
append_I0  :  append S([])  ⊤
append_I1  :  append S([1]) ⊤
append_I2  :  append S([2;1]) ⊤
```

Ainsi, la phase de clonage repose sur une évaluation des expressions avec un domaine enrichi de la valeur $\top$ (par rapport à la sémantique dénotationnelle). Une valeur est soit complètement connue, soit complètement inconnue. Ce mécanisme d'évaluation partielle est complété d'un mécanisme de mémorisation permettant de générer les fonctions résiduelles spécialisées. Les index de spécialisation sont constitués des valeurs associées aux paramètres des clones de fonction.

Structures partiellement connues

Lors de la phase d'analyse des temps de liaison, les évaluateurs partiels reposant sur le mécanisme décrit dans la section précédente ne pouvaient distinguer que deux types d'expressions : d'une part, celles pouvant être entièrement calculées et d'autre part, celles dont une des composantes n'est pas calculable et qui restent résiduelles. Cette méthode, suffisante pour traiter l'exemple de `append` ci-dessus, échoue sur des exemples à peine plus complexes. Par exemple, pour reprendre la spécialisation de `f`, nous aurions pu utiliser une version non-curryfiée de `append`. Dans ce cas, la méthode ne donne pas de bon résultat comme l'illustre le programme suivant.

Le programme correspondant à `f` lorsque `append` prend un couple comme argument est défini par :

```
let rec append p = match fst(p) with
          h :: t -> h :: append(t,snd(p))
        |   []   -> snd(p) ;;
let f z = append([2;1],z) ;;
```

Les fonctions `fst` et `snd` sont les projections sur la première et la seconde composante d'un couple.

La phase d'analyse des temps de liaison donne les annotations suivantes :

```
let rec append D:p = match D:fst(p) with
          h :: t -> D:h D:(::) D:append D:(D:t,D:snd(D:p))
        |   []   -> D:snd(D:p) ;;
let f z = D:append D:(S:[2;1],z) ;;
```

Seule la construction de la liste `[2;1]` est statique. Toutes les autres expressions sont dynamiques. En effet, le paramètre `p` ne peut pas être annoté statique car la seconde composante de cette paire est dynamique; d'où les annotations obtenues.

La phase de clonage ne peut donner qu'un programme équivalent au programme initial car toutes les expressions (sauf la liste constante) sont résiduelles. L'évaluation partielle n'apporte aucun gain d'efficacité.

Pour remédier à ce problème, les évaluateurs partiels plus récents étendent le domaine de l'analyse des temps de liaison en ajoutant des *valeurs partiellement connues* [Mog88]. Ces valeurs ne sont plus complètement calculables lors de la phase de spécialisation mais peuvent comporter des sous-valeurs inconnues. Par exemple, la paire associée à `p` est constituée d'une valeur statique et d'une valeur dynamique. Elle constitue une valeur partiellement connue.

Avec cette méthode, le programme ci-dessus est annoté comme suit :

```
let rec append P:p = match S:fst(p) with
          h :: t -> S:h D:(::) D:append P:(S:t,D:snd(D:p))
        |   []   -> D:snd(D:p) ;;
let f z = D:append P:(S:[2;1],z) ;;
```

Les paires ont été annotées par `P:` signifiant qu'elles sont partiellement connues.

La phase de spécialisation est aussi modifiée pour tenir compte de ces valeurs. Ainsi, le domaine de calcul de l'évaluateur partiel devient :

$$\mathcal{V} \quad = \quad \top \mid \texttt{C}\ (C \times \mathcal{V}) \mid \texttt{P}\ (\mathcal{V} \times \cdots \times \mathcal{V})$$

La valeur $\top$ apparaît comme valeur possible pour une sous-valeur.

Par contre, le mécanisme de mémorisation reste inchangé si ce n'est que les index de spécialisation sont composés des valeurs connues, partiellement connues ou complètement inconnues des paramètres d'un clone.

Sur l'exemple de la fonction `f`, le clonage s'effectue comme suit :

- Évaluation de `(f ⊤)` : ajout de `(append P([2;1],⊤))` à la liste des appels en suspens. La valeur associée à `(f ⊤)` est ⊤.

- Évaluation de `(append P([2;1],⊤))` : ajout de `(append P([1],⊤))` à la liste des appels en suspens. La valeur associée à `(append P([2;1],⊤))` est ⊤.

- Évaluation de `(append P([1],⊤))` : ajout de `(append P([],⊤))` à la liste des appels en suspens. La valeur associée à `(append P([1],⊤))` est ⊤.

- Évaluation de `(append P([],⊤))` : La valeur associée à `(append P([],⊤))` est ⊤.

L'évaluateur partiel s'arrête sur le programme :

```
let rec append_I0 p = match fst(p) with [] -> snd(p)
and     append_I1 p = match fst(p) with h :: t -> h :: append_I0(t,snd(p))
and     append_I2 p = match fst(p) with h :: t -> h :: append_I1(t,snd(p)) ;;
let f z = append_I2([2;1],z) ;;
```

Avec pour index de spécialisation :

```
append_I0  :  append P([],⊤)
append_I1  :  append P([1],⊤)
append_I2  :  append P([2;1],⊤)
```

Spécialisation de constructeurs

Une seconde voie a aussi permis de définir un autre concept de spécialisation. Dans les mécanismes décrits ci-dessus, seules les fonctions sont spécialisées : les fonctions et les appels de fonctions reçoivent un index de spécialisation.

Dans [Mog93], l'auteur étend le mécanisme de clonage en introduisant la notion de spécialisation de constructeurs. Avec le langage CL, cela consiste à ajouter des index de spécialisation aux constructeurs. Ainsi, il est possible de distinguer une valeur construite à un endroit du programme d'une valeur construite dans une autre partie. Pour que cette opération soit correcte, l'évaluateur doit aussi spécialiser les structures de contrôle. En effet, puisque nous distinguons un constructeur créé à un endroit du programme d'une valeur créé à un autre, nous pouvons créer plusieurs versions d'une même branche d'une structure de contrôle correspondant aux différentes versions d'un même constructeur.

Dans ce cadre, la grammaire des programmes clonés est la suivante :

$$
\begin{array}{lcll}
p & \rightarrow & d_1, \ldots, d_n & \\
d & \rightarrow & \texttt{def } f^i\ x_1\ \cdots\ x_n\ \texttt{=}\ e & f \in \mathcal{F}, x_k \in \mathcal{X}, arity(f) = n, i \in \mathcal{I} \\
 & | & \texttt{type } t\ \texttt{=}\ c_1(T)\ \texttt{|}\ \cdots\ \texttt{|}\ c_n(T) & t \in \mathcal{T}, c_k \in \mathcal{C} \\
 & | & \texttt{type } t^i\ \texttt{=}\ c_1^{i_1}(T)\ \texttt{|}\ \cdots\ \texttt{|}\ c_n^{i_n}(T) & t \in \mathcal{T}, c_k \in \mathcal{C}, i \in \mathcal{I}, i_k \in I \\
T & \rightarrow & t & t \in \mathcal{T} \\
 & | & t^i & t \in \mathcal{T}, i \in \mathcal{I} \\
 & | & T * \cdots * T & \\
e & \rightarrow & \texttt{var } x & x \in \mathcal{X} \\
 & | & \texttt{cons } c(e) & c \in \mathcal{C} \\
 & | & \texttt{cons } c^i(e) & c \in \mathcal{C}, i \in \mathcal{I} \\
 & | & \texttt{oper } o\ e_1\ \cdots\ e_n & o \in \mathcal{O}, n = arity(o) \\
 & | & \texttt{call } f^i\ e_1\ \cdots\ e_n & f \in \mathcal{F}, n = arity(f), i \in \mathcal{I} \\
 & | & \texttt{let } (x_1, e_1), \ldots, (x_n, e_n)\ \texttt{in } e & x_k \in \mathcal{X} \\
 & | & \texttt{match } e\ \texttt{with} & \\
 & & \quad c_1(x_1)\ \texttt{->}\ e_1\ \texttt{|}\ \cdots\ \texttt{|}\ c_n(x_n)\ \texttt{->}\ e_n & c_k \in \mathcal{C}, x_k \in \mathcal{X} \\
 & | & \texttt{match } e\ \texttt{with} & \\
 & & \quad c_1^{i_1}(x_1)\ \texttt{->}\ e_1\ \texttt{|}\ \cdots\ \texttt{|}\ c_n^{i_n}(x_n)\ \texttt{->}\ e_n & c_k \in \mathcal{C}, i_k \in \mathcal{I}, x_k \in \mathcal{X} \\
 & | & \pi_m^n(e) & 1 \leq m \leq n \\
 & | & (e_1, \ldots, e_n) & n \geq 0
\end{array}
$$

Comme tous les constructeurs et toutes les structures de contrôle ne sont pas spécialisés de cette manière, nous avons divisé les constructeurs en constructeurs du programme original et clones de constructeur. De même pour les conditionnelles.

Par rapport au programme initial, un index de spécialisation est ajouté aux fonctions, aux appels de fonctions, aux constructeurs et aux constructeurs apparaissant dans les motifs des différentes branches d'une structure de contrôle.

La phase d'analyse des temps de liaison se charge, en plus des annotations précédentes, de déterminer si un constructeur doit être spécialisé (porter un index).

La phase de spécialisation proprement dite est modifiée pour tenir compte des index de spécialisation ajoutés aux constructeurs. Ces index correspondent, comme les index de spécialisation ajoutés aux fonctions, à la valeur de l'argument de ces constructeurs. Le domaine d'évaluation reste le même que celui de l'évaluateur partiel précédent.

Lorsque, au cours de l'évaluation d'une expression, un constructeur est rencontré, le mécanisme de mémorisation vérifie tout d'abord si une version d'un constructeur avec la même valeur argument (connue ou partiellement connue) n'a pas déjà été rencontrée. Dans le cas contraire, le spécialiseur enregistre ce nouveau clone de constructeur. L'évaluateur partiel crée autant de branches qu'il n'y a de versions d'un constructeur dans les structures de contrôle résiduelles.

Ici, aucun mécanisme ne vérifie que ces clones de constructeurs vont effectivement arriver comme valeur testée de ces structures de contrôle et ce mécanisme peut engendrer beaucoup de branches inutiles. De plus, ce programme n'est pas correct lorsque un clone de constructeur apparaît comme résultat de la fonction spécialisée. Toutefois, ce mécanisme de spécialisation de constructeurs permet de résoudre des problèmes où d'autres évaluateurs partiels échouent puisqu'il étend les possibilités de transformation de programmes.

3.3.2 Principes des évaluateurs partiels *hors ligne*

Ainsi, à travers ce bref survol, nous pouvons déduire les principes sur lesquels reposent les programmes de spécialisation *hors ligne*. Ils se décomposent en deux phases. La première dite

analyse des temps de liaison repose sur une interprétation abstraite (ou un système d'inférence) du programme et tente de définir quelle est la partie statique et quelle est la partie dynamique des arguments de chaque objet spécialisable. Une seconde phase se charge de la spécialisation de ces objets en calculant les valeurs de leurs arguments.

Cette seconde phase comporte deux mécanismes. Le premier est un évaluateur dont le domaine est celui de la sémantique dénotationnelle enrichi de la valeur $\top$. Cette valeur spéciale dénote une absence totale d'information lors de la phase de spécialisation. Le second mécanisme mémorise les objets spécialisés déjà rencontrés et enregistre ceux qui restent à spécialiser. L'identification d'un objet spécialisé avec un autre porte sur l'égalité entre leurs index de spécialisation. Ces index sont constitués de la ou des valeurs des paramètres de ces objets.

3.4 Extension du domaine

Nous avons vu que le domaine d'évaluation des évaluateurs partiels est devenu de plus en plus précis. La sémantique dénotationnelle décrit les valeurs de manière complète. Les premiers évaluateurs partiels divisent ce domaine en deux : soit une valeur est complètement définie (statique), soit complètement inconnue (dynamique). Les successeurs de ces premiers évaluateurs permettent de définir des valeurs partiellement connues.

Nous pouvons alors nous demander si nous ne pouvons pas étendre ce domaine à d'autres valeurs. Cela permettrait de décrire les arguments des fonctions de manière plus précise et de spécialiser des fonctions où les évaluateurs partiels du type de ceux décrits ci-dessus échouent.

Une extension très intéressante consisterait à autoriser qu'une valeur puisse décrire plusieurs valeurs différentes. En effet, la cause principale de généralisation des valeurs par les évaluateurs partiels provient des structures de contrôle résiduelles. Lorsque le spécialiseur rencontre une structure de contrôle dont la valeur testée n'est pas connue, la valeur retournée par cette structure est toujours $\top$ indiquant que l'on ne connaît absolument rien sur le résultat. Une version plus précise pourrait retourner la partie commune des valeurs retournées par les différentes branches. Toutefois, la majorité des évaluateurs partiels existants retourne, dans ce cas, la valeur $\top$ (à l'exception de [Lau91a, Bec94]). Une autre source de généralisation vient des fonctions dont le résultat n'est pas entièrement connu et qui ne sont pas dépliées. Tous les évaluateurs partiels existants généralisent le résultat à $\top$ (à l'exception des deux cités ci-dessus). Cela résulte de la manière dont la phase de spécialisation évalue les fonctions. Supposons en effet, qu'une fonction résiduelle ait été rencontrée. Lors de l'évaluation partielle de son corps, le spécialiseur peut rencontrer un appel à la même fonction avec les mêmes arguments. Or, comme la valeur retournée par cette fonction n'est pas encore connue puisqu'elle est en train d'être calculée, le spécialiseur ne peut pas retourner la valeur associée à ce clone et doit supposer que cette valeur est $\top$.

Dans ce travail, nous voulons résoudre ces deux problèmes. D'une part, nous voulons étendre le domaine d'évaluation en autorisant qu'une valeur puisse prendre plusieurs valeurs possibles comme résultat d'une structure de contrôle résiduelle. D'autre part, nous désirons résoudre le problème de la généralisation du résultat d'une fonction résiduelle pour lui associer une valeur plus précise que la valeur $\top$.

3.5 Évaluation partielle d'un interpréteur TP

Pour cerner les motivations de ce travail, nous allons présenter un exemple classique [Lau91a] dans la littérature à propos de l'évaluation partielle qui nous permettra de comprendre l'insuffisance des évaluateurs partiels existants.

Considérons TP, le petit langage impératif à la Pascal décrit dans le chapitre précédent dont nous rappelons la syntaxe :

$$
\begin{array}{lcl}
Prog & = & \texttt{program}\ name\ \texttt{;}\ Decl_1\ \texttt{;}\cdots Decl_n\ \texttt{;}\ Command \\
Decl & = & \texttt{procedure}\ p\ \texttt{;}\ Command \\
Command & = & \texttt{begin}\ Command_1\ \texttt{;}\cdots\texttt{;}\ Command_n\ \texttt{end} \\
 & | & \texttt{if}\ Expr\ \texttt{then}\ Command\ \texttt{else}\ Command \\
 & | & x\ \texttt{:=}\ Expr \\
 & | & p \\
 & | & \texttt{while}\ Expr\ \texttt{do}\ Command \\
 & | & \texttt{read}\ x \\
 & | & \texttt{write}(Expr) \\
Expr & = & x \\
 & = & 0 \\
 & | & \texttt{succ}(Expr) \\
 & | & \texttt{pred}(Expr)
\end{array}
$$

3.5.1 Interprétation d'un programme TP

L'évaluation d'un programme TP demande de définir la liste des valeurs que va lire ce programme avec l'instruction `read`. Le résultat de cette évaluation est la liste des valeurs écrites avec l'instruction `write`.
Le programme CL suivant permet d'interpréter un programme TP :

```
def run_c com decls env = match com with
        Block(a) -> call run_cs a decls env
      | If(a) -> match call test π¹₃(a) env with
                      true(_) -> call run_c π²₃(a) decls env
                    | false(_) -> call run_c π³₃(a) decls env
      | While(a) -> match  test π²₁(a) env with
                      true(_) -> call run_c com decls (call run_c π²₂(a) decls env)
                    | false(_) -> env
      | Let(a) -> call update_var π²₁(a) (call run_e π²₂(a) env) env
      | Proc(p) -> call run_c (call get_proc_com p decls) decls env
      | Read(v) -> call update_var v (call get_read_val env) (call pop_read_vals env)
      | Write e -> call push_write_val (call run_e e env) env
def run_cs coms decls env = match coms with
        Cons(a) -> call run_cs π²₂(a) decls (call run_c π²₁(a) decls env)
      | Nil(_) -> env
def exec prog reads =
        call get_writes(call run_c π³₃(prog) π³₂(p) (call make_empty_env π²₂(reads)))
```

La fonction *exec* prend comme arguments un programme et une liste d'entiers (pour les instructions *read*). Elle retourne une liste d'entiers (fournis par les instructions *write*). La fonction *run_c* évalue une commande dans un environnement donné et retourne un environnement modifié. La fonction *run_cs* évalue une suite de commande (un bloc). La fonction *run_e* calcule une expression dans l'environnement courant tandis que *test* renvoie la valeur booléenne correspond à une expression. Les autres fonctions gèrent l'environnement courant et permettent de retrouver la définition associée à une procédure. Nous n'avons repris ici que les trois fonctions *exec*, *run_cs* et *run_c*.

3.5.2 Spécialisation de l'interpréteur TP

La fonction *exec* prend pour argument un programme TP et une liste de nombres. Elle exécute ce programme puis renvoie la liste des entiers écrits avec l'instruction *while*. Lorsque le programme est connu mais que la liste des entiers qui seront lus ne l'est pas, nous pouvons lancer un programme évaluant partiellement cet interpréteur avec pour argument statique le programme TP. Considérons, par exemple, le programme TP suivant qui calcule la somme de deux entiers :

```
program ADD;          (* R1+R2 -> W *)
 procedure add;       (* x+y -> z *)
   begin
     z:=y;
     while x do
       begin
         x:=pred(x);
         z:=succ(z)
       end
    end
 begin
   read x;            (* R1 -> x *)
   read y;            (* R2 -> y *)
   add;
   write z            (* z -> W *)
 end
```

Nous pouvons attendre qu'une spécialisation de l'interpréteur avec ce programme d'addition, réalise une véritable compilation du programme TP dans le langage de l'évaluateur partiel (ici CL). Un bon résultat serait le programme (traduit en CAML) suivant :

```
type Nat = S of Nat | Z ;;

let rec run_while x z = match x with
      S(x2) -> run_while x2 S(z)
    | N     -> z ;;

let run_add x y = [run_while x y] ;;

let exec_ADD reads = run_add (hd(reads)) (hd(tl(reads))) ;;
```

La fonction `exec_ADD` correspond à la fonction initiale `exec` avec pour premier argument le programme d'addition. La fonction `run_add` correspond à l'exécution de la procédure TP `add` et la fonction `run_while` à la boucle `while` du programme d'addition.

Nous voyons que ce programme ne fait plus référence au programme TP qui a complètement disparu. De plus, les environnements ont aussi disparu.

3.5.3 Spécialisation avec les évaluateurs partiels hors ligne

Domaine Statique/Dynamique

La spécialisation de ce programme avec les évaluateurs partiels décrit au début de ce chapitre donnent des résultats moyens. Ils réussissent à éliminer le programme TP du programme résiduel

mais n'arrivent pas à éliminer les environnements. En effet, si nous regardons les variables liées à des parties du programme TP d'addition, nous nous apercevons qu'elles sont annotées comme statiques. Par contre, comme le programme final doit comporter au moins une boucle (correspondant bien sûr à la boucle `while`), le programme doit comporter au moins une fonction résiduelle. Or cette fonction ne peut être qu'une des fonction `run_c` ou `run_cs`. Par conséquent, le résultat de cette fonction est généralisée comme nous l'avons signalé plus haut. Ainsi, puisque ces fonctions renvoient un environnement, ces structures sont forcément généralisées et aucun renseignement ne peut en être tiré. Ainsi, toute la partie concernant les environnements n'est pas spécialisée et les appels aux fonctions les gérant restent résiduels (ce n'est pas tout à fait exact car elles peuvent être spécialisées par rapport aux arguments provenant du programme — par exemple, la fonction `update` est spécialisée par rapport au nom de la variable à modifier).

Voici le programme annoté par la phase d'analyse des temps de liaison :

```
def run_c S:com S:decls D:env = D:match S:com with
        Block(a) -> D:call run_cs S:a S:decls D:env
      | If(a) -> D:match D:call test S:π¹₃(a) D:env with
                          true(_) -> D:call run_c S:π²₃(a) S:decls D:env
                        | false(_) -> D:call run_c S:π³₃(a) S:decls D:env
      | While(a) -> D:match  test S:π¹₂(a) D:env with
                          true(_) -> D:call run_c S:com S:decls (D:call run_c S:π²₂(a) S:decls D:env)
                        | false(_) -> D:env
      | Let(a) -> D:call update_var S:π¹₂(a) (D:call run_e S:π²₂(a) D:env) D:env
      | Proc(p) -> D:call run_c (S:call get_proc_com S:p S:decls) S:decls D:env
      | Read(v) -> D:call update_var S:v (D:call get_read_val D:env) (D:call pop_read_vals D:env)
      | Write e -> D:call push_write_val (D:call run_e S:e D:env) D:env
def run_cs S:coms S:decls D:env = D:match S:coms with
        Cons(a) -> D:call run_cs S:π²₂(a) S:decls (D:call run_c S:π¹₂(a) S:decls D:env)
      | Nil(_) -> D:env
def exec S:prog D:reads =
        D:call get_writes(D:call run_c S:π³₃(prog) S:π³₂(prog) (D:call make_empty_env D:π²₂(reads)))
```

Les variables et les expressions annotées comme statiques sont celles référant le programme TP. Toutes les opérations sur les environnements sont dynamiques.

Avec les structures partiellement connues

Avec l'ajout des structures partiellement connues, un évaluateur partiel ne peut pas éliminer les environnements car ceux-ci sont donnés comme résultat d'une fonction résiduelle et nous avons vu que les évaluateurs partiels généralisent le résultat de ces fonctions.

Par conséquent, l'interpréteur TP est annoté comme dans le cas précédent. Nous obtenons le même programme résiduel.

Spécialisation de constructeurs

La spécialisation de constructeurs n'apporte que peu d'améliorations en spécialisant les éléments des listes d'association portant les valeurs des variables par rapport au nom de la variable. Cela donne ici trois nouveaux constructeurs correspondants aux structures :

Cons(("x",⊤),⊤)
Cons(("y",⊤),⊤)
Cons(("z",⊤),⊤)

Le programme résiduel obtenu ressemble au programme suivant (traduit en CAML) :

```
let run_while env = match get_val_x env with
       true -> ( let env1 = update_var_x (pred(get_var_x env)) env
                 in run_while (update_var_z (succ(get_var_z env)) env) )
       false -> env ;;

let run_add env = run_while (update_var_z (get_val_y)) ;;

let run_cs1 env =
  let env1 = update_var_x (get_read_val env) (pop_read_vals env)
   in let env2 = update_var_y (get_read_val env1) (pop_read_vals env1)
       in push_write_val (get_var_z run_add env2) ;;

let exec_ADD reads = get_writes(run_cs1 (make_empty_env reads) ;;
```

Les définitions des fonctions sur les environnements qui sont assez similaires aux fonctions d'origine n'ont pas été représentés ici. De même, les fonctions `succ` et `pred` ont été ajoutées pour améliorer la lisibilité du programme.

La gestion des environnements reste résiduelle.

Passage au style par continuations

Pour que les environnements créés par l'interpréteur puissent être correctement traités, nous devons transformer l'interpréteur dans un style avec continuation. En effet, nous devons arriver à ce que les environnements ne soient plus résultats de fonctions résiduelles mais qu'ils soient toujours donnés en argument aux fonctions d'évaluation des commandes. L'idée consiste à transformer la fonction `run_c` pour qu'elle prenne une liste de commandes au lieu d'une seule commande. Cette liste de commandes représente la liste des commandes qu'il reste à exécuter avant de terminer le programme. Les commandes sont, en quelque sorte, aplaties.

Voici une version de l'interpréteur dérivée d'une sémantique par continuations :

```
def run_c com coms decls env = match com with
        Block(a) -> call run_cs (call append a coms) decls env
      | If(a) -> match call test π¹₃(a) env with
                    true(_) -> call run_c π²₃(a) coms decls env
                  | false(_) -> call run_c π³₃(a) coms decls env
      | While(a) -> match  test π²₁(a) env with
                    true(_) -> call run_c a (cons Cons(com,coms)) decls env
                  | false(_) -> call run_cs coms decls env
      | Let(a) -> call run_cs coms decls (call update_var π²₁(a) (call run_e π²₂(a) env) env)
      | Proc(p) -> call run_c (call get_proc_com p decls) coms decls env
      | Read(v) -> call run_cs coms decls (call update_var v (call get_read_val env)
                                                  (call pop_read_vals env))
      | Write e -> call run_cs coms decls (call push_write_val (call run_e e env) env)
def run_cs coms decls env = match coms with
        Cons(a) -> call run_c π²₁(a) π²₂(a) decls env
      | Nil(_) -> call get_writes env
def exec prog reads =
        call run_c π³₃(prog) (cons Nil()) π³₂(prog) (call make_empty_env π²₂(reads))
```

Cette version se spécialise bien sur l'exemple du programme TP en utilisant un évaluateur partiel avec structures partiellement connues. En effet, les variables dont la valeur est un environnement

sont annotées comme partiellement connues. La phase d'analyse des temps de liaison fournit le programme annoté suivant :

```
def run_c S:com S:coms S:decls P:env = D:match S:com with
        Block(a) -> D:call run_cs (S:call append S:a S:coms) S:decls P:env
      | If(a) -> D:match D:call test S:π¹₃(a) P:env with
                          true(_) -> D:call run_c S:π³₂(a) S:coms S:decls P:env
                        | false(_) -> D:call run_c S:π³₃(a) S:coms S:decls P:env
      | While(a) -> D:match  test S:π²₁(a) P:env with
                          true(_) -> D:call run_c S:a (S:cons Cons(S:com,S:coms)) S:decls P:env
                        | false(_) -> D:call run_cs S:coms S:decls P:env
      | Let(a) -> D:call run_cs S:coms S:decls (P:call update_var S:π²₁(a) (D:call run_e S:π²₂(a) P:env) P:env)
      | Proc(p) -> D:call run_c (S:call get_proc_com S:p S:decls) S:coms S:decls P:env
      | Read(v) -> D:call run_cs S:coms S:decls (P:call update_var S:v (D:call get_read_val P:env)
                                                  (P:call pop_read_vals P:env))
      | Write e -> D:call run_cs S:coms S:decls (P:call push_write_val (D:call run_e S:e P:env) P:env)
def run_cs S:coms S:decls P:env = D:match S:coms with
        Cons(a) -> D:call run_c S:π²₁(a) S:π²₂(a) S:decls P:env
      | Nil(_) -> D:call get_writes P:env
def exec S:prog D:reads =
        D:call run_c S:π³₃(prog) (S:cons Nil()) S:π³₂(prog) (P:call make_empty_env D:π²₂(reads))
```

La phase de spécialisation créera autant de versions spécialisées des fonctions *run_c* et *run_cs* qu'il y a de points du programme d'addition (un bon nombre sera déplié). Les environnements deviendront de simple produit de valeurs (une pour *read*, une pour *writes* et autant de valeur que le programme ne comporte de variables (x, y et z)). Le programme résiduel ressemblera au programme attendu (traduit en CAML) :

```
type Nat = S of Nat | Z ;;

let rec run_while x y z reads writes = match x with
      S(x2) -> run_while x2 y (S z) reads writes
    | N     -> z :: writes ;;

let run_add x y reads writes = run_while x y y reads writes ;;

let exec_ADD reads = run_add (hd(reads)) (hd(tl(reads))) (tl(tl(reads))) [] ;;
```

Ainsi, mis à part le passage d'arguments inutiles (`z`, `reads` et `write` de `run_while`), ce programme est satisfaisant puisqu'il spécialise correctement les environnements. La version de l'interpréteur dans le style des continuations se spécialise bien.

Problème de terminaison

Toutefois, cet interpréteur n'est pas satisfaisant pour tous les programmes TP. En effet, comme un programme TP peut appeler récursivement une procédure, la variable *coms* des fonctions *run_c* et *run_cs* qui est liée aux commandes restant à exécuter, n'est pas forcément une valeur bornée. Par exemple, nous pouvons réécrire le programme d'addition comme suit, en remplaçant la boucle `while` par un appel récursif :

```
program ADD;          (* R1+R2 -> W *)
procedure add;        (* x+y -> y *)
```

```
    begin
      if x
      then
        begin
          x:=pred(x);
          add;
          y:=succ(y)
        end
      else
        begin
        end
    end
  begin
    read x;              (* R1 -> x *)
    read y;              (* R2 -> y *)
    add;
    write y              (* y -> W *)
  end
```

Lors de l'exécution de ce programme, la profondeur maximale des appels à `add` correspond au nombre `x` qui n'est pas connu lors de l'évaluation partielle. Par conséquent, avec l'interpréteur dans le style des continuations, la variable `coms` peut contenir une liste non bornée de commandes `y:=succ(y)`. Ainsi, comme la variable `coms` est annotée comme statique, l'évaluateur partiel boucle en essayant de spécialiser la fonction `run_c` avec une infinité de listes de commandes différentes qui correspondent à toutes les continuations possibles à cet endroit.

La solution consistant à utiliser un interpréteur dans le style des continuations est correcte lorsque nous n'autorisons pas la définition de procédure, comme dans [Lau91a]. En effet, pour un programme TP donné, l'ensemble des continuations possibles est fini et l'évaluateur partiel termine toujours.

Ainsi, les évaluateurs partiels basés sur les principes exposés ci-dessus ne peuvent traiter de façon complètement satisfaisante la spécialisation de l'interpréteur pour le langage TP lorsque, au moment de la spécialisation, le programme interprété est connu mais pas les valeurs lues par les instructions `read`. Soit ils n'arrivent pas à traiter correctement les environnements, soit, en utilisant une version basée sur les continuations, la spécialisation peut ne pas terminer.

3.5.4 Échec de cette approche

La raison principale de cet échec provient du mécanisme de généralisation du résultat d'une structure de contrôle résiduelle et du résultat d'une fonction résiduelle. En fait, dans Similix [BJ93] ces deux cas se rejoignent car cet évaluateur partiel utilise comme point de spécialisation ces structures de contrôle résiduelles. Les fonctions résiduelles coïncident avec les structures de contrôle résiduelles. Les fonctions sont toujours dépliées.

Par conséquent, la mauvaise gestion des environnements dans l'interpréteur résulte de la généralisation des valeurs des différentes branches d'une structure de contrôle dont la valeur testée est dynamique. Pour le programme TP d'addition, ces points correspondent au test de la valeur de la variable `x` dans la boucle `while`. Comme cette structure retourne un environnement (éventuellement différent pour les deux branches), les évaluateurs partiels doivent généraliser le résultat de cette conditionnelle. En effet, le domaine d'évaluation des spécialiseurs ne peut pas décrire des valeurs pouvant prendre plusieurs valeurs. Puisque l'environnement retourné peut provenir de n'importe

quelle branche de la structure de contrôle, on ne peut pas le décrire dans le domaine utilisé. Une solution partielle consisterait à retourner la partie commune des différentes branches comme il est proposé dans [Bec94]. Toutefois, ce système généralise certaines informations sur le résultat d'une structure de contrôle résiduelle car seule la partie commune est retenue.

3.6 Extensions du domaine

Nous avons vu que le domaine d'évaluation est passé du domaine associé à la sémantique dénotationnelle et correspondant à une évaluation *complète* d'un programme, à un domaine enrichi de la valeur $\top$ pour représenter les valeurs inconnues lors de la spécialisation, pour déboucher finalement sur le domaine des valeurs partiellement connues.

Nous pourrions envisager d'étendre le domaine d'évaluation avec des valeurs représentant les différentes valeurs possibles des branches d'une conditionnelle.

En fait, un des problèmes particulièrement difficiles à résoudre concerne le choix entre un évaluateur partiel puissant mais pouvant boucler dans de nombreux cas et un évaluateur moins performant mais qui termine plus souvent. Ce dilemme s'était posé dans l'exemple de la spécialisation de l'interpréteur TP où nous avions le choix entre obtenir un résultat moyen avec un mécanisme qui termine toujours (spécialisation de l'interpréteur normal) ou un bon résultat avec un processus qui peut ne pas terminer (spécialisation de l'interpréteur dans le style des continuations). Dans le même ordre d'idée, les évaluateurs partiels utilisant des valeurs partiellement connues ont tendance à boucler lorsque ces structures grandissent indéfiniment au cours du processus de spécialisation. Bien sûr, un évaluateur partiel n'utilisant pas ces valeurs pourra terminer là où celui-là ne termine pas.

Par conséquent, nous devons nous attendre à ce qu'en augmentant le domaine de l'évaluateur partiel, ce dernier termine encore moins souvent. Malheureusement, cela arrive presque toujours si nous ajoutons sans précaution ce type de valeur. La terminaison du processus de spécialisation déjà critique dans certains cas devient complètement inacceptable ici.

Voici un exemple très simple. Considérons la fonction `append` :

```
let rec append x y = match x with
          h::l -> h::append l y
        |  []  -> y ;;
```

Essayons de la spécialiser en supposant que `x` est inconnu mais que `y` vaut `[1]`. Le test porte sur la valeur de `x` et la structure de contrôle est résiduelle. Le programme résiduel devient :

```
let rec append_bis x = match x with
          h::l -> h::append_bis l
        | []   -> [1] ;;
```

Toutefois, le résultat de cette fonction peut être utilisé par d'autres fonctions du programme. Par conséquent, nous devons calculer une approximation de son résultat. Dans les évaluateurs partiels présentés précédemment, sa valeur serait $\top$,qui est la seule valeur pouvant décrire le résultat de `append_bis` lorsque le domaine ne contient que des valeurs partiellement connues. Par contre, si nous ajoutons la possibilité de décrire une valeur par les différentes valeurs associées aux branches de structures de contrôle résiduelles, nous découvrons que le résultat de `append_bis` peut être :

- $[1]$: lorsque `x` est vide.
- $[\top; 1]$: lorsque `x` comporte un élément.

– $[\top; \top; 1]$: lorsque `x` comporte deux éléments.
...

La valeur devrait pouvoir représenter l'ensemble des listes se terminant par l'élément 1. Le calcul de cet ensemble demanderait une infinité d'étapes et l'évaluateur partiel bouclerait.

Ainsi, même sur cet exemple très simple, l'évaluation partielle ne terminerait pas.

Par contre, si nous analysons l'exemple ci-dessus, nous nous apercevons que le résultat de `append_bis` est soit la liste `[1]` reçue en argument, soit une liste construite avec le constructeur `::` apparaissant dans le corps de cette fonction, dont l'élément de tête est $\top$ et la suite, le résultat de `append_bis`. Par conséquent, la valeur associée au résultat de cette fonction peut se décrire comme une alternative entre la liste `[1]` et une liste construite avec le constructeur de liste de `append_bis`. L'argument de ce constructeur est une paire dont le premier élément est $\top$ et le second la même alternative que précédemment :

$$\begin{array}{lcl} Ret_{\texttt{append\_bis}} & = & Cons_1 | Cons_2 \\ Arg_{Cons_1} & = & (1, Nil_1) \\ Arg_{Cons_2} & = & (\top, Cons_1 | Cons_2) \end{array}$$

Ici, $Cons_1$ et Nil_1 sont les constructeurs qui créent la liste `[1]` et $Cons_2$ est le constructeur `::` apparaissant dans la fonction `append_bis`. Les valeurs décrites par cette représentation est une liste de longueur inconnue mais se terminant par la liste `[1]`.

3.6.1 Les valeurs alternatives

Cet exemple permet de définir le domaine de l'évaluateur partiel dont nous recherchons la définition. Au lieu de définir le domaine comme une simple structure de valeurs partiellement connues, nous allons définir le domaine des valeurs par la donnée d'une valeur et d'un ensemble d'équations définissant la valeur associée à l'argument des constructeurs.

Ainsi, la valeur de retour de `append_bis` est l'alternative entre deux constructeurs de liste $Cons_1$ et $Cons_2$. Les valeurs de leur argument sont des couples définis par les équations ci-dessus.

Pour définir ces équations, nous devons pouvoir identifier chaque constructeur apparaissant dans le programme en cours de spécialisation. Comme dans la partie sur les évaluateurs partiels existants, nous allons donner un index unique à chaque constructeur. Soit $\mathcal{I}_\mathcal{C}$ l'ensemble de ces index. Le clone d'un constructeur est un couple (c, i) noté c^i avec $c \in \mathcal{C}$ et $\in \mathcal{I}$. L'ensemble des clones sera noté $\mathcal{C}^\mathcal{I}$

Le domaine $\mathcal{V}$ des valeurs est définie par l'équation :

$$\begin{array}{rcll} \mathcal{V} & = & \top & \\ & | & \bot & \\ & | & P\ (\mathcal{V} \times ... \times \mathcal{V}) & \\ & | & S\ (C) & C \subset \mathcal{C}^\mathcal{I} \end{array}$$

La valeur $\top$ signifie qu'aucune information n'est connue sur cette valeur. $\bot$ signifie que nous ne connaissons encore rien sur cette valeur (elle représente une absence d'information qui devra être comblée ultérieurement). Une valeur peut être un produit de valeurs. Enfin, elle peut être l'alternative entre un ensemble de clones de constructeurs. La fonction $CArg : \mathcal{I}_\mathcal{C} \rightarrow V$ permet de trouver la valeur associée à un constructeur particulier (en utilisant son index).

Par rapport au domaine des valeurs partiellement connues, ce domaine permet de décrire, d'une part, des valeurs provenant de plusieurs branches de structures de contrôle et, d'autre part, des valeurs cycliques.

3.6.2 Valeurs alternatives et mécanisme de clonage

Une fois défini le domaine avec lequel nous aimerions que notre évaluateur partiel travaille, il nous reste à trouver un mécanisme d'évaluation partielle adapté aux valeurs alternatives. Cette tâche est plus complexe qu'il n'y parait.

Une première idée consisterait à étendre les évaluateurs partiels décrits précédemment pour tenir compte de ces nouvelles valeurs. Malheureusement, cette tâche est difficile. En effet, le mécanisme de mémorisation de ces évaluateurs partiels n'est pas adapté aux valeurs alternatives. Ce problème porte sur la reconnaissance de l'égalité entre les versions d'une même fonction. Nous avons vu que les évaluateurs partiels sont basés sur un mécanisme de mémorisation des appels déjà rencontrés et sur la comparaison entre ceux-ci et les nouveaux appels. Cette comparaison porte sur l'égalité, dans le domaine de l'évaluateur partiel, des valeurs associées aux deux listes de paramètres de ces appels.

Mais, avec le domaine des valeurs alternatives, cette égalité entre valeurs ne donne pas de résultats satisfaisants. D'une part, nous devons choisir entre l'égalité structurelle ou bien l'égalité physique. Pour que deux valeurs soient égales, doit-on regarder récursivement les valeurs et les arguments des constructeurs ? Dans ce cas, ce test est très coûteux car les valeurs forment des structures cycliques. De plus, la structure de ces valeurs reposent plus sur un treillis que sur une notion d'isomorphisme. Ainsi, nous pouvons déduire une relation d'ordre similaire à la notion d'inclusion d'ensembles (les alternatives sont des ensembles de clones de constructeurs) avec une borne inférieure $\sqcap$ et une borne supérieure $\sqcup$ pour tout ensemble fini de valeurs :

$$
\begin{array}{lcl}
\top \sqcup v & = & \top \\
\bot \sqcup v & = & v \\
P(x_1, \ldots, x_n) \sqcup P(y_1, \ldots, y_n) & = & P(x_1 \sqcup y_1, \ldots, x_n \sqcup y_n) \\
S(C_1) \sqcup S(C_2) & = & S(C_1 \cup C_2)
\end{array}
$$

$$
\begin{array}{lcl}
\top \sqcap v & = & v \\
\bot \sqcap v & = & \bot \\
P(x_1, \ldots, x_n) \sqcap P(y_1, \ldots, y_n) & = & P(x_1 \sqcap y_1, \ldots, x_n \sqcap y_n) \\
S(C_1) \sqcap S(C_2) & = & S(C_1 \cap C_2)
\end{array}
$$

Les autres configurations (deux produits dont le nombre d'arguments ne correspond pas ou un produit et une alternative) ne devraient pas se rencontrer si le programme est bien typé.

Bien sûr, l'élément maximal de $\mathcal{V}$ est $\top$ et l'élément minimal $\bot$.

La relation d'ordre $\sqsubset$ se définit par :

$$
\begin{array}{ll}
\bot \sqsubset v & \forall v \in \mathcal{V} \\
v \sqsubset \top & \forall v \in \mathcal{V} \\
P(x_1, \ldots, x_n) \sqsubset P(y_1, \ldots, y_n) & si\ x_1 \sqsubset y_1, \ldots, x_n \sqsubset y_n \\
S(C_1) \sqsubset S(C_2) & si\ C_1 \subset C_2
\end{array}
$$

Dans ce cadre, si nous avons déjà spécialisé une fonction avec la valeur $v_1 \in \mathcal{V}$ puis que nous rencontrions un appel à la même fonction mais avec pour argument la valeur v_2 telle que $v_2 \sqsubset v_1$, nous pourrions identifier les deux appels à cette fonction puisque le premier clone peut être utilisé pour ce second appel (les valeurs du second sont incluses dans les valeurs du premier). Malgré tout, ce mécanisme d'identification n'est pas une relation d'équivalence et serait difficile à mettre en œuvre. De plus, il repose sur la différence des clones de constructeurs. Or, rien ne permet de conclure que leur nombre au cours de la spécialisation est borné et cela peut conduire à la non-terminaison du processus d'évaluation partielle.

3.6.3 Autres mécanismes d'identification

Les remarques brièvement introduites nous ont poussé à chercher un mécanisme différent de l'égalité entre valeurs, nous permettant d'identifier les clones de fonction.

Le problème semble simple à poser. Nous devons trouver un moyen satisfaisant pour identifier deux appels à la même fonction lorsque le spécialiseur juge qu'ils sont similaires. Nous devons aussi pouvoir identifier deux clones d'un même constructeur.

Nous avons affirmé que la comparaison entre les valeurs des arguments de ces clones est difficile à mettre en œuvre, pour des raisons théoriques et pour des raisons pratiques. Par conséquent, nous allons baser notre mécanisme d'identification sur la notion d'*événement* qui ont concouru à *activer* un appel particulier ou à créer un clone de constructeur particulier.

3.7 Notion d'événement

Dans ce travail, nous proposons un autre type d'identification des clones qui nous semble propice à résoudre le problème des valeurs alternatives. Le domaine de comparaison ne va pas être constitué de valeurs attachées aux arguments d'une fonction ou d'un constructeur, mais de structures représentant les *événements* qui ont eu lieu pour activer telle fonction ou tel opérateur. Alors que les mécanismes d'évaluation partielle décrits ci-dessus reposent sur l'égalité entre ce qui reste à traiter (la valeur des arguments), nous proposons de travailler sur ce qui a déjà été utilisé des données.

Pour mettre en œuvre ce concept, nous devons tout d'abord définir ce qu'est un événement. Puisque nous nous intéressons au clonage de fonctions et que l'intérêt principal est de découvrir les branches inutiles des structures de contrôle, les événements vont provenir des tests effectués par le programme. Les événements seront constitués par un couple composé d'un constructeur et de la structure de contrôle qui a reçu cette valeur.

L'identification des appels à une fonction consistera à comparer les événements qui ont activé ces appels (qui sont nécessaires pour exécuter un appel). La section suivante tente de préciser cette notion d'événement et d'identification de clones sur quelques exemples simples.

3.7.1 Exemples

Considérons de nouveau la spécialisation de la fonction `append` lorsque seul le premier argument est connu :

```
let rec append x y = match x with
     h::t -> h::append t y
   |  []  -> y ;;
```

Cette fonction parcourt la liste `x` en la dupliquant et finit par ajouter la liste `y` à la fin de cette nouvelle liste. Si cette fonction reçoit pour premier argument une liste dont la longueur est connue, un évaluateur partiel devrait pouvoir créé autant de versions spécialisées de la fonction `append` que cette liste ne comporte d'éléments (plus un pour la liste vide). Par exemple, supposons que cette liste comporte deux éléments `[e1;e2]`. Nous obtenons le programme cloné suivant, comme nous l'avions vu au début de ce chapitre :

```
let append_0 x y = match x with [] -> y ;;
let append_1 x y = match x with h::t -> h::append_0 t y ;;
let append_2 x y = match x with h::t -> h::append_1 t y ;;
```

La fonction `append_2` est spécialisée pour un argument valant `[e1;e2]`, `append_1` pour `[e2]` et `append_0` pour `[]`. Ainsi, nous avons réussi à réduire à une seule branche les structures conditionnelles originales. Lorsque nous analysons les événements qui ont eu lieu, nous nous apercevons que trois événements apparaissent. Tous trois concernent la structure `match x with ...` de la fonction `append`. Dans `append2`, le premier constructeur de la liste `[e1;e2]` est testé, dans `append1`, le second constructeur de liste et dans `append0`, la liste vide. Ainsi, si nous arrivons à distinguer ces trois constructeurs, nous avons trois événements distincts et les trois versions spécialisées de `append` vont pouvoir être créées car elles sont activées par trois 'événements distincts.

Lorsque ces trois constructeurs proviennent de codes différents, il est facile de les distinguer en associant à chaque constructeur un indice supplémentaire indiquant le code où il a été créé. Par exemple, si la fonction ci-dessus avait été appelée par la fonction `f` qui ajoute la liste `[1;2]` devant la liste donnée en argument :

```
let f y = append [1;2] y ;;
```

Dans ce cas, les constructeurs de la liste `[1;2]` ne proviennent pas du même code (la construction de la liste peut se réécrire `1::(2::[])` et les deux constructeurs de liste et la liste vide apparaissent clairement).

Les interpréteurs basés sur l'évaluation d'expressions comme l'interpréteur TP forment des exemples bien adaptés à ce mécanisme de différenciation par le code qui a généré une valeur. En effet, lorsque le programme interprété est connu lors de l'évaluation partielle, le spécialiseur découvre facilement la structure du programme interprété en spécialisant la boucle principal de l'interpréteur pour chaque expression (et sous-expression) du programme.

Dans cette première version de la notion d'événement, la partie constructeur est caractérisée par un constructeur (constructeur de liste, liste vide, ...) et par le code qui l'a créé. Deux constructeurs du même type, construit avec le même code mais à l'intérieur de deux clones d'une même fonction sont identifiés. Cela n'est pas toujours satisfaisant. Considérons par exemple la fonction `g` suivante qui concatène trois listes dont la première est la liste `[1;2]` :

```
let g x y = append (append [1;2] x) y ;;
```

Suivant l'approche précédente, le spécialiseur crée trois versions de `append` correspondant à l'appel `(append [1;2] x)`. Cet appel renvoie la valeur `1::2::x` (avec `x` la valeur de la variable correspondante). Les deux constructeurs de liste sont construits par le même code de la fonction originale `append` mais dans deux clones différents. Nous ne pouvons pas les distinguer et la spécialisation du second appel à la fonction `append` ne fonctionne pas correctement. En effet, le premier événement porte sur le premier constructeur de liste. Le second sur le deuxième constructeur. Comme ces deux constructeurs sont identifiés (même type et même code générateur), entre le premier et le second appel récursifs de `append`, aucun nouvel événement n'a lieu et les appels sont identifiés. La fonction `g` sera mal spécialisée.

Si nous désirons pouvoir spécialiser cet exemple correctement, nous devons pouvoir distinguer les deux clones de constructeurs de liste créés par les clones de la fonction `append`. Or, si nous regardons plus précisément comment ces deux valeurs ont été générées, nous nous apercevons que la première dépend de l'événement avec le premier constructeur de liste de la liste `[1;2]` et la seconde de celui entre le second constructeur de cette même liste. Par conséquent, deux constructeurs générés par le même code mais dans deux clones de fonctions différents devraient être différenciés.

Malheureusement, si nous distinguons tous les constructeurs par le clone dans lequel ils se trouvent, le domaine devient infini car le nombre de clones n'a aucune raison d'être borné. Le contre-exemple suivant démontre cela :

```
let rec f = function h::t -> f(h::t) | [] -> () ;;
```

```
let g () = f [1] ;;
```

La fonction `f`, exécutée avec une liste non vide, boucle. Sinon elle renvoie la valeur `()`. Avec un mécanisme d'événement trop précis, la spécialisation de `g` boucle. Si la partie constructeur des événements est constituée du constructeur indexé par *le clone de la fonction* où il a été créé, alors, l'évaluateur partiel va produire une infinité de clones de la fonction `f` :

```
g ()
  f [1]
    f [1]
      ....
```

Chaque appel diffère du précédent par le constructeur de la liste en argument : pour le premier appel à `f`, il provient de la liste initiale, pour le second, du constructeur construit par ce premier clone de `f`, pour le troisième, du constructeur construit par le second clone...

Cet exemple nous montre que différencier deux constructeurs lorsqu'ils n'appartiennent pas au même clone peut conduire à faire boucler le programme de spécialisation. Les événements sont ici trop précis. Nous devons trouver une définition des événements moins précise que celle définie ci-dessus mais plus précise que celle du code qui l'a produit.

Dans ce travail, nous allons adopter un mécanisme d'événements à deux niveaux. Le niveau supérieur distingue deux constructeurs lorsqu'ils ne proviennent pas du même code. Le niveau inférieur distingue deux constructeurs produit par le même code si l'ensemble des événements de premier niveau qui les ont générés sont égaux.

Le chapitre suivant décrit le mécanisme de clonage en utilisant cette notion d'événement.

Chapitre 4

Événements et clonage

Ce chapitre est consacré à la description du mécanisme de clonage en utilisant les notions de valeurs alternatives et d'événement. Cette description porte sur les points suivants :

- Description du clonage avec le langage CL : passage d'un programme source à un programme cloné par l'ajout d'index de spécialisation et propagation des valeurs alternatives dans les expressions clonées.
- Description de la définition des index de spécialisation par la notion d'événement. L'identification de deux clones résulte de ce mécanisme qui s'attache à comparer les valeurs par l'analyse des ressources qui ont été nécessaires à leur création.

Dans un premier temps, nous aborderons l'algorithme de clonage avec une description algébrique des événements. Cette algébrisation laisse beaucoup de liberté sur le choix de leur définition. Dans une seconde partie, nous décrirons les solutions que nous avons retenus pour mettre en œuvre les événements. Nous aborderons cette question à travers différents exemples.

Les notions importantes liées au clonage portent sur les points suivants :

- Description statique d'un programme en cours de clonage par la description de fonctions sur les index de spécialisation.
- Description des mécanismes introduisant ou modifiant le programme en cours de clonage par modification des fonctions sur les index de spécialisation résultant de la propagation à travers les expressions clonées des valeurs alternatives. Le point crucial concerne la transformation d'un appel vers un clone de fonction en un appel récursif vers un autre clone de fonction car ce mécanisme est à la base de l'identification de fonctions spécialisées (un peu l'équivalent de la comparaison entre fonctions spécialisées dans les évaluateurs partiels contemporains, basée sur l'égalité des parties connues des paramètres des fonctions).
- Choix sur la définition des événements et description des événements attachés aux objets spécialisés.

4.1 Clones de fonctions et d'expressions

Comme l'opération de clonage consiste à dupliquer les fonctions du programme original et à spécialiser les structures de contrôle, nous devons pouvoir identifier les clones entre eux et leur

associer certaines informations. Pour cela, nous associons un index à chaque clone de fonction et à chaque objet du programme cloné.

Soit $\mathcal{I}$ un ensemble d'index. Nous supposons que cet ensemble est infini (dénombrable) et comporte deux éléments particuliers notés $\perp_{\mathcal{I}}$ et 0. L'index $\perp_{\mathcal{I}}$ sert à désigner l'objet initiale du programme source qui n'a pas encore été spécialisé. L'index 0 correspond à au clone de la fonction à spécialiser.

Le clone d'une fonction f est un couple (f, i) noté f^i avec $f \in \mathcal{F}$ et $i \in \mathcal{I}$. L'ensemble des clone est noté $\mathcal{F}^{\mathcal{I}}$. La valeur $(f, \perp_{\mathcal{I}})$ désigne la fonction f initiale (non spécialisée).

De même, nous parlerons de clone de constructeurs $\mathcal{C}^{\mathcal{I}}$, de clone d'opérateurs $\mathcal{O}^{\mathcal{I}}$ pour une primitive, de clone de structures de contrôle, de clone d'une branche d'une conditionnelle et de clone d'appel de fonction.

Comme les index forment un ensemble ne distinguant pas le type de l'objet qu'il réfère, nous noterons :

- $\mathcal{I}_{\mathcal{F}}$ le sous-ensemble des éléments de $\mathcal{I}$ se référant à un clone d'une fonction.
- $\mathcal{I}_{\mathcal{C}}$ pour les clones de constructeurs.
- $\mathcal{I}_{\mathcal{O}}$ pour les opérateurs.
- $\mathcal{I}_{\mathcal{M}}$ pour les clones de structures de contrôle.
- $\mathcal{I}_{\mathcal{B}}$ pour les clones des branches des conditionnelles.
- $\mathcal{I}_{\mathcal{A}}$ pour les clones d'appel de fonction.

Au cours du mécanisme d'évaluation partielle, le programme initial se transforme peu à peu en une version spécialisée. Soit P un programme étiqueté et f_P la fonction de P à spécialiser. Le mécanisme de clonage consiste, à partir de f_P et de P, à spécialiser cette fonction en utilisant les définitions de fonctions de P. Le résultat est un couple composé d'une fonction f_P^0 qui correspond à un clone de la fonction originale f_P et d'un programme cloné de P.

Nous supposons que les arguments de la fonction f_P sont tous inconnus lors du processus de spécialisation. Ils prennent la valeur $\top$.

Les informations concernant chaque clone sont conservées et mises à jour à travers un certain nombre de fonctions qui associent à chaque objet les informations nécessaires au mécanisme de spécialisation. En particulier, la fonction

$$Index : \mathcal{I} \to \mathcal{L} \to \mathcal{I}$$

permet de retrouver l'index associé à chaque clone d'expression du corps d'une fonction ou d'une branche d'une structure de contrôle et de retrouver l'index de l'objet défini par cette expression (clone de constructeur, ...). Le premier argument de $Index$ appartient à $\mathcal{I}_{\mathcal{F}} \cup \mathcal{I}_{\mathcal{B}}$. Il désigne l'index du clone de fonction ou de branche de conditionnelle qui correspond à une expression. Nous parlerons d'*d'index d'environnement*. Le second argument correspond à l'étiquette de cet objet dans le programme source. Cette fonction $Index$ permet de ne pas introduire d'index aux clones d'expression puisqu'ils sont désignés par l'index de la fonction ou de la branche où cette expression apparaît (son index d'environnement) et par son étiquette.

Lorsqu'un nouveau clone de fonction f^i est créé mais que son corps n'a pas été encore analysé, $Index(i)$ est la fonction constante égale à $\perp_{\mathcal{I}}$. Un index significatif sera donné ultérieurement lorsque le corps de la fonction sera analysé.

4.1.1 Domaines des événements

Dans cette première partie, nous décrivons les événements par une abstraction de données, donnant une description algébrique de cette notion. La notion d'événement étant assez proche de celle d'environnement, nous supposerons l'existence d'un ensemble Φ dont les éléments décrivent des *environnements d'événements* et qui sont associés aux expressions du programme.

À chaque type d'objet correspond une notion d'événement qui est fonction de l'environnement d'événements dans lequel cet objet est créé et de son type.

Ainsi nous parlerons des événements attachés à un constructeur. Leur ensemble est noté $\Phi_{\mathcal{C}}$.

Les opérations sur les environnements d'événement sont :

- $AddMatch : \mathcal{C} \times \Phi_{\mathcal{C}} \times \mathcal{L} \to \Phi \to \Phi$: cette fonction ajoute l'événement constitué par la rencontre entre un clone de constructeur (dont on ne retient que le nom et les événements qui lui sont associés) et une structure de contrôle (désignée par son étiquette), dans l'environnement d'événements courant. Cette fonction sera appelée lorsque le programme rencontre une structure de contrôle dont la valeur de test provient (entre autre) d'un clone de constructeur. Elle permet de trouver l'environnement d'événement associé à la branche activée par ce constructeur. Nous parlerons de *structures de contrôle spécialisées.*

- $AddRMatch : \mathcal{C} \times \mathcal{L} \to \Phi \to \Phi$: ajoute l'événement constitué par la rencontre entre la valeur $\top$ et la branche d'une structure de contrôle (désignée par son étiquette) correspondant au constructeur. Nous parlerons de *structures de contrôle résiduelles* puisque la valeur testée est complètement inconnue.

- $AddCall : \mathcal{F}^{\mathcal{I}_{\mathcal{F}}} \to \Phi \to \Phi$: ajoute l'appel à un clone de fonction à un environnement d'événements. Lorsque le programme rencontre un appel à une fonction, $AddCall$ permet de trouver l'environnement d'événements associé au corps du clone de fonction qui remplacera la fonction originale appelée.

La projection des environnements d'événements sur le domaine des événements associés aux clones de constructeur est :

- $CEv : \mathcal{C} \times \mathcal{L} \to \Phi \to \Phi_{\mathcal{C}}$: donne les événements associés à un constructeur à partir de son étiquette et de l'environnement d'événements dans lequel il est créé.

Enfin, un mécanisme d'exception permet de contrôler les appels récursifs à une fonction et de remplacer un appel vers un clone de fonction par un appel à un clone déjà analysé. Ce test forme la base du mécanisme permettant de trouver les nouvelles définitions récursives des clones de fonctions. Il engendre une rupture du mécanisme d'évaluation car l'appel modifié est en cours d'analyse. Après le remplacement de cet appel par un appel vers une fonction définie antérieurement, l'évaluation reprend au niveau de cet appel. Nous noterons

$$TestRec : \mathcal{F} \to \Phi \to \texttt{unit}$$

la procédure qui génère ces exceptions. Comme cette procédure ne rend normalement aucun résultat significatif, la valeur retournée lorsque aucune exception n'est levée est l'unique élément de `unit`.

Nous n'aborderons pas le problème des primitives dans ce chapitre.

4.1.2 Structure d'un programme au cours du clonage

Lorsque un programme est spécialisé, sa structure se modifie progressivement. Au départ, il n'est constitué que du clone de la fonction initiale. Puis, le corps de cette fonction est analysé. Cela

entraîne la création de nouveaux clones. Ce processus prend fin lorsque aucun nouveau clone ne peut être créé et que les valeurs associées aux clones ont atteint un point fixe. Le clonage repose sur un calcul de point fixe similaire à une interprétation abstraite associé à un mécanisme qui engendre ou identifie des clones.

Nous pouvons diviser la description de l'état du système à un instant donné, en deux parties. La première décrit le programme au cours de son évolution. Elle est constituée de la liste des objets du programme, associée aux liens entre ces objets. Nous l'appellerons la *description statique* du programme. À la fin du clonage, elle fournit le résultat recherché qui est constitué du programme cloné. La seconde partie donne les informations nécessaires à l'évolution de ce programme. Elle porte sur les environnements, les valeurs des arguments et du résultat des objets. Nous la nommerons *propriétés dynamiques*.

Description statique

Cette description permet de donner le squelette du programme en cours de clonage. La description statique de chaque type d'objet est la suivante :

- Pour le clone f^i d'une fonction : la fonction

  $$Index^i : \mathcal{L} \to \mathcal{I}$$

  permet de trouver l'index des clones des objets apparaissant dans le corps de ce clone. L'index $\perp_I$ indique que ce clone n'a pas encore été analysé.

- Pour le clone c^i d'un constructeur : la fonction

  $$GetCEv : \mathcal{I}_\mathcal{C} \to \Phi_\mathcal{C}$$

  permet de trouver les événements associés à ce clone. Au niveau des structures de contrôle, deux constructeurs du même type et avec les mêmes événements ne génèrent qu'une seule branche spécialisée (nous les identifions de cette manière).

- Pour le clone m^i d'une structure de contrôle : les structures de contrôle sont divisées en deux catégories. Le prédicat

  $$RMatch : \mathcal{I}_\mathcal{M} \to bool$$

  permet de savoir si une structure de contrôle est résiduelle ou spécialisée.

  Les structures de contrôle résiduelles correspondent à une structure de contrôle dont la valeur testée est $\top$. Dans ce cas, toutes les branches de la conditionnelle d'origine sont présentes. La fonction

  $$RBranch : \mathcal{I}_\mathcal{M} \to \mathcal{C} \to \mathcal{I}_\mathcal{B}$$

  renvoie le clone de branche de conditionnelle correspondant à un constructeur.

  Par contre, une structure de contrôle spécialisée comporte un ou plusieurs clones de branches de conditionnelle. Dans ce cas, la fonction

  $$Cases : \mathcal{I}_\mathcal{M} \to \mathcal{P}(\mathcal{C} \times \Phi_C)$$

  renvoie les couples composés d'un constructeur et des événements qui ont généré les branches spécialisées. La fonction

  $$Branch : \mathcal{I}_\mathcal{M} \to \mathcal{C} \times \Phi_C \to \mathcal{I}_\mathcal{B}$$

  renvoie l'index de la branche correspondante. Il permet de retrouver à travers la fonction $Index$, les index des objets appartenant à cette branche.

- Pour un clone b^i d'une branche d'une structure de contrôle spécialisée : la fonction

$$Index^i : \mathcal{L} \to \mathcal{I}$$

retourne les index des objets apparaissant dans cette branche.

- Pour le clone d'un appel à une fonction : deux cas peuvent se produire. Soit cet appel pointe vers un clone créé spécialement pour cet appel, soit cet appel correspond à un appel récursif. Le prédicat

$$RecFun : \mathcal{I}_{\mathcal{A}} \to bool$$

est vrai si le clone pointe vers une fonction définie précédemment. Dans les deux cas, la fonction

$$FClone : \mathcal{I}_{\mathcal{A}} \to \mathcal{I}_{\mathcal{F}}$$

renvoie l'index du clone appelé.

Nous noterons h^i pour $h(i)$ lorsque i est un index et h une fonction dont le premier argument est un index.

Le dernier type de clone, concernant les appels de fonctions, demande quelques explications. La structure globale du programme est, à chaque instant, un arbre mêlant deux structures. La première structure suit l'arborescence des expressions. La seconde suit la définition des clones de fonctions, à travers les appels spécialisés. Cet arbre a pour racine la fonction initiale à spécialiser. La structure arborescente suit les clones d'appels de fonction :

- Un appel dit non récursif a pour fils les appels apparaissant dans le clone d'expression qui définit la fonction appelée.

- Un appel récursif est une feuille. Il réfère une fonction définie plus près de la racine.

Propriétés dynamiques

Au cours de l'évaluation, nous avons besoin de conserver un certain nombre d'informations sur chaque clone. Elles nous permettent soit d'atteindre un point fixe sur les valeurs associées aux expressions, soit de modifier la structure du programme (ajouter de nouveaux cas à une structure de contrôle spécialisée, transformer une structure de contrôle spécialisée en une structure de contrôle résiduelle, analyser un appel à une fonction non encore traité ou bien transformer un appel non récursif en un appel récursif).

Pour chaque type d'objet, ces propriétés sont les suivantes :

- Pour un clone de fonction :

  - la fonction

$$FArgs : \mathcal{I}_{\mathcal{F}} \to \mathcal{X} \to \mathcal{V}$$

renvoie la valeur associée aux arguments de ce clone.

  - la fonction

$$FRet : \mathcal{I}_{\mathcal{F}} \to \mathcal{V}$$

retourne la valeur associée au résultat de ce clone.

- la fonction

$$Events : \mathcal{I}_{\mathcal{F}} \rightarrow \Phi$$

retourne l'environnement d'événements associé à ce clone. Cet environnement d'événements est donné à tous les clones d'objet apparaissant dans l'expression définissant ce clone jusqu'aux premières branche de conditionnelle (où l'index d'environnement change).

- Pour un clone de constructeur : la fonction

$$CArg : \mathcal{I}_{\mathcal{C}} \rightarrow \mathcal{V}$$

retourne la valeur associée à son argument.

- Pour un clone d'une branche d'une structure de contrôle : la fonction

$$Events : \mathcal{I}_{\mathcal{B}} \rightarrow \Phi$$

retourne l'environnement d'événements associé à cette branche. Cet environnement d'événements est donné à tous les clones d'objet apparaissant dans l'expression définissant ce clone jusqu'aux premières branche de conditionnelle (où l'index d'environnement change).

D'autres informations sur les clones sont aussi conservées dans la mise en œuvre du clonage (le programme LaMix) pour des raisons de stratégies d'évaluation pour atteindre le point fixe et pour des raisons d'efficacité du mécanisme de spécialisation. Nous ne les introduirons pas dans ce chapitre.

Les propriétés dynamiques décrivent les valeurs associées aux arguments et au résultat des deux principaux types de clones apparaissant dans un programme (clones de fonction et clones de constructeur). Le mécanisme de spécialisation s'achève lorsque ces valeurs ne sont plus modifiées et qu'aucun nouveau clone n'est créé au cours d'une phase d'évaluation.

4.1.3 Analyse et clonage

Cette section décrit l'algorithme de clonage. Comme nous l'avons déjà souligné, il repose sur deux concepts. Le premier est un mécanisme d'évaluation similaire à une interprétation abstraite et repose sur un calcul de point fixe. Le second mécanisme modifie la structure du programme en utilisant les transformations suivantes :

- un index i est créé pour chaque objet non encore analysé lorsque l'évaluateur analyse pour la première fois l'expression $\ell : \cdots$ qui le définit (lorsque $Index^k(\ell) = \bot_{\mathcal{I}}$ avec k l'index d'environnement de l'expression). $Index^k(\ell)$ est positionné à i. Alors, pour chaque type d'expression :
  - pour un constructeur c^i, $CArg^i$ est initialisé à $\bot$ et $GetCEv^i$ à $CEv(c, \ell)(Events^k)$.
  - pour une structure conditionnelle m^i, $RMatch^i$ est positionné à faux (laissant croire pour le moment que cette structure sera spécialisée). L'ensemble $Cases^i$ est vide (aucune branche).
  - pour un appel a^i à une fonction f correspondant à l'expression l : `call` $f(\cdots)$, le programme regarde s'il ne peut pas remplacer un appel à la même fonction apparaissant plus près de la racine par un appel récursif. Nous verrons ce mécanisme en détail ultérieurement. Il est réalisé par la procédure $TestRec$. Dans ce cas, l'évaluation reprend au

niveau de l'appel modifié et nous avons une rupture du mécanisme d'analyse. Dans le cas contraire, le programme initialise $RecFun^i$ à faux, crée un nouveau clone de fonction f^j et positionne $FClone^i$ à j. La création de f^j consiste à initialiser $Index^j$ à la fonction constante égale à $\perp_{\mathcal{I}}$, $FArgs^j$ à la fonction constante égale à $\perp$, $FRet^j$ à $\perp$ et $Events^j$ à $AddCall(f^j)(Events^k)$.

- lorsque le programme rencontre une structure de contrôle m^i spécialisée ($RMatch^i$ est faux) mais que la valeur testée vaut $\top$, il la transforme en structure de contrôle résiduelle. Les éventuelles branches spécialisées sont oubliées. Cela consiste à positionner $RMatch^i$ à vrai, à créer autant de clones de branche de conditionnelle que cette structure n'en comporte dans le programme source et à initialiser $RBranch$. Créer une branche b^j correspondant au constructeur c consiste à initialiser $Index^j$ à la fonction constante égale à $\mathcal{I}_\perp$ et $Events^j$ à $AddRMatch(c, \ell)(Events^k)$.

- lorsque le programme rencontre une structure de contrôle spécialisée m^i et que la valeur testée comporte un clone de constructeur c^j tel que $(c, GetCEv^l)$ n'est pas un motif de m^i ($Cases^i$), une nouvelle branche est créé. Cela consiste à ajouter $(c, GetCEv^l)$ à $Cases^i$, à créer un clone b^j de branche à partir de la branche correspondant à c et à positionner $Branch^i(c, GetCEv^l)$ à j. La création de la branche b^j initialise $Index^j$ à la fonction constante égale à $\mathcal{I}_\perp$ et $Events^j$ à $AddMatch(c, GetCEv^l, \ell)(Events^k)$.

Après ces modifications de structure, l'évaluation de l'expression est poursuivie.

La partie évaluation repose sur le domaine que nous avons présenté à la fin du chapitre précédent. Nous le rappelons brièvement :

$$\begin{array}{rcll} \mathcal{V} & = & \top & \\ & | & \perp & \\ & | & P\ (\mathcal{V} \times ... \times \mathcal{V}) & \\ & | & S\ (C) & C \subset \mathcal{C}^{\mathcal{I}} \end{array}$$

La valeur $\top$ dénote une valeur complètement inconnue. Elle provient, en général, d'un des arguments de la fonction initiale à spécialiser et des valeurs déduites de celle-ci. Une autre source provient des primitives (nous n'aborderons pas la gestion des opérateurs dans ce travail). La valeur $\perp$ dénote une absence d'information sur une valeur. Elle est notamment donnée comme résultat d'un clone de fonction lorsque l'évaluateur partiel le rencontre pour la première fois. Cette valeur devrait disparaître à la fin du clonage sauf si une fonction ne retourne jamais de valeur. En particulier, les fonctions ne terminant jamais (par exemple la fonction `let rec loop x = loop x`) retournent $\perp$. Un autre cas se produit lorsque, pour une structure de contrôle spécialisée, aucun motif ne correspond à la valeur testée (cas d'une erreur de filtrage comme pour la fonction `hd` lorsque la liste donnée en argument est vide). La valeur $P(...)$ dénote le produit de plusieurs valeurs tandis que $S(...)$ dénote l'alternative entre plusieurs clones de constructeur. La fonction $CArg$ renvoie, à un instant donné, la valeur de l'argument de ces clones. Une valeur peut donc représenter une structure cyclique (comme une liste dont on ne connaît pas la longueur mais composée d'éléments connus).

Comme l'évaluation repose sur une interprétation abstraite, nous avons besoin de définir la borne inférieure et la borne supérieure de deux valeurs, notées respectivement $\sqcap$ et $\sqcup$. Le domaine des valeurs alternatives, muni de ces deux opérateurs, forme un treillis (non-complet).

$$\begin{array}{lcl} \top \sqcup v & = & \top \\ \perp \sqcup v & = & v \\ P(x_1, \ldots, x_n) \sqcup P(y_1, \ldots, y_n) & = & P(x_1 \sqcup y_1, \ldots, x_n \sqcup y_n) \\ S(C_1) \sqcup S(C_2) & = & S(C_1 \cup C_2) \end{array}$$

$$\begin{array}{lcl} \top \sqcap v & = & v \\ \bot \sqcap v & = & \bot \\ P(x_1,\ldots,x_n) \sqcap P(y_1,\ldots,y_n) & = & P(x_1 \sqcap y_1,\ldots,x_n \sqcap y_n) \\ S(C_1) \sqcap S(C_2) & = & S(C_1 \cap C_2) \end{array}$$

Les autres configurations (deux produits dont le nombre d'éléments ne correspond pas ou un produit et une alternative) ne peuvent se présenter si le programme original est bien typé.

Bien sûr, l'élément maximal de $\mathcal{V}$ est $\top$ et l'élément minimal $\bot$. La relation d'ordre $\sqsubset$ est définie par :

$$\begin{array}{ll} \bot \sqsubset v & \forall v \in \mathcal{V} \\ v \sqsubset \top & \forall v \in \mathcal{V} \\ P(x_1,\ldots,x_n) \sqsubset P(y_1,\ldots,y_n) & \textbf{si } x_1 \sqsubset y_1,\ldots,x_n \sqsubset y_n \\ S(C_1) \sqsubset S(C_2) & \textbf{si } C_1 \subset C_2 \end{array}$$

L'évaluation d'une expression est réalisée dans un environnement $\rho : \mathcal{X} \to \mathcal{V}$ qui associe à chaque paramètre et chaque variable locale une valeur. La fonction $(i, e, \rho) \mapsto E[\![e]\!]^i_\rho$ évalue l'expression e dans l'environnement ρ. Le paramètre i permet d'obtenir l'environnement d'événements associé à cette expression ($Events^i$) et surtout de retrouver et de modifier les informations concernant un clone de cette expression à travers la fonction $Index^i$. Les actions réalisées par cet évaluateur après vérification de l'existence du clone correspondant (voir la section précédente) sont :

- $E[\![\ l : \texttt{var}\ x]\!]^i_\rho$: retourne la valeur $\rho(x)$.

- $E[\![\ l : \texttt{cons}\ c(e)]\!]^i_\rho$: calcule la valeur v associée à e, ajoute éventuellement cette valeur à l'argument du clone : $CArg^j \mapsto CArg^j \sqcup v$, où $j = Index^i(l)$ est l'index du clone. La valeur retournée est $S\{c^j\}$.

- $E[\![\ l : \texttt{call}\ f(e_1,\ldots,e_n)]\!]^i_\rho$: calcule les valeurs $v_1,\ldots,v_n$ associées aux expressions $e_1,\ldots,e_n$ et ajoute ces valeurs aux arguments du clone : $FArgs^j(x_l) \mapsto FArgs^j(x_l) \sqcup v_l$, où $j = FClone^{Index^i(l)}$. Si cet appel n'est pas récursif ($RecFun^{Index^i(l)}$ est faux), le programme évalue le clone appelé. Si, au cours de l'évaluation du clone, une exception demande que cet appel soit transformé en appel récursif, le programme met à jour les fonctions $RecFun$ et $FClone$. Dans tous les cas, le programme retourne la valeur associée au résultat du clone appelé ($FRet^j$).

- $E[\![\ l : \texttt{let}\ (x_1,e_1),\ldots,(x_n,e_n)\ \texttt{in}\ e]\!]^i_\rho$: évalue les expressions $e_1,\ldots,e_n$ dans l'environnement ρ, lie les variables $x_1,\ldots,x_n$ aux valeurs calculées puis évalue l'expression e dans ce nouvel environnement.

- $E[\![\ l : \texttt{match}\ e\ \texttt{with}\ c_1(x_1)\ \texttt{->}\ e_1\ \texttt{|}\cdots\texttt{|}\ c_n(x_n)\ \texttt{->}\ e_n]\!]^i_\rho$: commence par évaluer l'expression e. Soit $j = Index^i(l)$ l'index de la conditionnelle. Trois cas peuvent se présenter :

  - Lorsque cette structure de contrôle est résiduelle ($RMatch^j$ est vrai), le programme évalue toutes les branches initiales de la structure dans un environnement où la variable x_i est liée à la valeur $\top$. La valeur retournée est la borne supérieure (au sens du treillis) des valeurs retournées par toutes les branches. L'index d'environnement des branches est récupéré avec la fonction $RBranch^j$.
  - Si la structure est spécialisée mais que la valeur testée (correspondant à e) vaut $\top$, la structure est transformée en structure de contrôle résiduelle et le programme continue comme dans le premier cas.

  - Dans le cas contraire, suivant la valeur testée, de nouvelles branches sont ajoutées. Puis, le programme évalue les branches spécialisées (correspondant à un élément de $Cases^j$) dans un environnement où la variable x_i est liée à la borne supérieure des arguments des clones de constructeur qui correspondent à ce cas. L'index d'environnement est récupéré avec $Branch^j$. Ensuite, le programme retourne la borne supérieure des valeurs retournées par les différentes branches. Il est à noter que s'il n'existe aucune branche spécialisée, la valeur retournée est $\bot$. Ce cas se produit lorsque la valeur testée vaut $\bot$ ou ne correspond à aucune branche du programme original (erreur de filtrage).

- $E[\![\ l : \pi_m^n(e)]\!]_\rho^i$: évalue l'expression e, puis, suivant la valeur retournée :
  - avec $\bot$: retourne $\bot$.
  - avec $\top$: retourne $\top$.
  - avec $P(v_1, ..., v_n)$: retourne v_m.
- $E[\![\ l : (e_1, ..., e_n)]\!]_\rho^i$: évalue les n expressions puis retourne $P(v_1, ..., v_n)$ où v_m est la valeur retournée par la m-ième expression.

L'évaluation d'un clone f^i de fonction où f est définie par `def` $f(x_1, \ldots, x_n)$ `=` e, se déroule comme suit :

- Le programme évalue le corps e du clone dans un environnement où les paramètres $x_1, \ldots, x_n$ de cette fonction sont liées aux valeurs des arguments données par $FArgs^i$:

$$E[\![\ e]\!]^i_{FArgs^i}$$

- le résultat v de cette évaluation est ajouté au résultat du clone : $FRet^i \mapsto FRet^i \sqcup v$.
- Suivant la stratégie d'évaluation, cette fonction peut être réévaluée une nouvelle fois (suivant les mises en œuvre).

Le programme de clonage se déroule de la manière suivante :

- Le programme commence par évaluer la fonction f_P^0 à spécialiser avec pour valeur associée aux paramètres, la valeur $\top$ ($FArgs^0$ est la fonction constante égale à $\top$). La fonction $Index^0$ est initialisée à $\bot_{\mathcal{I}}$
- le processus recommence tant que les fonctions $CArg$, $FArgs$ $FRet$ n'ont pas atteint un point fixe et que de nouveaux clones apparaissent (fonctions $Index$, $RMatch$, $Cases$, $RecFun$, $FClone$).

4.1.4 Exemples de clonage

Pour comprendre un peu mieux le mécanisme de clonage décrit ci-dessus, nous présentons quelques exemples de spécialisation.

Clonage de `append c x`

Reprenons l'exemple de la spécialisation de la fonction `append` lorsque son premier argument est une liste connue mais que le second est inconnu. La spécialisation porte sur la fonction `f` définie par le programme étiqueté suivant (les mots-clés `call` , `cons` ... n'ont pas été repris) :

```
def append x y = 1:match x with
```

```
      Cons(p) -> 2:Cons(fst(p), 3:append (snd(p)) y)
    | Nil(_)  -> y
def f z = 4:append (5:Cons(2, 6:Cons(1, 7:Nil))) z
```

Les étiquettes sont représentés par des entiers et certains ont été omis. Les fonctions `fst` et `snd` sont les projections sur la première et la seconde composante d'une paire.

Le mécanisme de clonage commence par créer un premier clone pour la fonction `f`, noté `f_0`. Puis, le corps de cette fonction est évalué avec `z` valant $\top$. Des clones sont créés pour l'appel à la fonction `append` (noté `append_1`) et pour les constructeurs (`Cons_A`, `Cons_B` et `Nil_C`). Le programme cloné est alors :

```
def f_0 z = 4:append_1 (5:Cons_A(2, 6:Cons_B(1, 7:Nil_C))) z
```

Puis, l'exécution continue par l'évaluation du premier clone de la fonction `append` avec pour `x` une alternative comportant un seul clone de constructeur : celui correspondant à `Cons_A`. La structure de contrôle de `append_1` est spécialisée pour cette valeur. Une branche spécialisée est créée et l'évaluation continue avec cette branche. Un nouvel appel à `append` est rencontré. Le mécanisme de transformation d'un appel en un appel non-récursif échoue (on ne peut encore comparer les événements car il n'existe que deux appels à `append`). Un nouveau clone `append_2` est créé et la structure du programme devient :

```
def append_1 x y = 1:match x with
      Cons_A(p) -> 2:Cons_D(fst(p), 3:append_2 (snd(p)) y)
def f_0 z = 4:append_1 (5:Cons_A(2, 6:Cons_B(1, 7:Nil_C))) z
```

L'évaluation continue avec ce nouveau clone `append_2`. De même que pour `append_1`, la structure de contrôle est spécialisée pour le constructeur `Cons_B`. Le programme rencontre un nouvel appel à `append`. La comparaison entre les événements s'étant produit, d'une part, entre le premier appel `append_1` et le second appel `append_2` et, d'autre part, entre `append_2` et ce nouvel appel, les montre incompatibles. Un nouveau clone `append_3` est donc créé. En effet, le seul événement arrivant entre `append_1` et `append_2` est la rencontre du constructeur `Cons_A` avec la structure de contrôle dont l'étiquette est `1`. De même, l'événement entre `append_2` et `append_3` est la rencontre du constructeur `Cons_B` avec la même structure de contrôle. Ces deux constructeurs n'ayant pas la même étiquette, ils sont incomparables (le code du programme initial qui les crée est différent).

Le programme devient alors :

```
def append_2 x y = 1:match x with
      Cons_B(p) -> 2:Cons_E(fst(p), 3:append_3 (snd(p)) y)
def append_1 x y = 1:match x with
      Cons_A(p) -> 2:Cons_D(fst(p), 3:append_2 (snd(p)) y)
def f_0 z = 4:append_1 (5:Cons_A(2, 6:Cons_B(1, 7:Nil_C))) z
```

L'exécution continue avec l'évaluation de ce nouveau clone `append_3`. Cette fois la valeur testée correspond au constructeur `Nil_C`. Par conséquent, le programme sélectionne la seconde branche. Le programme est maintenant décrit par :

```
def append_3 x y = 1:match x with
      Nil_C(_)  -> y
def append_2 x y = 1:match x with
      Cons_B(p) -> 2:Cons_E(fst(p), 3:append_3 (snd(p)) y)
def append_1 x y = 1:match x with
      Cons_A(p) -> 2:Cons_D(fst(p), 3:append_2 (snd(p)) y)
def f_0 z = 4:append_1 (5:Cons_A(2, 6:Cons_B(1, 7:Nil_C))) z
```

À ce moment là, la valeur de retour de `append_3` est fixée à $\top$. Elle provient de l'argument initial de `f`. Ensuite, les valeurs de retours des autres clones sont calculées. A la fin de cette première phase, le programme est défini comme ci-dessus. Les valeurs associées aux différents clones sont :

$$\begin{array}{lcl}
CArg(\texttt{Cons_A}) & = & P(2, S\{\texttt{Cons_B}\}) \\
CArg(\texttt{Cons_B}) & = & P(1, S\{\texttt{Nil_C}\}) \\
CArg(\texttt{Cons_D}) & = & P(2, S\{\texttt{Cons_E}\}) \\
CArg(\texttt{Cons_E}) & = & P(1, \top) \\
FArgs(\texttt{f_0}) & = & \texttt{z} \mapsto \top \\
FRet(\texttt{f_0}) & = & S\{\texttt{Cons_D}\} \\
FArgs(\texttt{append_1}) & = & \texttt{x} \mapsto S\{\texttt{Cons_A}\} \\
 & & \texttt{y} \mapsto \top \\
FRet(\texttt{append_1}) & = & S\{\texttt{Cons_D}\} \\
FArgs(\texttt{append_2}) & = & \texttt{x} \mapsto S\{\texttt{Cons_B}\} \\
 & & \texttt{y} \mapsto \top \\
FRet(\texttt{append_2}) & = & S\{\texttt{Cons_E}\} \\
FArgs(\texttt{append_3}) & = & \texttt{x} \mapsto S\{\texttt{Nil_C}\} \\
 & & \texttt{y} \mapsto \top \\
FRet(\texttt{append_3}) & = & \top
\end{array}$$

Un nouveau cycle d'évaluation est exécuté car le programme a été modifié. Comme ce cycle ne crée pas de nouveau clone et que les valeurs qui sont associées aux objets restent inchangées, le clonage s'arrête sur ce programme.

Nous voyons que trois clones de `append` ont été créés, ce qui est le résultat attendu. Les phases postérieures de spécialisation pourront déplier ces fonctions et simplifier les arguments pour donner finalement le programme :

```
def f_0 z = Cons_D(2,Cons_E(1,z))
```

Clonage de `append x c`

Spécialisons maintenant la fonction `append` lorsque le premier argument est inconnu et que le second est la liste `[2;1]`. Le programme s'écrit :

```
def append x y = 1:match x with
      Cons(p) -> 2:Cons(fst(p), 3:append (snd(p)) y)
    | Nil(_)  -> y
def f z = 4:append z (5:Cons(2, 6:Cons(1, 7:Nil)))
```

Le clonage commence par créer le clone `f_0` puis le clone `append_1`, comme dans l'exemple précédent :

```
def f_0 z = 4:append_1 z (5:Cons_A(2, 6:Cons_B(1, 7:Nil_C)))
```

Cette fois-ci, la valeur testée par la structure de contrôle de `append_1` vaut $\top$. Cette structure devient résiduelle et les deux branches sont évaluées. Le cas correspondant à `Cons` donne un nouveau clone de `append` (`append_2`). Le programme est alors :

```
def append_1 x y = 1:rmatch x with
      Cons(p) -> 2:Cons_D(fst(p), 3:append_2 (snd(p)) y)
    | Nil(_)  -> y
def f_0 z = 4:append_1 z (5:Cons_A(2, 6:Cons_B(1, 7:Nil_C)))
```

Le mot-clé `rmatch` indique que la structure de contrôle est résiduelle.

L'exécution se poursuit par l'analyse `append_2`. De même que pour `append_1`, la structure de contrôle est résiduelle. Le programme devient :

```
def append_2 x y = 1:rmatch x with
      Cons(p) -> 2:Cons_E(fst(p), 3:append (snd(p)) y)
    | Nil(_)  -> y
def append_1 x y = 1:rmatch x with
      Cons(p) -> 2:Cons_D(fst(p), 3:append_2 (snd(p)) y)
    | Nil(_)  -> y
def f_0 z = 4:append_1 z (5:Cons_A(2, 6:Cons_B(1, 7:Nil_C)))
```

Dans la branche correspondant à `Cons`, le spécialiseur rencontre un nouvel appel à `append`. Il regarde alors s'il ne doit pas transformer un appel en appel récursif. L'événement entre `append_1` et `append_2` est la rencontre de $\top$ avec la structure de contrôle de `append_1`. L'événement entre `append_2` et ce nouvel appel est la rencontre entre $\top$ et la structure de contrôle résiduelle de `append_2`. Ces deux événements sont similaires. Par conséquent, le programme décide de transformer l'appel à `append_2` apparaissant dans `append_1` en un appel récursif vers `append_1`. Une exception est levée qui est capturée par la fonction d'évaluation de l'appel à `append_2`. Celle-ci transforme cet appel pour donner le programme suivant :

```
def append_1 x y = 1:rmatch x with
      Cons(p) -> 2:Cons_D(fst(p), 3:rcall append_1 (snd(p)) y)
    | Nil(_)  -> y
def f_0 z = 4:append_1 z (5:Cons_A(2, 6:Cons_B(1, 7:Nil_C)))
```

Le mot-clé `rcall` indique un appel récursif.

Ensuite, le calcul du point fixe se poursuit sans que n'apparaissent de nouveaux clones. Le programme nécessite un cycle supplémentaire pour calculer la valeur de l'argument de `Cons_D`. Lors du premier cycle, cette valeur vaut $P(\top, \bot)$ car, à ce moment là, la valeur associée au résultat de `append_1` est $\bot$. Lors de la seconde phase, la valeur associée à `append_1` est $S\{\texttt{Cons\_A}, \texttt{Cons\_D}\}$. Par conséquent, la valeur associée à l'argument de `Cons_D` est $P(\top, S\{\texttt{Cons\_A}, \texttt{Cons\_D}\})$. Le spécialiseur s'arrête au cycle suivant car les valeurs sont stabilisées.

Les valeurs associées aux clones sont :

$$
\begin{array}{lcl}
\mathit{CArg}(\texttt{Cons\_A}) & = & P(2, S\{\texttt{Cons\_B}\}) \\
\mathit{CArg}(\texttt{Cons\_B}) & = & P(1, S\{\texttt{Nil\_C}\}) \\
\mathit{CArg}(\texttt{Cons\_D}) & = & P(\top, S\{\texttt{Cons\_A}, \texttt{Cons\_D}\}) \\
\mathit{FArgs}(\texttt{f\_0}) & = & \texttt{z} \mapsto \top \\
\mathit{FRet}(\texttt{f\_0}) & = & S\{\texttt{Cons\_A}, \texttt{Cons\_D}\} \\
\mathit{FArgs}(\texttt{append\_1}) & = & \texttt{x} \mapsto \top \\
 & & \texttt{y} \mapsto S\{\texttt{Cons\_A}\} \\
\mathit{FRet}(\texttt{append\_1}) & = & S\{\texttt{Cons\_A}, \texttt{Cons\_D}\}
\end{array}
$$

Après le clonage, nous voyons que la programme n'a pas été modifié. Il ressemble au programme initial. Ici encore, le résultat est bien ce que nous cherchions. Une phase ultérieure pourra éliminer le paramètre `y` de `append_1`. De plus, cette phase a pu déterminer que le résultat de `append_1` et, par conséquent, celui de `f_0` est une liste de longueur inconnue mais se terminant toujours par la liste `[2;1]`.

Clonage de `append (append c x) y)`

Essayons de spécialiser une fonction `f` qui concatène trois listes dont la première est la liste `[2;1]`. Le programme suivant calcule cette fonction en utilisant deux fois la fonction `append` :

```
def append x y = 1:match x with
      Cons(p) -> 2:Cons(fst(p), 3:append (snd(p)) y)
    | Nil(_)  -> y
def f a b = 8:append
              (4:append (5:Cons(2, 6:Cons(1, 7:Nil))) a) b
```

La spécialisation de `f` commence car créer deux clones de `append`. La première `append_1` correspondant à l'étiquette `4` et la seconde `append_4` à l'étiquette `8`. Ensuite, le programme spécialise le clone `append_1`. Cette opération donne le même bout de programme que le premier exemple sur `append`. Nous obtenons alors le programme :

```
def append_3 x y = 1:match x with
      Nil_C(_)  -> y
def append_2 x y = 1:match x with
      Cons_B(p) -> 2:Cons_E(fst(p), 3:append_3 (snd(p)) y)
def append_1 x y = 1:match x with
      Cons_A(p) -> 2:Cons_D(fst(p), 3:append_2 (snd(p)) y)
def f_0 a b = 8:append_4
                (4:append_1
                      (5:Cons_A(2, 6:Cons_B(1, 7:Nil_C))) a) b
```

Puis, le programme spécialise le clone `append_4`. Ici, le code qui génère les clones de constructeur `Cons_D` et `Cons_E` est le même (leur étiquette est `2`). Pourtant, le mécanisme lié aux événements arrive à les distinguer car `Cons_D` a été créé grâce au constructeur `Cons_A` et `Cons_E` grâce au constructeur `Cons_B`. Or ces deux constructeurs ne portent pas la même étiquette. Ainsi, ces deux constructeurs, bien que portant la même étiquette (issu du même code du programme original), peuvent être différenciés, ainsi que les événements qu'ils engendrent. Par conséquent, nous obtenons le programme suivant :

```
def append_3 x y = 1:match x with
      Nil_C(_)  -> y
def append_2 x y = 1:match x with
      Cons_B(p) -> 2:Cons_E(fst(p), 3:append_3 (snd(p)) y)
def append_1 x y = 1:match x with
      Cons_A(p) -> 2:Cons_D(fst(p), 3:append_2 (snd(p)) y)
def append_6 x y = 1:rmatch x with
      Cons(p) -> 2:Cons_H(fst(p), 3:rcall append_6 (snd(p)) y)
    | Nil(_)  -> y
def append_5 x y = 1:match x with
      Cons_B(p) -> 2:Cons_G(fst(p), 3:append_6 (snd(p)) y)
def append_4 x y = 1:match x with
      Cons_A(p) -> 2:Cons_F(fst(p), 3:append_5 (snd(p)) y)
def f_0 a b = 8:append_4
                (4:append_1
                      (5:Cons_A(2, 6:Cons_B(1, 7:Nil_C))) a) b
```

Les phases ultérieures de l'évaluateur partiel donneront le programme suivant :

```
def append_6 x y = 1:rmatch x with
      Cons(p) -> 2:Cons_H(fst(p), 3:rcall append_6 (snd(p)) y)
    | Nil(_)  -> y
def f_0 a b = Cons_F(1,Cons_G(2,append_6 a b))
```

Nous pouvons remarquer que nous aurions pu définir `f` de manière différente en utilisant l'associativité de `append`. Le clonage de cette version donnerait exactement le même programme spécialisé final. Toutefois, avec cette seconde version, le programme comporterait moins de clones et serait plus rapide à cloner.

4.2 Retour sur les événements

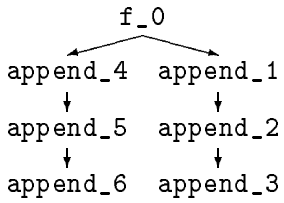
Dans le début de ce chapitre nous avons délibérément retardé la description exacte de la notion d'événement.

4.2.1 Buts de la notion d'événement

Nous avions introduit ce concept, à la fin du chapitre précédent, pour trouver un mécanisme d'identification des clones de fonctions (et des clones de constructeurs) ne reposant pas sur l'égalité entre les parties connues des arguments de ces clones. À la place de ces tests, nous voulions un mécanisme reposant sur la comparaison des événements qui ont activé deux clones d'une fonction (ou d'un constructeur).

Soit f, une fonction récursive. Petit à petit, le mécanisme de clonage va créer un certain nombre de clones de cette fonction. Considérons la relation qui existe entre ces clones. Nous avons dit que la structure des appels de fonction est, à chaque instant, un arbre. Ces clones forment une structure arborescente. Nous pouvons déterminer si un clone est un ancêtre d'un autre clone, un descendant ou bien s'ils sont simplement cousins.

Dans le dernier exemple de la section précédente, la structure arborescente du programme cloné final est la suivante :



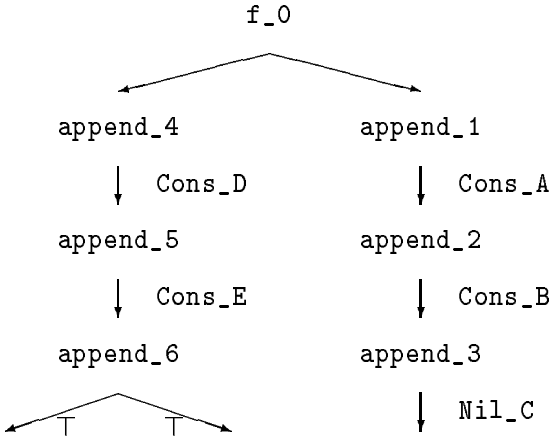
L'appel récursif vers `append_6` dans le corps de cette même fonction n'apparaît pas (puisqu'il est récursif).

Le mécanisme permettant d'identifier deux appels repose sur la comparaison des événements qui ont eu lieu entre les appels à une même fonction sur un chemin de cet arbre. Ainsi, nous pouvons comparer les événements qui ont eu lieu entre les appels à `append_1` et `append_2` et ceux qui ont eu lieu entre `append_2` et `append_3`. Par contre, le mécanisme de comparaison ne compare pas les événements entre `append_1` et `append_2` et ceux entre `append_4` et `append_5` car ils ne sont pas sur le même chemin.

Dans le mécanisme de clonage, cette comparaison a lieu uniquement lorsqu'un nouvel appel est rencontré (l'index de ce clone est $\perp_{\mathcal{I}}$). Cela est une conséquence des propriétés des environnements d'événements. En effet, les événements qui ont lieu entre le clone racine et un clone particulier

ne dépendent que de ce chemin et ne sont pas modifiés par les nouveaux cycles d'évaluation et la création de nouveaux clones.

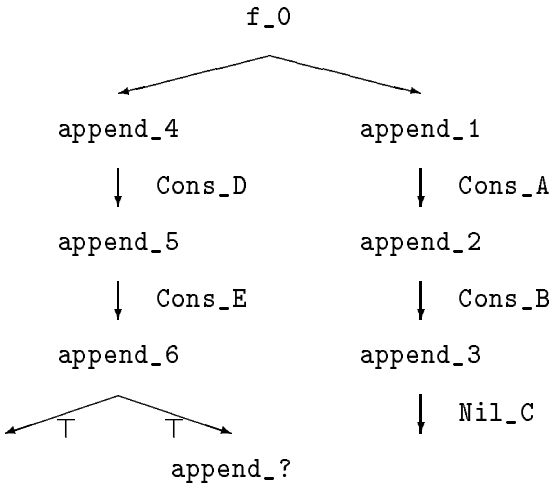
Le diagramme suivant précise les différents événements qui ont eu lieu entre les appels des clones pour l'exemple précédent :



Chaque flèche est étiquetée par la partie constructeur des événements qui ont activé le nœud correspondant. En effet, la partie structure de contrôle est toujours la même. Elle correspond au test effectué par la fonction `append` d'origine.

Le spécialiseur a créé tous ces clones car entre les appels à deux clones de `append`, les événements (le clone de constructeur correspondant) ont pu être différenciés des événements apparaissant entre deux autres appels à `append` sur le chemin qui mène à la racine. En effet, `Cons_E` est différent de `Cons_D`. De même, `Cons_B` est différent de `Cons_A`.

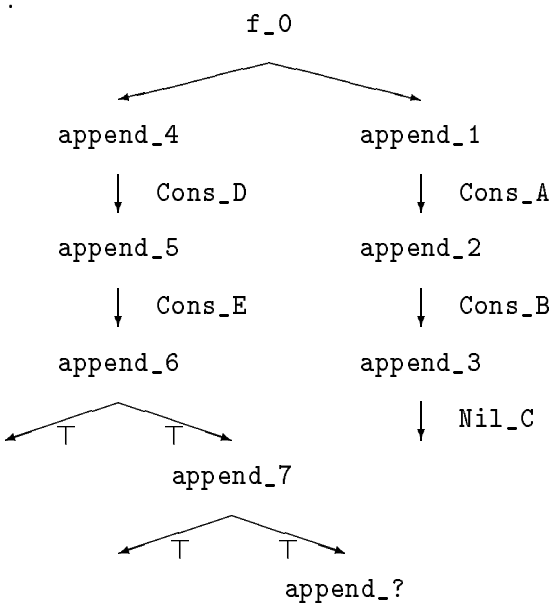
Dans cet exemple, nous avons identifié l'appel au clone `append_6` dans `append_5` avec l'appel apparaissant dans `append_6`. Pour expliquer ce phénomène, reprenons la spécialisation de ce programme lorsque l'évaluateur analyse pour la première fois le clone `append_6`. A ce moment là, la structure en arbre est :



Le clone `append_?` n'a pas encore été créé.

Lorsque l'évaluateur rencontre ce nouvel appel, il vérifie s'il ne doit pas transformer un appel en un appel récursif. Ce mécanisme parcourt l'environnement d'événements courant et calcule les événements qui se sont produits entre ce nouvel appel et le premier clone de `append` sur le chemin menant à la racine. Ce clone est `append_6`. Par conséquent, les événements qui se sont produits entre ce nouvel appel et l'appel précédent sont constitués du test de la valeur $\top$ par la structure de contrôle de `append_6`. Il compare alors ces événements avec les événements qui se sont produits

entre deux appels à des clones de `append` sur le chemin menant à la racine. Ce sont `Cons_E` et `Cons_D`. Comme ces deux événements sont incompatibles avec $\top$, le mécanisme d'identification des appels de clone échoue et un nouveau clone de `append` est créé pour remplacer `append_?`. Nommons le `append_7`. Ensuite, le programme continue en évaluant le corps de ce nouveau clone. Il tombe alors sur la situation suivante :



De nouveau, l'évaluateur tombe sur un appel à `append` et il doit vérifier s'il ne peut pas transformer un appel en un appel récursif. Pour cela, il calcule les événements entre `append_?` et `append_7` et trouve l'événement $\top$. Or, les événements entre `append_7` et `append_6` sont aussi composés du test avec $\top$. Ces deux listes d'événements sont comparables. Par conséquent, le programme décide de transformer l'appel à `append_7` apparaissant dans `append_6` par un appel récursif à `append_6`. Nous retombons alors dans la situation finale du programme comme nous l'avons vu plus haut.

Une remarque importante que nous devons signaler ici est que le mécanisme de transformation des appels en appels récursifs aurait pu identifier l'appel `append_?` avec l'appel à `append_7` dans `append_6`. Cela aurait donné un programme avec un clone supplémentaire de `append` (`append_7`) sans que l'efficacité du programme résultat soit améliorée. En identifiant, non pas l'appel en cours de création avec un appel plus près de la racine, mais, plutôt le précédent appel, nous limitons le nombre de clones. En général, le programme obtenu est plus satisfaisant car il comporte moins de variantes d'une même fonction.

4.2.2 Mécanisme d'identification de clones de fonction

L'identification des clones de fonctions est réalisée en transformant dans l'arbre du programme cloné, un appel vers un clone en un appel récursif vers un clone plus près de la racine.

Ce mécanisme repose sur la notion d'environnement d'événements qui décrit les événements qui ont eu lieu entre une expression d'un clone et la racine du programme cloné. Du point de vue de l'évaluation, cela représente les différentes structures de contrôle et d'appels de fonction que l'évaluateur a rencontré pour arriver à ce point du programme dans l'arbre des appels non récursifs. Nous pourrions aussi le décrire comme la pile des appels et des branches de conditionnelles empruntées pour activer ce point. Par exemple, avec le programme présenté dans la section précédente, pour arriver à l'appel du clone `append_6` apparaissant dans le clone `append_5`, l'évaluateur a dû traverser successivement :

– la fonction initiale `f_0`.

- l'appel à `append_4`.
- la branche correspondant à `Cons_D` de la structure de contrôle de `append_4`.
- l'appel à `append_5`.
- la branche correspondant à `Cons_E` de la structure de contrôle de `append_5`.

Un environnement d'événement est une liste composée soit d'un événement entre un constructeur et une structure de contrôle, soit d'un appel à un clone d'une fonction.

L'ensemble Φ des environnements d'événements est une liste et se décrit par :

$$\begin{array}{lcll} \phi & = & \psi_1, \ldots, \psi_n & \\ \psi & = & \mathrm{Call}(f^i) & f^i \in \mathcal{F}^{\mathcal{I}_\mathcal{F}} \\ & | & \mathrm{Match}(c, \phi_c, l) & c \in \mathcal{C}, \phi_c \in \Phi_\mathcal{C}, l \in \mathcal{L} \\ & | & \mathrm{RMatch}(c, l) & c \in \mathcal{C}, l \in \mathcal{L} \end{array}$$

Les éléments ψ composant les environnements d'événements sont appelés *événements*. Ils appartiennent à l'ensemble Ψ. Pour un événement entre un constructeur et une structure de contrôle spécialisée, c représente le type de constructeur, ϕ_c les événements associés au clone de ce constructeur et l l'étiquette de la structure de contrôle. Pour un événement entre $\top$ et une structure de contrôle résiduelle, l représente l'étiquette de la structure de contrôle et c le constructeur correspondant à la branche empruntée. Enfin, pour un appel à un clone de fonction, f est la fonction du programme initiale et i est l'index du clone appelé.

Nous reviendrons dans la section suivante sur les événements associés à un clone de constructeur $\Phi_\mathcal{C}$. Pour l'instant nous supposerons que cet ensemble est muni d'une relation d'équivalence noté $\equiv_{\Phi_\mathcal{C}}$.

Nous pouvons définir la notion d'*événements équivalents* par la relation d'équivalence $\equiv_\Psi$ suivante :

$$\begin{array}{lcll} \mathrm{Match}(c_1, \phi_{c1}, l_1) & \equiv_\Psi & \mathrm{Match}(c_2, \phi_{c2}, l_2) & \textbf{si } c_1 = c_2, \phi_{c1} \equiv_{\Phi_\mathcal{C}} \phi_{c2}, l_1 = l_2 \\ \mathrm{RMatch}(c_1, l_1) & \equiv_\Psi & \mathrm{RMatch}(c_2, l_2) & \textbf{si } l_1 = l_2, c_1 = c_2 \\ \mathrm{Call}(f_1^{i1}) & \equiv_\Psi & \mathrm{Call}(f_2^{i2}) & \textbf{si } f_1 = f_2 \end{array}$$

Soit $\phi = \psi_1, \ldots, \psi_n$ un environnement d'événements. Les fonctions sur les événements sont définies par :

$$\begin{array}{lcl} \mathit{AddCall}(f^i)(\phi) & = & \mathrm{Call}(f^i), \psi_1, \ldots, \psi_n \\ \mathit{AddMatch}(c, \phi_c, l)(\phi) & = & \mathrm{Match}(c, \phi_c, l), \psi_1, \ldots, \psi_n \\ \mathit{AddRMatch}(c, l)(\phi) & = & \mathrm{RMatch}(c, l), \psi_1, \ldots, \psi_n \end{array}$$

La procédure $\mathit{TestRec} : \mathcal{F} \to \Phi \to$ `unit` détermine si, lors d'un nouvel appel à une fonction dans un environnement d'événements particulier, un précédent appel à cette même fonction doit être transformé en un appel récursif. L'algorithme $\mathit{TestRec}\ f\ (\phi = \psi_1, \ldots \psi_n)$ effectue les opérations :

- Si l'ensemble $\{m | \psi_m = \mathrm{Call}(f^i), i \in \mathcal{I}_\mathcal{F}\}$ est vide, la procédure s'arrête sans lever d'exception car la fonction vient d'être appelée pour la première fois.
- Sinon, soit k le minimum de cet ensemble.
- Si l'ensemble

  $$\{m | \psi_m = \mathrm{Call}(f^i), i \in \mathcal{I}_\mathcal{F}, m \geq 1, \psi_{m+1} \equiv_\Psi \psi_1, \ldots, \psi_{m+k} \equiv_\Psi \psi_k\}$$

  est vide, la procédure s'arrête sans lever d'exception. Les événements ne sont pas comparables.
- Dans le cas contraire, soit m le minimum de cet ensemble. Soient i tel que $\psi_k = \mathrm{Call}(f^i)$ et Alors, la procédure lève une exception demandant que l'appel au clone f^i soit remplacé par un appel récursif au clone f^{i_0}.

4.2.3 Événements et constructeurs

Pour terminer la description des mécanismes associés aux événements, nous devons décrire les événements associés aux constructeurs.

Dans la section précédente, l'ensemble $\Phi_{\mathcal{C}}$ est utilisé sans qu'une définition précise ne soit produite. Dans cette section, seule la relation d'équivalence est nécessaire.

Ces événements associés aux clones de constructeur participent à deux mécanismes. Le premier a été décrit en détail dans la section précédente et consiste à remplacer un appel à un clone en un appel récursif. La notion importante était la relation d'équivalence entre événements associés à des constructeurs. Le second mécanisme repose sur la création des branches spécialisées des structures de contrôle spécialisées.

Lors de l'évaluation d'un telle structure, de nouvelles branches sont créées lorsque la valeur testée comporte des clones de constructeur non compatibles avec les branches existantes. La partie filtrage d'une branche étant composée d'un couple $(c, \phi_{\mathcal{C}})$ avec $c \in \mathcal{C}$ et $\phi_{\mathcal{C}} \in \Phi_{\mathcal{C}}$, un clone de constructeur c_1^i convient à cette branche si $c_1 = c$ et $GetCEv^i \equiv_{\Phi_{\mathcal{C}}} \phi_{\mathcal{C}}$. En somme, un clone de constructeur "suit" une branche particulière lorsque le type du constructeur concorde avec la branche et lorsque les événements du clone et de la branche sont équivalents. Ici encore cette relation d'équivalence est primordiale.

Par conséquent, il ne nous reste plus qu'à décrire l'ensemble $\Phi_{\mathcal{C}}$ et sa relation d'équivalence. Nous devons, pour se faire, analyser les propriétés souhaitées pour cette équivalence.

Une première propriété provient du désir de distinguer deux clones de constructeur produit par deux codes différents du programme initial. Cela est réalisé en distinguant deux clones lorsqu'ils ne portent pas la même étiquette.

Nous désirons aussi pouvoir distinguer deux clones de constructeur lorsqu'ils ont été créés grâce à la présence de deux clones de constructeurs distingués par la première propriété. Nous pouvons reformuler cette exigence en disant que deux clones seront distingués si les environnements d'événements où ils sont créés sont suffisamment différents.

Différentes solutions possibles

Voici quelques exemples de structure d'événements :

- Une première solution consiste à identifier $\Phi_{\mathcal{C}}$ et $\mathcal{L}$. Alors, $CEv(c, l)\phi_{\mathcal{C}} = l$ et l'équivalence $\equiv_{\Phi_{\mathcal{C}}}$ est l'égalité. Cette première solution possède la première propriété mais pas la seconde. Elle n'est pas assez fine.

- Une seconde solution consiste à identifier $\Phi_{\mathcal{C}}$ et $\mathcal{L} \times \Phi$. Malheureusement, cela donne une définition récursive pour Φ et $\Phi_{\mathcal{C}}$ et une telle définition donnerait des structures cycliques (comme pour les valeurs). Le problème d'équivalence serait aussi difficile à résoudre que pour l'égalité entre valeurs.

- Une troisième solution demande une formulation un peu différente. Au lieu d'associer à un clone un élément de $\Phi_{\mathcal{C}}$ issu de la fonction $CEv : \mathcal{C} \times \mathcal{L} \to \Phi \to \Phi_{\mathcal{C}}$ où seuls le type et l'étiquette du constructeur sont donnés, nous pouvons ajouter l'index du clone. Nous aurions une fonction du type $CEv : \mathcal{C} \times \mathcal{I}_{\mathcal{C}} \times \mathcal{L} \to \Phi \to \Phi_{\mathcal{C}}$, plus générale que la précédente. Dans ce cas, nous pourrions distinguer tous les clones entre eux en définissant $\Phi_{\mathcal{C}}$ comme étant $\mathcal{I}_{\mathcal{C}}$ et avec $CEv(c, i, l)\phi = i$. Cette solution est trop fine et déboucherait trop souvent sur une spécialisation ne terminant pas (le nombre de clones de constructeur n'est pas borné).

Événements et événement simples

Pour toutes ces raisons, nous avons choisi de définir $\Phi_{\mathcal{C}}$ en utilisant une notion *d'événement simple*.

Un événement simple $\psi_s \in \Psi_s = \mathcal{C} \times \mathcal{L} \times \mathcal{L}$ est un triplet (c, l_c, l_m) :

- c est un constructeur.
- l_c est l'étiquette où ce clone de constructeur apparaît.
- l_m est l'étiquette d'une structure de contrôle.

Un événement simple est créé lorsque un clone de constructeur active une branche d'une structure de contrôle spécialisée. C'est une version plus simple d'un événement de Ψ entre un clone de constructeur et une structure de contrôle. Nous noterons $\Phi_s = \mathcal{P}(\Psi_s)$ l'ensemble des parties de Ψ_s.

Nous pouvons alors définir $\Phi_{\mathcal{C}}$ comme le produit $\mathcal{L} \times \Phi_s$. De manière informelle, si (l, ϕ_s) décrit les événements associés à un clone de constructeur, l représente l'étiquette associée à ce clone et ϕ_s l'ensemble des événements simples qui ont participé à sa création.

L'équivalence sur $\Phi_{\mathcal{C}}$ est l'égalité des deux composantes (pour la seconde composante, l'inclusion réciproque des ensembles d'événements simples).

Pour terminer, la fonction $CEv : \mathcal{C} \times \mathcal{L} \to \Phi \to \Phi_{\mathcal{C}}$ est définie par :

$$CEv(c, l)(\phi) = (l, SimpleEvents(\phi))$$

et $SimpleEvents : \Phi \to \Phi_s$ par :

$$SimpleEvents(\phi = \psi_1, ..., \psi_n) = \bigcup_{1 \leq i \leq n} \phi_s \cup \{(c, l_c, l_m)\}, \psi_i = \text{Match}(c, (l_c, \phi_s), l_m)$$

Cette définition permet de distinguer deux clones de constructeurs lorsqu'ils ont été créés par deux constructeurs ayant deux étiquettes différentes ou récursivement si ces deux constructeurs ont été créés grâce à deux constructeurs différents. En un sens, cette notion d'événement simple traduit une notion de ressource. Un événement simple provient de la consommation d'un constructeur par le programme initial. Comme, en général, une fois qu'un test a eu lieu, ce test ne sera pas renouvelé (nous connaissons la valeur testée), la notion d'événement simple capture la manière dont les programmes sont construits. En particulier, la notion de récurrence suivant une structure de données convient parfaitement à ce mécanisme car le programme accède petit à petit à la structure. La fonction `append` est un cas typique de ce genre de fonctions. Par extension, cela convient aussi aux programmes définis par une récurrence sur un type de données, mais avec une possibilité de reprendre la récurrence sur une sous-structure lorsque le programme atteint une feuille. Les évaluateurs d'expressions basés sur la sémantique dénotationnelle (par exemple l'interprète TP) sont des exemples de ce type de programmes.

La définition de $\Phi_{\mathcal{C}}$ convient aussi aux programmes copiant fortement les structures de donnée (par exemple avec une fonction `copy` copiant ces structures ou plus vraisemblablement des fonctions remplaçant des feuilles d'une structure arborescente par d'autres arbres (comme la fonction `append`)).

Événements dans l'implantation LaMix

LaMix utilise une notion plus simple événement. En effet, les expériences que nous avons réalisées sur une première version de LaMix basée sur la notion d'événement ci-dessus, tendaient à créer trop de clones. Cela est au détriment de la rapidité du mécanisme de clonage. Nous avons testé

une version plus simple et constaté que les résultats obtenus (les programmes résiduels) étaient les mêmes qu'avec la version complète d'événement.

Dans cette version, nous nous concentrons sur la donnée statique. LaMix se rapproche de la notion de spécialisation de programme au sens où un programme est spécialisé vis-à-vis d'une constante. Dans cette version, seuls les constructeurs provenant de la partie déterministe de la fonction à spécialiser créent un événement. En effet, dans la version complexe, tous les constructeurs portant une étiquette distincte sont différenciés alors que LaMix ne distingue qu'une partie d'entre eux.

Dans cette version, $\Phi_{\mathcal{C}} = \mathcal{P}(\mathcal{L})$. Les étiquettes associées à un clone de constructeur sont celles des clones de constructeur qui ont activé ce clone ou bien l'étiquette de ce constructeur si l'expression où il apparaît n'est pas incluse dans une conditionnelle.

Soit c^i un clone de constructeur correspondant à ℓ : `cons` $c(e)$ avec j comme index d'environnement. Soit $\phi = \psi_1, \ldots, \psi_n = Events^j$, l'environnement d'événements de ce clone. Alors, les événements associés à c^i, $GetCEv^i$, sont :

$$CEv(c,l)(\phi) = \begin{cases} \{l\} & \textbf{si } \forall k, 1 \leq k \leq n, \psi_k = \text{Call}(f^m) \\ \bigcup_{1 \leq k \leq n} \phi_c, \psi_i = \text{Match}(c, \phi_c, l_m) & \end{cases}$$

Chapitre 5

Généralisation du programme cloné

Ce chapitre a pour objet la correction du programme cloné par rapport au programme source. En effet, l'introduction d'objets spécialisés (et en particulier de clones de constructeur) pose un certain nombre de problèmes puisque ces objets ne sont pas forcément compatibles avec les objets initiaux (en particulier, vis-à-vis du typage). Cette phase résout les problèmes suivants :

- *Phase de réévaluation* : Cette phase reprend l'analyse des valeurs alternatives associées aux expressions clonées et introduit des index de spécialisation aux produits.

  En effet, certaines branches de structure de contrôle spécialisée sont inutiles car le code correspondant aux clones de constructeur activant leur motif a disparu lors du clonage. Ceci résulte principalement de la transformation d'un appel de fonction en un appel récursif et de la transformation d'une conditionnelle en conditionnelle résiduelle lorsque la valeur testée passe d'une alternative à $\top$.

  L'introduction des index de spécialisation des produits permettra de les regrouper en *familles*, puis, lors de la phase de *spécialisation* de les transformer de manière uniforme (en éliminant les arguments constants).

- Résoudre les problèmes de typage entre les expressions dont le type doit être celui du programme source (par exemple, les valeurs attachées aux paramètres de la fonction à spécialiser, ou bien la valeur retournée par cette fonction dont le type doit être le même que le type de la fonction initiale). Pour cela, nous introduisons un mécanisme de *généralisation* qui indique quels sont les constructeurs et les produits qui génèrent des valeurs dont le type doit être identique au type de ces objets dans le programme source.

- Ces généralisations imposent que certaines structures de contrôle doivent être *généralisées* car le type de l'expression testée a été généralisé. Ceci impose la *fusion* de branches de structures de contrôle spécialisées correspondant à des clones de constructeur du même constructeur. Cette identification de branches de conditionnelles entraîne la fusion d'expressions puis celle des autres objets spécialisés (en particulier de clones de fonction qui sont regroupés en *classes*) et parfois de nouvelles généralisations...

- Une dernière tache de cette phase de fusion/généralisation consiste à regrouper les clones de constructeurs non généralisés en *classes* (donnant lieu à un nouveau constructeur) et en *familles* (donnant lieu à un nouveau type) et à regrouper les clones de produit en *familles* donnant lieu à un nouveau produit. Ces regroupements seront essentiels

lors de la phase suivante de spécialisation car les transformations sur ces objets devront respecter les classes et les familles.

Le résultat de ces opérations est constitué du programme cloné initial, de l'ensemble des objets qui ont été généralisés (clones de constructeur et de produit), de la description des classes de clones de fonction et de constructeur et de celle des familles de clones de constructeur et de produit.

5.1 Correction du programme cloné

Après l'opération de clonage décrite dans le chapitre précédent, nous obtenons une version clonée du programme initial dont les fonctions ont été dupliquées, les structures de contrôle spécialisées et les objets annotés par un index.

Globalement, le programme obtenu diffère du programme initial par l'ajout des index et la spécialisation des structures de contrôle. L'indexation consiste à créer plusieurs objets là où le programme initial ne les différenciait pas. La spécialisation des structures de contrôle consiste à ne retenir que les branches utiles des conditionnelles.

Le programme résiduel comporte un certain nombres de modifications dont nous devons vérifier la correction. Plutôt que de présenter un résultat de correction nous allons aborder, dans un premier temps, les problèmes qui subsistent et la méthode nous permettant de les éliminer.

5.2 Clones de constructeurs inutiles

Une première constatation sur le programme obtenu après clonage est que les valeurs attachées aux différentes expressions sont parfois trop générales par rapport à ce qu'une analyse statique de la version clonée du programme donnerait. En particulier, il peut subsister des constructeurs ne correspondant à aucun code et par conséquent des branches de structures de contrôle inutiles.

Ce problème provient du mécanisme de clonage qui mélange à la fois un mécanisme d'analyse statique et un mécanisme créant et effaçant des clones de fonctions ou de branches de conditionnelles. En effet, lorsque l'évaluateur partiel rencontre une structure de contrôle précédemment spécialisée mais dont la valeur testée est devenue égale à $\top$, il la transforme en structure de contrôle résiduelle et oublie les branches spécialisées qu'il a construites au cours des phases précédentes. Par conséquent, si dans ces branches certains clones de constructeur avaient été construits, ils apparaîtraient toujours comme valeur possible pour une expression alors que le code qui les avait créé a disparu.

Ce cas de figure est peu courant pour les algorithmes suivant la structure arborescente d'un type. Il se rencontre, par exemple, lorsque deux boucles, dont le contrôle est indépendant, sont imbriquées et que la boucle extérieure est résiduelle. Un exemple caractéristique est la fonction `it_list` qui calcule l'itération d'une fonction sur tous les éléments d'une liste. L'expression `it_list f a [b1; ...; bn]` est équivalente à l'expression `f (... (f (f a b1) b2) ...)`. Lorsque la fonction `f` et l'argument `a` sont connus mais pas la liste, l'évaluateur partiel va, dans un premier temps, spécialiser `f` avec pour premier argument `a` et comme second $\top$. Puis, lors des itérations suivantes, il la spécialise de nouveau avec la valeur retournée par cette première version et $\top$. Cela peut engendrer la généralisation de la structure de contrôle de `f`. Par exemple, avec `append` pour la fonction `f` :

```
let rec append l1 l2 = match l1 with
    []   -> l2
  | h::t -> h::(append t l2) ;;
let rec append_it_list a l = match l with
```

```
    []   -> a
  | h::t -> append_it_list (append a h) t ;;
```

Cette fonction renvoie la concaténation de `a` et des éléments de la liste `l` (qui est une liste de listes). Elle ressemble à la fonction `flat_map` qui aplatit une liste de listes. En fait, si `a` est la liste vide, nous retrouvons exactement cette fonction.

Nous pouvons spécialiser `append_it_list` lorsque la liste `l` est inconnue et `a` est la liste vide. La valeur testée par `append_it_list` n'étant pas connue ($\top$), la structure de contrôle est résiduelle et le mécanisme de clonage va identifier l'appel récursif à `append_it_list` au premier clone de cette fonction. Ainsi, la version spécialisée est identique à la version initiale. Par contre, lorsque le premier appel à `append` est rencontré, la valeur de `l1` est la liste vide. Cela donne une version spécialisée de cette fonction où la structure de contrôle est spécialisée :

```
let rec append_1 l1 l2 = match l1 with
    []   -> l2 ;;
let rec append_it_list_0 a l = rmatch l with
    []   -> a
  | h::t -> append_it_list_0 (append_1 a h) t ;;
```

La valeur retournée par `append_1` est $\top$ car `l2` provient de la liste inconnue correspondant à `l`. La valeur associée à l'argument `a` de `append_it_list_0` est alors la borne supérieure de la liste vide et de $\top$ (l'un provenant de l'appel initial à `append_it_list_0` et l'autre de l'appel récursif). Par conséquent, la nouvelle valeur du paramètre `a` est $\top$.

Le programme est ensuite analysé une nouvelle fois. À ce moment là, l'argument `l1` de `append_1` vaut $\top$ et non plus la liste vide puisqu'il provient de l'argument `a`. La valeur testée par `append_1` n'est donc plus la liste vide mais $\top$ et la structure de contrôle devient résiduelle. Lors de cette opération, la branche spécialisée de `append_1` est effacée.

Dans cette exemple, l'effacement de la branche spécialisée n'a pas de conséquence car elle ne crée pas de nouvelle valeur (aucun constructeur n'y apparaît). Par contre, avec une fonction plus complexe, nous pouvons imaginer que cette branche comporte des clones de constructeur. Ceux-ci pourront alors apparaître comme élément d'une valeur associée à une expression. Ils peuvent engendrer des branches dans d'autres structures de contrôle spécialisées. Après la disparition de cette branche par transformation de la structure de contrôle en structure résiduelle, ces structures de contrôle conserveront les branches correspondant à ces constructeurs alors que le code qui les calcule a disparu du programme.

Le programme obtenu n'est pas incorrect mais comporte éventuellement des branches de structures de contrôle ne pouvant jamais être empruntées.

5.2.1 Réévaluation du programme

Pour éliminer ces branches inutiles, nous pouvons analyser de nouveau le programme cloné en initialisant toute les valeurs associées aux expressions à la valeur $\bot$ (*FArgs*, *FRet* et *CArg*), en réinitialisant l'ensemble des motifs des branches de conditionnelle spécialisé (*Branch*) Puis, le programme est réévalué en n'analysant que les branches des structures de contrôle qui sont activées au cours de l'analyse (données par *Branch*).

Durant cette réévaluation, aucune modification de la structure du programme n'est réalisée (et ne peut l'être). Puis, lorsque le point fixe est atteint, le programme élimine les branches obsolètes (point fixe sur *FArgs*, *FRet*, *CArg* et *Branch*).

5.3 Résultat de la fonction spécialisée

Le second problème de correction est plus important. Il porte sur la valeur retournée par la fonction spécialisée. En effet, le mécanisme de clonage renvoie un programme où les constructeurs portent des index ce qui permet de les distinguer les uns des autres. Cela débouchera sur la définition de nouveaux types correspondant à ces clones de constructeur. Or, pour que le programme obtenu soit correct, la fonction spécialisée doit retourner les mêmes valeurs que la fonction initiale lorsque leurs arguments sont égaux. En particulier, toute valeur retournée ne doit pas comporter de constructeur portant un index, car le type du résultat de la fonction spécialisée doit être identique à celui de la fonction initiale pour que la valeur retournée puisse être *compréhensible* par les programmes utilisant la fonction spécialisée.

Par conséquent, tout constructeur pouvant apparaître dans le résultat de la fonction spécialisée ne doit pas être spécialisé (ne doit pas comporter d'index). Par exemple, considérons de nouveau la spécialisation de `append` lorsque le premier argument est connu (la liste [2;1]) mais pas le second :

```
def append x y = 1:match x with
      Cons(p) -> 2:Cons(fst(p), 3:append (snd(p)) y)
    | Nil(_)  -> y
def f z = 4:append (5:Cons(2, 6:Cons(1, 7:Nil))) z
```

Nous avons vu que le mécanisme de clonage débouchait sur le programme suivant :

```
def append_3 x y = 1:match x with
      Nil_C(_)  -> y
def append_2 x y = 1:match x with
      Cons_B(p) -> 2:Cons_E(fst(p), 3:append_3 (snd(p)) y)
def append_1 x y = 1:match x with
      Cons_A(p) -> 2:Cons_D(fst(p), 3:append_2 (snd(p)) y)
def f_0 z = 4:append_1 (5:Cons_A(2, 6:Cons_B(1, 7:Nil_C))) z
```

Trois clones de `append` ont été créés correspondant aux trois constructeurs de la liste `[2;1]`.

Les clones `append_1` et `append_2` comportent deux clones de constructeur de liste `Cons_D` et `Cons_E`. Or, ces deux constructeurs servent à construire la valeur retournée par la fonction `f_0`. Comme le résultat de cette fonction est une liste, sa valeur doit être construite avec les constructeurs de liste du programme initial et nous devons abandonner les index de ces deux constructeurs.

```
def append_3 x y = 1:match x with
      Nil_C(_)  -> y
def append_2 x y = 1:match x with
      Cons_B(p) -> 2:Cons(fst(p), 3:append_3 (snd(p)) y)
def append_1 x y = 1:match x with
      Cons_A(p) -> 2:Cons(fst(p), 3:append_2 (snd(p)) y)
def f_0 z = 4:append_1 (5:Cons_A(2, 6:Cons_B(1, 7:Nil_C))) z
```

Nous avons envisagé trois méthodes permettant de résoudre le problème de la valeur retournée par la fonction spécialisée.

5.3.1 Modification du type du résultat

La première méthode, la plus simple, consiste à spécifier un certain nombre d'informations sur le résultat de la fonction spécialisée. Nous pouvons donner une description de son type et la

correspondance entre ce type et le type du résultat de la fonction initiale. Cette méthode revient à préciser la relation entre un constructeur spécialisé et le constructeur initial. Ainsi, la description du type résultat comporterait une définition classique d'un type ainsi qu'une fonction associant à chaque constructeur spécialisé son constructeur d'origine. Dans ce cadre, le programme cloné n'a pas besoin d'être modifié.

Cette solution n'est pas entièrement satisfaisante car le mécanisme de clonage n'est plus indépendant de l'utilisation de la fonction spécialisée. Pour utiliser une version spécialisée d'une fonction, nous ne devons pas seulement remplacer la définition de la fonction initiale par celle de la version clonée mais aussi modifier l'utilisation du résultat.

De plus, comme nous le verrons dans les sections suivantes, cela ne résout pas tous les problèmes de correction du programme cloné vis-à-vis du programme initial.

Par conséquent, nous avons abandonné cette voie intéressante car pouvant donner lieu à des programmes plus spécialisés. Nous avons adopté une stratégie moins satisfaisante du point de vue de l'efficacité mais permettant de produire des programmes strictement équivalents aux programmes initiaux.

5.3.2 Copie du résultat

Une seconde méthode consiste à ajouter à la fonction spécialisée, une fonction permettant de traduire les valeurs retournées par cette fonction dans le type initial. Ce mécanisme de traduction revient à ajouter un appel à une fonction de copie de structures de données. Le mécanisme de clonage s'étend naturellement à cette fonction. Avec l'exemple de la spécialisation de `append`, la fonction `copy_list` copie le résultat de la fonction initiale `f` :

```
def copy_list x = 9:match x with
      Cons(p) -> 10:RCons(fst(p), 11:copy_list(snd(p)))
    | Nil(p)  -> RNil(p)
def append x y = 1:match x with
      Cons(p) -> 2:Cons(fst(p), 3:append (snd(p)) y)
    | Nil(_)  -> y
def f z = 8:copy_list(4:append (5:Cons(2, 6:Cons(1, 7:Nil))) z)
```

Les constructeurs `RCons` et `RNil` indique que ces constructeurs ne doivent pas être clonés au cours du mécanisme de clonage.

Après clonage, nous obtenons le programme suivant :

```
def copy_list_6 x = 9:match x with
      Cons(p) -> 10:RCons(fst(p), 11:copy_list_6(snd(p)))
    | Nil(p)  -> RNil(p)
def copy_list_5 x = 9:match x with
      Cons_E(p) -> 10:RCons(fst(p), 11:copy_list_6(snd(p)))
def copy_list_4 x = 9:match x with
      Cons_D(p) -> 10:RCons(fst(p), 11:copy_list_7(snd(p)))
def append_3 x y = 1:match x with
      Nil_C(_)  -> y
def append_2 x y = 1:match x with
      Cons_B(p) -> 2:Cons_E(fst(p), 3:append_3 (snd(p)) y)
def append_1 x y = 1:match x with
      Cons_A(p) -> 2:Cons_D(fst(p), 3:append_2 (snd(p)) y)
def f_0 z = 8:copy_list_4
```

```
(4:append_1 (5:Cons_A(2, 6:Cons_B(1, 7:Nil_C))) z)
```

Cette méthode résout le problème de cohérence entre le résultat du programme cloné et celui du programme original. Toutefois, cette solution exige une phase de copie préjudiciable à l'efficacité du programme résiduel.

La solution consistant à introduire une fonction de copie ne résout pas tous les problèmes de cohérence des programmes clonés, comme nous le verrons par la suite.

Par conséquent, nous n'avons pas retenu cette seconde solution en laissant aux utilisateurs le soin d'ajouter artificiellement une fonction de de copie, comme nous l'avons fait dans l'exemple précédent.

En fait, sur l'exemple de la spécialisation de la fonction `append`, la bonne solution consiste à retirer les index de spécialisation des constructeurs pouvant apparaître dans le résultat de `f_0`. Le programme devrait être :

```
def append_3 x y = 1:match x with
      Nil_C(_)  -> y
def append_2 x y = 1:match x with
      Cons_B(p) -> 2:RCons(fst(p), 3:append_3 (snd(p)) y)
def append_1 x y = 1:match x with
      Cons_A(p) -> 2:RCons(fst(p), 3:append_2 (snd(p)) y)
def f_0 z = 4:append_1 (5:Cons_A(2, 6:Cons_B(1, 7:Nil_C))) z
```

Ici les deux constructeurs de liste `Cons_D` et `Cons_E` ont été remplacés par les constructeurs originaux (préfixés avec la lettre `R`).

5.3.3 Généralisation de constructeurs

Nous avons vu ci-dessus que la meilleure solution pour résoudre ce problème dans le cas de `append` consiste à retirer l'index de spécialisation des constructeurs `Cons_D` et `Cons_E`. Nous appellerons cette opération le *généralisation* d'un clone de constructeur.

Dans ce travail, nous avons retenu cette idée qui consiste à oublier l'index des constructeurs pouvant apparaître dans la valeur retournée par la fonction à spécialiser. La méthode consiste à trouver ces constructeurs puis à les remplacer par un constructeur générique, sans index. Pour trouver ces constructeurs, il suffit d'analyser la valeur associée au résultat de cette fonction, de trouver les clones de constructeur qui la composent et, récursivement, ceux des valeurs associées à l'argument de ces clones. Pour cela, nous pouvons calculer le point fixe de la famille G_k définie pour la fonction à spécialiser f_P^0 par :

$$\begin{array}{lcl} G_0 & = & CClone(FRet^{f_P^0}) \\ G_{k+1} & = & G_k \cup \bigcup_{c^i \in G_k} CClone(CArg^i) \end{array}$$

avec $CClone : \mathcal{V} \rightarrow \mathcal{P}(C^{\mathcal{I}})$ la fonction :

$$\begin{array}{lcl} CClone(\top) & = & \emptyset \\ CClone(\bot) & = & \emptyset \\ CClone(P(v_1, ..., v_n) & = & \bigcup_{i=1}^{n} CClone(v_i) \\ CClone(S(C)) & = & C \end{array}$$

La seconde opération consiste à remplacer dans le programme tous les clones de constructeur par le constructeur initial correspondant. Nous le noterons avec le préfixe `R` dans les programmes CL.

Ainsi, sur l'exemple précédent, le mécanisme de clonage retourne le programme :

```
def append_3 x y = 1:match x with
      Nil_C(_)  -> y
def append_2 x y = 1:match x with
      Cons_B(p) -> 2:Cons_E(fst(p), 3:append_3 (snd(p)) y)
def append_1 x y = 1:match x with
      Cons_A(p) -> 2:Cons_D(fst(p), 3:append_2 (snd(p)) y)
def f_0 z = 4:append_1 (5:Cons_A(2, 6:Cons_B(1, 7:Nil_C))) z
```

Les valeurs associées aux différents clones sont :

$$
\begin{array}{lcl}
CArg(\texttt{Cons\_A}) & = & P(2, S\{\texttt{Cons\_B}\}) \\
CArg(\texttt{Cons\_B}) & = & P(1, S\{\texttt{Nil\_C}\}) \\
CArg(\texttt{Cons\_D}) & = & P(2, S\{\texttt{Cons\_E}\}) \\
CArg(\texttt{Cons\_E}) & = & P(1, \top) \\
FArgs(\texttt{f\_0}) & = & \texttt{z} \mapsto \top \\
FRet(\texttt{f\_0}) & = & S\{\texttt{Cons\_D}\} \\
FArgs(\texttt{append\_1}) & = & \texttt{x} \mapsto S\{\texttt{Cons\_A}\} \\
 & & \texttt{y} \mapsto \top \\
FRet(\texttt{append\_1}) & = & S\{\texttt{Cons\_D}\} \\
FArgs(\texttt{append\_2}) & = & \texttt{x} \mapsto S\{\texttt{Cons\_B}\} \\
 & & \texttt{y} \mapsto \top \\
FRet(\texttt{append\_2}) & = & S\{\texttt{Cons\_E}\} \\
FArgs(\texttt{append\_3}) & = & \texttt{x} \mapsto S\{\texttt{Nil\_C}\} \\
 & & \texttt{y} \mapsto \top \\
FRet(\texttt{append\_3}) & = & \top
\end{array}
$$

Par conséquent, les constructeurs pouvant apparaître dans le résultat de `f_0` sont `Cons_D` et `Cons_E` ainsi que les constantes 1 et 2. Le résultat de la généralisation de ces constructeurs donne le programme :

```
def append_3 x y = 1:match x with
      Nil_C(_)  -> y
def append_2 x y = 1:match x with
      Cons_B(p) -> 2:RCons(fst(p), 3:append_3 (snd(p)) y)
def append_1 x y = 1:match x with
      Cons_A(p) -> 2:RCons(fst(p), 3:append_2 (snd(p)) y)
def f_0 z = 4:append_1 (5:Cons_A(R2, 6:Cons_B(R1, 7:Nil_C))) z
```

Les constantes `R2` et `R1` correspondent aux valeurs originales `2` et `1`.

Cette opération ne permet pas toujours d'obtenir directement un programme correct bien que le programme ci-dessus le soit. En effet, durant la phase de clonage, les clones de constructeur apparaissent dans le corps des fonctions mais aussi comme motif de structures de contrôle spécialisées (en fait, leur projection sur $\mathcal{C} \times \Phi_{\mathcal{C}}$). Cela n'est pas le cas pour l'exemple de `append` car les clones `Cons_D` et `Cons_E` ainsi que les constantes 1 et 2 ne participent pas au contrôle du programme.

Dans le cas où un constructeur pourrait à la fois construire une valeur retournée par la fonction spécialisée et servir comme valeur de test d'une structure de contrôle spécialisée, nous devrions aussi modifier cette structure de contrôle pour que le motif correspondant au clone de constructeur soit, lui aussi transformé en un motif sur le constructeur initial. En somme, la transformation doit aussi enlever l'index de spécialisation des motifs correspondant aux clones de constructeurs à généraliser.

Ce point est pourtant difficile à réaliser car le motif associé à une branche d'une structure de contrôle spécialisée ne correspond pas forcément à un unique clone de constructeur mais éventuellement à plusieurs (provenant de plusieurs clones d'un même constructeur). Une méthode consisterait à dupliquer la branche, une copie pour les éventuels clones de constructeur et une autre pour le constructeur générique. Toutefois ceci ne fonctionne pas toujours car il peut exister plusieurs branches différentes de la structure de contrôle spécialisée comportant un clone de constructeur généralisé. L'un devrait activer une branche et un autre, une autre branche. Or, en généralisant ces clones, nous perdons leur index et par conséquent le moyen de les distinguer.

De plus, lorsque un clone de constructeur a été généralisé, le type qui lui associé est le type du constructeur original. Par conséquent, tout autre constructeur pouvant apparaître au même endroit doit être de ce type et doit lui aussi être généralisé.

Pour toutes ces raisons, nous allons employer une méthode de *fusion/généralisation* d'expressions résolvant ces problèmes de cohérence.

En fait, ce problème de généralisation des constructeurs fait partie d'un problème plus général de cohérence du type des expressions. La section suivante présente ce problème plus général.

5.4 Cohérence du type des expressions

Lorsque nous avons présenté le mécanisme de clonage et en particulier celui du clonage des constructeurs, nous n'avons abordé que la question de la spécialisation des constructeurs. L'utilisation des valeurs qu'ils généraient n'était pas abordée, sinon pour la création de branches de structures de contrôle spécialisées.

Prenons un constructeur apparaissant dans le clone d'une fonction. Les utilisations possibles de la valeur créée par ce constructeur sont les suivantes :

- Ce clone de constructeur apparaît comme valeur de test d'une structure de contrôle spécialisée : il active une branche particulière de la structure tout en étant éliminé par filtrage.

- La valeur où apparaît ce clone est généralisée. Cela arrive lorsque l'analyse statique doit trouver la borne supérieure de plusieurs valeurs dont une est $\top$. Ainsi, le résultat associé à une structure de contrôle comportant plusieurs branches est la borne supérieur des valeurs associées au résultat de toutes les branches. De même, le calcul de l'argument d'un constructeur, des paramètres d'une fonction ou de son résultat peuvent généraliser une valeur. Dans ce cas, une alternative comportant ce constructeur est remplacé par la valeur $\top$ qui peut représenter n'importe quelle valeur.

- Le constructeur apparaît comme partie de la valeur associée au résultat de la fonction à spécialiser.

- Le constructeur apparaît comme argument d'un opérateur (problème qui ne sera pas abordé dans ce travail).

- Le constructeur peut ne jamais être utilisé.

Nous pouvons alors diviser les clones de constructeur en deux classes :

- Ceux qui ne sont jamais utilisés, utilisés uniquement par des structures de contrôle spécialisées ou par des opérateurs spécialisables avec ces constructeurs. Nous les nommerons clones *spécialisables.*

- Les autres clones de constructeur, utilisés comme résultat de la fonction spécialisée, dont la valeur est généralisée ou qui rencontrent une primitive ne pouvant se spécialiser. Ce sont les clones *généralisés*.

Malheureusement, cette division des constructeurs est plus complexe pour un langage fortement typé comme CL que pour un langage non-typé comme Lisp ou Scheme [CR91]. En effet, après avoir calculé cette division des constructeurs, le but va consister à oublier les index des constructeurs généralisés. Le type de ces derniers doit être le type du constructeur initial. Par conséquent, pour des raisons de correction du type des expressions, elles ne peuvent posséder un type provenant de constructeurs spécialisables et de constructeurs généralisés.

Le problème consiste, par conséquent, à diviser les expressions du programme en leur donnant soit un *type spécialisable*, soit un *type généralisé*. Ces derniers correspondent aux types du programme initial.

Les contraintes sur les constructeurs deviennent des contraintes sur le type des expressions :

- Le type d'une expression dont la valeur est $\top$ est généralisé.
- Le type de la fonction spécialisée est généralisé.
- Si une expression a un type généralisé alors toutes les expressions permettant de calculer sa valeur associée ont un type généralisé.

Toutefois, cette description n'est pas suffisamment précise car elle ne tient pas compte des produits. Le type associé à une expression peut être à la fois généralisé et spécialisable pour un type produit. De plus, nous devons aussi calculer le type associé aux constructeurs spécialisables.

Cette dernière tâche consiste à associer à chaque clone de constructeur tous les clones de constructeur qui permettent de décrire ce type spécialisable. Dans une première approximation, le type associé à un constructeur spécialisable est un ensemble de clones de constructeur dont il fait partie et qui représente tous les clones de constructeurs pouvant se trouver au même endroit que ce constructeur. En fait, cet ensemble est la fermeture transitive et symétrique de cette relation. Nous reviendrons sur cette définition ultérieurement.

Pour décrire cette division des constructeurs, la fonction

$$RCons : \mathcal{I}_{\mathcal{C}} \rightarrow bool$$

permet de savoir si un clone de constructeur est spécialisable ($false$) ou généralisé ($true$).

Pour pouvoir déterminer facilement les clones de constructeur qui doivent être généralisés lorsqu'ils rencontrent la valeur $\top$, nous avons modifié le domaine des valeurs associées aux expressions de la manière suivante :

$$\begin{array}{rcll} \mathcal{V} & = & \top(\mathcal{V}) & \\ & | & \bot & \\ & | & P\ (\mathcal{V} \times ... \times \mathcal{V}) & \\ & | & S\ (C) & C \subset \mathcal{C}^{\mathcal{I}} \end{array}$$

Le paramètre de la valeur $\top$ fournit les éventuelles valeurs qui ont été généralisées. La borne supérieure de deux valeurs s'obtient par :

$$\begin{array}{lcl} \top(v_1) \sqcup v_2 & = & \top(v_1 \sqcup v_2) \\ v_1 \sqcup \top(v_2) & = & \top(v_1 \sqcup v_2) \\ \bot \sqcup v & = & v \\ P(x_1, ..., x_n) \sqcup P(y_1, ..., y_n) & = & P(x_1 \sqcup y_1, ..., x_n \sqcup y_n) \\ S(C_1) \sqcup S(C_2) & = & S(C_1 \cup C_2) \end{array}$$

La valeur $\top$ du précédent domaine se définie comme étant $\top(\bot)$.

Ainsi, nous gardons une trace des valeurs qui disparaissent lorsqu'elles rencontrent $\top$. Le calcul des valeurs associées aux expressions du programme cloné se réalise comme au début de ce chapitre. En fait, nous avons modifié cette phase en tenant compte de ce nouveau domaine d'analyse. De plus, au cours de ce mécanisme, nous enregistrons les valeurs associées à toutes les expressions du programme. La fonction

$$Val : \mathcal{I}_{\mathcal{F}\cup\mathcal{B}} \rightarrow \mathcal{L} \rightarrow \mathcal{V}$$

renvoie la valeur associée à l'expression d'un clone de fonction ou d'une branche d'une structure de contrôle spécialisée et portant une certaine étiquette.

5.4.1 Généralisation de clones de constructeur

La méthode permettant de diviser les constructeurs en clones généralisés et clones spécialisables repose sur l'analyse des valeurs associées à toutes les expressions du programme cloné et des généralisations de constructeurs qui en découlent. Le principe général est le suivant :

Soit une expression du programme dont la valeur associée est $v \in \mathcal{V}$:

- $v = \top(v_1)$: la valeur v doit être généralisée.
- $v = \bot$: aucune opération.
- $v = P(v_1, \ldots, v_n)$: analyser les sous-valeurs v_1,...,v_n.
- $v = S(C)$: s'il existe un clone de constructeur déjà généralisé (s'il existe $c^i \in C$ tel que $RCons^i = true$), alors v doit être généralisée. Sinon, le type associé à ces clones de constructeurs doit être calculé. D'après notre première définition, ce type est la réunion des ensembles de clones de constructeur définissant le type des clone de C.

L'opération qui généralise une valeur v est décrite par :

- $v = \top(v_1)$: la valeur v_1 doit être généralisée.
- $v = \bot$: aucune opération.
- $v = P(v_1, \ldots, v_n)$: les valeurs v_1,...,v_n doivent être généralisées.
- $v = S(C)$: Les constructeurs $c^i \in C$ sont généralisés. Cela consiste à modifier $RCons^i$ et à généraliser récursivement la valeur de l'argument de c^i si ce clone n'était pas déjà généralisé. En effet, si un constructeur est généralisé, son type est généralisé et correspond au constructeur du programme initial. Par conséquent, son argument doit aussi avoir un type généralisé et donc être généralisé :
  - $RCons^i \mapsto true$.
  - Généraliser $CArg^i$.

Ce mécanisme de généralisation se poursuit tant que de nouvelles généralisations apparaissent au cours de l'analyse de toutes les expressions du programme cloné.

5.4.2 Fusion d'expressions

Malheureusement, ce mécanisme n'est pas suffisant pour obtenir un programme correct avec la définition des types spécialisables correspondant aux clones de constructeurs.

En effet, le fait de généraliser un constructeur peut avoir des conséquences néfastes sur les structures de contrôle spécialisées. Ce phénomène se comprend facilement. Supposons que le mécanisme décrit ci-dessus généralise un certain nombre de clones de constructeur. Dans ce cas, nous devons oublier l'index de ces clones et en particulier nous ne pouvons plus distinguer deux clones de constructeur différents. Or, cette distinction est à la base de la spécialisation des structures de contrôle. Ainsi, la valeur associée au test d'une structure de contrôle spécialisée peut être d'un type généralisé. Or cette structure a pu créer plusieurs branches différentes pour un même constructeur initial.

Considérons par exemple le programme suivant :

```
def f l = match l with
    Nil    -> Nil
  | Cons c -> Cons((fst c,l),f (snd c))
def g x = f (match x with
    true  -> Cons(1,Nil)
  | false -> Cons(2,Nil))
```

Cette fonction renvoie `[1,[1]]` si `x` est vrai et `[2,[2]]` dans le cas contraire.
Après clonage, nous obtenons le programme suivant :

```
def f_3 l = match l with
    Nil_D    -> Nil_H
def f_2 l = match l with
    Nil_B    -> Nil_F
def f_1 l = match l with
    Cons_A c -> Cons_E((fst c,l),f_2 (snd c))
  | Cons_C c -> Cons_G((fst c,l),f_3 (snd c))
def g_0 x = f_1 (rmatch x with
    true _  -> Cons_A(1,Nil_B)
  | false _ -> Cons_C(2,Nil_D)
```

Les valeurs associées aux objets sont décrites par :

$$
\begin{array}{lcl}
CArg(\texttt{Cons\_A}) & = & P(1, S\{\texttt{Nil\_B}\}) \\
CArg(\texttt{Cons\_C}) & = & P(2, S\{\texttt{Nil\_D}\}) \\
CArg(\texttt{Cons\_E}) & = & P(P(1, S\{\texttt{Cons\_A}\}), S\{\texttt{Nil\_F}\}) \\
CArg(\texttt{Cons\_G}) & = & P(P(2, S\{\texttt{Cons\_C}\}), S\{\texttt{Nil\_H}\}) \\
FArgs(\texttt{g\_0}) & = & \texttt{x} \mapsto \top \\
FRet(\texttt{g\_0}) & = & S\{\texttt{Cons\_E}, \texttt{Cons\_G}\} \\
FArgs(\texttt{f\_1}) & = & \texttt{l} \mapsto S\{\texttt{Cons\_A}, \texttt{Cons\_C}\} \\
FRet(\texttt{f\_1}) & = & S\{\texttt{Cons\_E}, \texttt{Cons\_G}\} \\
FArgs(\texttt{f\_2}) & = & \texttt{l} \mapsto S\{\texttt{Nil\_B}\} \\
FRet(\texttt{f\_2}) & = & S\{\texttt{Nil\_F}\} \\
FArgs(\texttt{f\_3}) & = & \texttt{l} \mapsto S\{\texttt{Nil\_D}\} \\
FRet(\texttt{f\_3}) & = & S\{\texttt{Nil\_H}\}
\end{array}
$$

Par conséquent, comme la valeur de retour associée à la fonction `g_0` doit être généralisée, les constructeurs `Cons_E` et `Cons_G` doivent être généralisés ainsi que les valeurs de leur argument.

Cela généralise aussi `Nil_F` et `Nil_G`. Mais, comme les listes initiales apparaissent aussi comme arguments de ces constructeurs, les quatre constructeurs `Cons_A`, `Nil_B`, `Cons_C` et `Nil_D` sont aussi généralisés. Or, la fonction `f_1` utilise la propriété que nous pouvons distinguer les constructeurs `Cons_A` et `Cons_C` pour activer l'une des deux branches. En les généralisant, nous oublions leurs index et les moyens de les distinguer.

Le programme après avoir généralisé les constructeurs ci-dessus devrait être :

```
def f_3 l = match l with
    RNil    -> RNil
def f_2 l = match l with
    RNil    -> RNil
def f_1 l = match l with
    RCons c -> RCons((fst c,l),f_2 (snd c))
  | RCons c -> RCons((fst c,l),f_3 (snd c))
def g_0 x = f_1 (rmatch x with
    true _  -> RCons(R1,RNil)
  | false _ -> RCons(R2,RNil)
```

Nous voyons que la structure de contrôle de `f_1` comprend deux motifs identiques correspondant aux deux branches de la structure de contrôle spécialisée.

Pour résoudre ce problème, nous avons décidé d'opérer une fusion des branches concernées. Dans l'exemple ci-dessus, cela revient à fusionner les expressions

`RCons c -> RCons((fst c,l),f_2 (snd c))` et

`RCons c -> RCons((fst c,l),f_3 (snd c))`.

Nous voyons que cette contrainte impose que les appels à `f_2` et `f_3` soient aussi fusionnés. Cette seconde contrainte entraîne que le corps du résultat de cette fusion (appelé `f_2_3`) soit :

```
def f_2_3 l = match l with
    RNil    -> RNil
  | RNil    -> RNil
```

Puis, de la même manière, les deux cas ci-dessus sont fusionnés, ce qui donne finalement le programme :

```
def f_2_3 l = match l with
    RNil    -> RNil
def f_1 l = match l with
    RCons c -> RCons((fst c,l),f_2_3 (snd c))
def g_0 x = f_1 (rmatch x with
    true _  -> RCons(R1,RNil)
  | false _ -> RCons(R2,RNil)
```

Ainsi, dans cette phase, il est nécessaire d'utiliser un mécanisme de fusion de clones qui est l'opération inverse du clonage. La correction du programme, pour que la division entre partie spécialisable et partie généralisée soit bien définie, impose que le programme devienne moins spécialisé que le programme cloné.

5.4.3 Analyse de la fusion

Comme nous l'avons vu, nous avons besoin d'un mécanisme de fusion. Dans l'exemple ci-dessus, nous avons dû, dans un premier temps, fusionner deux branches d'une structure de contrôle spécialisée car la valeur de test était d'un type généralisé. Cela a entraîné la fusion de deux clones d'une

même fonction `f_2` et `f_3`, puis la fusion des deux branches de la structure de contrôle de cette dernière.

En fait, la fusion d'expression ne porte que sur des clones d'expression provenant de la même expression du programme original. Il n'est pas question ici de fusionner deux clones de deux fonctions différentes comme `g_0` et `f_0` ou bien deux branches d'une structure de contrôle dont les motifs originaux ne sont pas identiques comme les branches correspondant à `true` et `false` du clone `g_0`.

Ainsi, nous n'avons qu'à nous préoccuper de la fusion d'expressions provenant de la même expression du programme original. Cela impose que deux expressions qui doivent fusionner portent la même étiquette. Par conséquent, la fusion de deux expressions ne portent pas sur les caractéristiques des expressions originales (qui sont identiques) mais sur les fonctions décrivant les spécificités des clones.

Le résultat de la fusion d'expression est un programme où les clones de fonctions et les branches des structures de contrôle sont regroupés en classes d'équivalence. Cela impose aussi que les clones de constructeurs soient regroupés en classe d'équivalence. En somme, les clones des objets du programme initial sont regroupés en classe. Pour chaque type d'objet nous aurons à trouver :

- Pour les clones $f^{i_1}, \ldots, f^{i_n}$ d'une même fonction f du programme original : une partition des index $i_1, \ldots, i_n$ de $\mathcal{I}_\mathcal{F}$. La fonction

$$Class : \mathcal{I}_\mathcal{F} \to \mathcal{P}(\mathcal{I}_\mathcal{F})$$

  associe à chaque clone de fonction la classe à laquelle il appartient.

- Pour les clones $e^{i_1}, \ldots, e^{i_n}$ d'une expression e : une partition des index $i_1, \ldots, i_n$ de $\mathcal{I}_{\mathcal{F} \cup \mathcal{B}}$ donne les clones d'expression fusionnés. Les classes découlent de la fusion de clones de fonctions (fusion de leur corps) et de la fusion de branches de conditionnelles pour un même motif du programme original. Il est inutile de les décrire par une fonction.

- Pour les clones $c^{i_1}, \ldots, c^{i_n}$ d'un constructeur c : une division en constructeurs spécialisables et constructeurs généralisés décrit par la fonction

$$RCons : \mathcal{I}_\mathcal{C} \to bool$$

  De plus, pour les constructeurs spécialisables, une première partition regroupe les clones de constructeurs lorsque le mécanisme de fusion des expressions impose que ces clones soient identifiés ou bien lorsqu'ils doivent emprunter la même branche d'une structure de contrôle. Cette partition donnera la définition d'un constructeur spécialisé pour chaque classe. La seconde partition regroupe ces classes pour former des types spécialisés. La fonction

$$Class : \mathcal{I}_\mathcal{C} \to \mathcal{P}(\mathcal{I}_\mathcal{C})$$

  renvoie la classe associée à un clone de constructeur. La seconde partition regroupe ces classes pour former des types spécialisés. La fonction

$$Family : \mathcal{I}_\mathcal{C} \to \mathcal{P}(\mathcal{I}_\mathcal{C})$$

  renvoie l'ensemble des clones de constructeurs devant appartenir au même type.

- Pour les clones de branches de structures de contrôle spécialisées, les classes de branches seront déduites de la fusion des clones d'expression et des valeurs de test des conditionnelles fusionnées.

Le résultat de cette phase de fusion est constitué du programme cloné et de la description des fonctions $RCons$, $Class$ et $Family$.

La correction du programme impose que les contraintes suivantes soient respectées :

- La fonction $Class$ doit décrire une relation d'équivalence sur les clones de fonction et sur les clones de constructeur spécialisables. La fonction $Family$ doit décrire une relation d'équivalence sur les clones de constructeur compatible avec la relation définie par $Class$. Ainsi, si c^i est un clone de constructeur alors $Class^i \subset Family^i$.

- Le programme ne doit pas mélanger les constructeurs spécialisables et les constructeurs généralisés pour toutes les valeurs associées aux expressions.

- Le programme doit être cohérent vis-à-vis des types décrits par $Family$. Cela signifie que les clones de constructeur formant la valeur associée à une expression (ou à la fusion d'expressions) doivent appartenir à une même famille.

- Le programme ne doit pas imposer de nouvelles fusions de clones de fonctions ou de clones de constructeur ou encore de nouvelles généralisations de clones de constructeurs.

La première condition impose que les fonctions $Class$ et $Family$ soient modifiées de manière cohérente. La deuxième et la troisième reposent sur le problème de typage du programme résultat, du fait que CL est un langage fortement typé (contrairement par exemple à *Scheme* où cette contrainte peu disparaître). Enfin, la quatrième condition provient de la cohérence du programme final qui doit être bien défini et résoudre le problème initial de généralisation des clones de constructeur.

5.4.4 Génération de contraintes

Pour tester si la description donnée par les fonctions $RCons$, $Class$ et $Family$ vérifie les conditions ci-dessus, nous allons définir une méthode permettant de trouver les contraintes sur ces fonctions imposées par le programme qu'elles décrivent. Ces contraintes sont de trois sortes. D'une part, certains clones de constructeur doivent être généralisés car ils apparaissent dans la valeur retournée par la fonction spécialisée ou parce qu'ils sont généralisés en rencontrant la valeur $\top$. D'autre part, des contraintes liées à la fusion d'expressions imposent des contraintes sur $Class$ (fusion de clones de fonction ou de clones de constructeur) et indirectement sur $Family$. Enfin, des contraintes sur la cohérence des classes imposent des contraintes sur $RCons$, $Class$ et $Family$.

Considérons un programme cloné et la donnée des trois fonctions $RCons$, $Class$ et $Family$. La fonction $Class$ décrit une partition des clones de fonctions. La génération des contraintes est décrite par :

- Soit $I = Class^i$ la classe d'un clone de fonctions f^i. Si la définition de f est `def` $f(x_1, \ldots, x_n)$ `=` e , les contraintes engendrées résultent des vérification suivantes :

  - Engendrer les contraintes imposées par les valeurs associées à chaque paramètre de cette fonctions : Pour tout $x \in \{x_1, \ldots, x_n\}$, engendrer les contraintes imposées par la valeur $\bigsqcup_{i \in I} FArgs^i(x)$.
  - De même pour la valeur de retour : engendrer les contraintes imposées par la valeur $\bigsqcup_{i \in \mathcal{I}} FRet^i$.
  - Engendrer les contraintes imposées par la fusion des clones d'expression de corps e avec I comme ensemble d'index d'environnement.

– Soit $e = \ell : \cdots$, une expression et $I = \{i_1, \ldots, i_k\}$, un ensemble d'index d'environnement. Il correspond aux index des différents clones de fonctions et de branches de conditionnelles permettant, à travers la fonction $Index$, de retrouver les différents clones associés à une expression (grâce aux étiquettes). Il impose la fusion des expressions correspondant aux clones de fonction ou de branche de conditionnelle décrits par ces index. Ces index appartiennent à $\mathcal{I}_{\mathcal{F}} \cup \mathcal{I}_{\mathcal{B}}$. La première opération consiste à générer les contraintes sur la valeur

$$\bigsqcup_{i \in I} Val^i(l)$$

qui est la borne supérieure des valeurs retournées par les différents clones d'expressions. Puis, suivant le type d'expression, nous avons les contraintes suivantes :

- ℓ : `var` x : aucune contrainte.
- ℓ : `cons` $c(e)$: les clones de constructeur doivent correspondre au même constructeur final. Ils doivent appartenir à la même classe. Posons

  $$J = \{Index^i(\ell) | i \in I\}$$

  l'ensemble des index des clones de constructeurs de ces expressions. Lorsque ces clones ne sont pas généralisés ($\forall j \in J, \neg RCons$), nous avons les contraintes :

  $$\forall j_1, j_2 \in J \begin{cases} Class^{j_1} = Class^{j_2} \\ Family^{j_1} = Family^{j_2} \end{cases}$$

- ℓ : `call` $f(e_1, \ldots, e_n)$: les clones de fonctions doivent correspondre à la même fonction finale. Soit

  $$J = \{FClone^{Index^i(\ell)} | i \in I\}$$

  l'ensemble des index des clones de fonction appelés. Nous avons les contraintes :

  $$\forall j_1, j_2 \in J \; Class^{j_1} = Class^{j_2}$$

  De plus, nous devons engendrer les contraintes issues des expressions e_1,...,e_n avec I comme ensemble d'index d'environnement.
- ℓ : `let` $(x_1, e_1), \ldots, (x_n, e_n)$ `in` e : engendrer les contraintes issues des expressions $e_1, \ldots, e_n$ et e avec I comme ensemble d'index d'environnement.
- ℓ : `match` e `with` $c_1(x_1)$`->` $e_1 | \cdots | c_n(x_n)$`->` e_n : nous devons engendrer les contraintes issues de e avec I comme ensemble d'index d'environnement. Soit

  $$J = \{Index^i(\ell) | i \in I\}$$

  les index des clones de la structure de contrôle. Soit

  $$v = \bigsqcup_{i \in I} Val^i(label(e))$$

  la valeur de test du résultat de la fusion des conditionnelles. La fusion des clones de structure de contrôle donne trois formes de conditionnelles :

  - S'il existe une structure de contrôle résiduelle, le résultat est une conditionnelle résiduelle. Nous devons généraliser la valeur de test v et fusionner les branches correspondant à un même motif du programme original :

    $$\text{si } \exists j \in J \text{ tel que } RMatch^j$$

alors, nous devons généraliser v. Puis, nous devons engendrer les contraintes issues des n branches de cette conditionnelle. Soit $m, 1 \leq m \leq n$, nous devons trouver les index d'environnement correspondant à la m-ième branche. Soit

$$K_m = \begin{array}{l} \{Branch^j(c_m, \phi_c) | j \in J \wedge \neg RMatch^j \wedge (c_m, \phi_c) \in Cases^j\} \\ \cup \{RBranch^j(c_m) | j \in J \wedge RMatch^j\} \end{array}$$

K_m représente l'ensemble des index des branches des différentes conditionnelles correspondant à c_m. Nous devons engendrer les contraintes imposées par l'expression e_m avec K_m comme ensemble d'index d'environnement.

- S'il n'existe pas de structure de contrôle résiduelle mais que la valeur v a été généralisée, alors la structure de contrôle résultat est spécialisée mais ne comporte que les motifs du programme initial. La valeur v est de la forme $S(C)$ avec

$$\forall c^i \in C \ RCons^i$$

  La structure de contrôle résultat ressemble à celle des conditionnelles résiduelles mais avec éventuellement moins de branches. Par rapport au cas précédent, K_m peut être vide. Nous ne devons retenir que les branches où K_m n'est pas vide.
- Par contre, si aucun des clones de conditionnelle n'est résiduel et si les clones de constructeur de v ne sont pas généralisés, la fusion des structures de contrôle est spécialisée et comporte des motifs spécialisables. Nous devons trouver les branches de cette structure à partir des branches des différents clones de conditionnelles et engendrer les contraintes sur $Class$ nécessaires. En effet, si deux clones de constructeur activent une même branche, ils doivent appartenir à la même classe. La valeur v est de la forme $S(C)$ avec $C \subset \mathcal{C}^{\mathcal{I}}$. La première opération va consister à engendrer les contraintes sur les clones de constructeur. Deux clones doivent appartenir à la même classe lorsque leurs projections sur $\mathcal{C} \times \Phi_{\mathcal{C}}$ sont égales :

$$\forall c_1^{i_1}, c_1^{i_2} \in C \text{ si } c_1 = c_2 \wedge GetCEv^{i_1} \equiv_{\Phi_{\mathcal{C}}} GetCEv^{i_2} \text{ alors } \left\{ \begin{array}{l} Class^{i_1} = Class^{i_2} \\ Family^{i_1} = Family^{i_2} \end{array} \right.$$

  Ensuite, chaque classe de clones de constructeur de C va créer une branche de la structure de contrôle résultat qui proviendra de la fusion de plusieurs branches spécialisées. Soient $U_1, \ldots, U_k$ les classes des éléments de C. Pour $m, 1 \leq m \leq k$, les éléments de U_m sont de la forme $c^i_{l_{U_m}}$ avec $1 \leq l_{U_m} \leq n$. L'ensemble des index d'environnement pour la branche m est

$$K_m = \{Branch^j(c, GetCEv^i) | j \in J \wedge c^i \in U_m \wedge (c, GetCEv^i) \in Cases^j\}$$

  Nous devons engendrer les contraintes imposées par l'expression $e_{l_{U_m}}$ avec K_m comme index d'environnement.
  - $\ell : \pi^n_m(e)$: engendrer les contraintes issues de e avec comme index d'environnement I.
  - $\ell : (e_1, \ldots, e_n)$: engendrer les contraintes issues des expressions $e_1, \ldots, e_n$ avec comme index d'environnement I.

- Pour une valeur v, les contraintes portent sur la division entre valeur spécialisée/généralisée et sur les familles de clones de constructeurs de v. Si, une sous-valeur de v comporte une alternative, nous devons vérifier que si un des clones de constructeurs de cette alternative est généralisé alors tous les autres doivent l'être. Dans le cas contraire, nous devons vérifier que

ces clones sont correctement divisés en familles. La méthode de génération de contraintes est la suivante pour les différents cas de valeur :

- $\top(v)$: les constructeurs de v doivent être généralisés.
- $\bot$: aucune contrainte.
- $P(v_1, \ldots, v_n)$: engendrer les contraintes issues des valeurs $v_1, \ldots, v_n$.
- $S(C)$: S'il existe un clone généralisé, les autres doivent l'être : s'il existe $c^i \in C$ tel que $RCons^i$ alors généraliser la valeur v. Dans le cas contraire, nous devons imposer la contrainte que les clones de constructeurs doivent appartenir à la même famille :

$$\forall c_1^{i_1}, c_2^{i_2} \in C \; Family^{i_1} = Family^{i_2}$$

- Pour engendrer les généralisations des clones de constructeurs apparaissant dans une valeur v, nous avons, suivant le cas de la valeur :
  - $\top(v)$: généraliser les constructeurs de v.
  - $\bot$: aucune généralisation.
  - $P(v_1, \ldots, v_n)$: généraliser les constructeurs des valeurs $v_1, \ldots, v_n$.
  - $S(C)$: généraliser les clones de constructeur de C :

$$\forall j \in C \; RCons^j = true$$

Lorsque cette phase de génération de contraintes ne satisfait pas la description des fonctions $RCons$, $Class$ et $Family$, ces fonctions sont mises à jour. Puis, un nouveau cycle de vérification est lancé.

- Les clones de constructeurs devant être généralisés voient la valeur de $RCons$ associée à ces constructeurs positionnée à $true$.
- Pour une contrainte $Class^{i_1} = Class^{i_2}$, soit $I = Class^{i_1} \cup Class^{i_2}$. La fonction $Class$ est mise à jour de telle manière que $Class^i = I$ pour tout $i \in I$.
- De même, pour une contrainte $Family^{i_1} = Family^{i_2}$ soit $I = Family^{i_1} = Family^{i_2}$. La fonction $Family$ est mise-à-jour de telle manière que $Family^i = I$ pour tout $i \in I$.

Le mécanisme de fusion débute avec la fonction $RCons$ positionnée à $false$ pour tous les clones de constructeur. Les fonctions $Class$ et $Family$ associent à chaque clone de fonction ou de constructeur le singleton constitué par cet unique clone.

5.4.5 Résultat des généralisations et des fusions

Cette phase de fusion/généralisation tente de résoudre les problèmes de correction du programme cloné. Cela consiste à diviser les clones de constructeurs en deux types. Certains clones seront généralisés. Dans cette opération, ils vont perdre ce qui les distingue des autres clones de constructeur provenant d'un même constructeur. Leur type sera le type du constructeur du programme original. Par contre, les clones non généralisés pourront engendrer de nouveaux types de données. Ces clones sont regroupés en classe et ces classes en familles. La contrainte de correction, qui débute par la généralisation des constructeurs pouvant apparaître dans le résultat de la fonction spécialisée ou de ceux qui rencontre la valeur $\top$ (schématiquement, qui rencontre une valeur

issue des arguments inconnus de la fonction spécialisée), engendre un certain nombre de fusions de branches de structures de contrôle et de clones de fonction.

À la fin de ce processus, la fonction $RCons$ détermine les clones de constructeurs généralisés. Pour les autres clones de constructeurs, la fonction $Class$ associe à un clone tous les clones de constructeurs qui doivent être identifiés. La fonction $Family$ fournit l'ensemble de ces classes de clones qui doivent appartenir à un même type spécialisé. Ces deux fonctions permettent de définir de nouveaux types en créant un constructeur pour chaque classe de clones de constructeur et en les regroupant pour former des types spécialisés. Une famille donnera la définition d'un type. Cette information est cruciale pour la phase de simplification décrite dans le chapitre suivant.

5.5 Généralisation des produits

Lors de la phase de spécialisation (ou de simplification), qui sera décrite dans le chapitre suivant, nous aurons besoin d'une notion de produits spécialisables ou généralisés, comme nous avons défini la notion de clones de constructeur spécialisables ou généralisés. L'idée consiste à diviser les produits apparaissant dans le programme cloné en deux ensembles suivant l'utilisation des valeurs qu'ils produisent. De plus, nous aurons besoin de regrouper les produits spécialisables en familles pour déterminer les transformations applicables à une famille de produits.

Nous distinguons les produits qui ne sont jamais utilisés et ceux dont nous sommes sûr qu'ils seront toujours utilisés par les projections du programme cloné, des autres. Comme les clones de constructeurs, la valeur d'un produit peut :

- Ne jamais être utilisée.
- Être projetée sur une de ses composantes par une expression $\ell : \pi^n_m(e)$.
- Être généralisée si elle rencontre une valeur $\top$.
- Apparaître dans la valeur de retour associée à la fonction spécialisée.

Dans le troisième et quatrième cas, le produit doit être généralisé. Dans le cas contraire, il est spécialisable.

Pour déterminer cette division, nous avons modifié le domaine des valeurs pour tenir compte d'une notion de clone de produit. Le domaine d'analyse devient :

$$\begin{array}{rcll} \mathcal{V} & = & \top(\mathcal{V}) & \\ & | & \bot & \\ & | & P\ (I \times \mathcal{V} \times \ldots \times \mathcal{V}) & I \subset \mathcal{I}_\mathcal{P} \\ & | & S\ (C) & C \subset \mathcal{C}^\mathcal{I} \end{array}$$

L'ensemble $\mathcal{I}_\mathcal{P}$ est un ensemble d'index de clone de produit. Pour une expression $\ell : (e_1, \ldots, e_n)$ avec i comme index d'environnement, $Index^i(\ell)$ renvoie l'index du clone de ce produit. Un produit est annoté avec l'ensemble des clones de produit pouvant apparaître à un endroit du programme cloné.

La borne supérieure de deux valeurs s'obtient par :

$$\begin{array}{lcl} \top(v_1) \sqcup v_2 & = & \top(v_1 \sqcup v_2) \\ v_1 \sqcup \top(v_2) & = & \top(v_1 \sqcup v_2) \\ \bot \sqcup v & = & v \\ P(I_1, x_1, ..., x_n) \sqcup P(I_2, y_1, ..., y_n) & = & P(I_1 \cup I_2, x_1 \sqcup y_1, ..., x_n \sqcup y_n) \\ S(C_1) \sqcup S(C_2) & = & S(C_1 \cup C_2) \end{array}$$

La phase de réévaluation des valeurs associées aux objets du programme cloné, décrite au début de ce chapitre, est modifiée pour tenir compte de ce nouveau domaine :

- Lorsque un produit $\ell : (e_1, \ldots, e_n)$ est rencontré pour la première fois, un nouveau clone de produit est créé.
- L'expression $\ell : (e_1, \ldots, e_n)$ avec i comme index d'environnement, renvoie la valeur

$$P(\{Index^i(\ell)\}, v_1, \ldots, v_n)$$

  avec $v_1, \ldots, v_n$ les valeurs associées aux expressions $e_1, \ldots, e_n$.
- L'expression $\ell : \pi_n^m(e)$, lorsque la valeur associée à e est $P(I, v_1, \ldots, v_n)$ renvoie v_m.

Le résultat de cette phase donne, en plus des informations déjà fournies par la version précédente, les index des clones de produit qui ont créé un produit apparaissant dans une valeur.

La phase de généralisation et de fusion est modifiée en conséquence. Tout d'abord, la fonction

$$RProd : \mathcal{I}_\mathcal{P} \rightarrow bool$$

décrit les clones de produit qui doivent être généralisés. La fonction

$$Family : \mathcal{I}_\mathcal{P} \rightarrow \mathcal{P}(\mathcal{I}_\mathcal{P})$$

renvoie la famille associée à un clone de produit.

La génération de contraintes est modifiée par :

- Pour une valeur $v = P(I, v_1, \ldots, v_n)$: s'il existe un clone généralisé, les autres doivent l'être :
  - si $\exists i \in I$ tel que $RProd^i$ alors généraliser la valeur v.
  - Sinon, les clones de produits doivent appartenir à la même famille :

$$\forall i_1, i_2 \in I \; Family^{i_1} = Family^{i_2}$$

- Si la valeur $P(I, v_1, \ldots, v_n)$ doit être généralisée alors les clones de produits doivent l'être :

$$\forall j \in I \; RProd^j = true$$

De cette manière, nous pouvons connaître les produits utilisés uniquement à l'intérieur du programme cloné et ne rencontrant jamais de clone généralisé. Les clones de produit spécialisables sont aussi regroupés en familles ce qui permettra de les simplifier lors de la phase suivante de spécialisation.

5.6 Conclusion

Le but du mécanisme décrit dans ce chapitre est de trouver quelle est l'utilisation des clones de constructeurs et de produits pour les séparer en clones généralisés et en clones spécialisables. Cette généralisation repose sur un critère de correction du programme obtenu vis-à-vis du type associé aux expressions du programme qui peut être soit généralisé, soit spécialisable. Après cette phase, la séparation entre ces clones est correct au sens du typage des expressions.

Cette phase assure aussi que les clones de constructeurs ou de produits spécialisés sont toujours utilisés à l'intérieur du programme cloné. Nous pouvons aussi déterminer quelles sont les structures qui les décomposent.

Cette phase regroupe les clones spécialisables en classes et en familles permettant de créer de nouveaux types et autorisant les transformations de programme du chapitre suivant sur les types spécialisables.

Cette phase peut être encore améliorée en forçant certains clones de constructeurs à appartenir à la même classe. En effet, la méthode de clonage tend à créer de nombreuses versions d'un même constructeur. Par exemple, les constructeurs `true` et `false` peuvent engendrer plusieurs clones et introduire de nombreuses branches spécialisées de structure de contrôle. Cette tendance devrait être étudiée et des solutions trouvées. Ce problème sera illustré dans l'annexe consacré aux exemples concrets de spécialisation avec LaMix.

Le programme obtenu après les phases de clonage, de réévaluation et de fusion/généralisation est un programme CL correct (en ce qui concerne le typage) que l'on peut décrire avec la grammaire suivante :

p	$\rightarrow$	$d_1, \ldots, d_n$	
d	$\rightarrow$	`def` $f^I\ x_1\ \cdots\ x_n$ `=` e	$f \in \mathcal{F}, x_k \in \mathcal{X}, arity(f) = n, I \in \mathcal{P}(\mathcal{I}_\mathcal{F})$
	$\mid$	`rtype` t `=` $c_1(T)$ `|` $\cdots$ `|` $c_n(T)$	$t \in \mathcal{T}, c_k \in \mathcal{C}$
	$\mid$	`type` t^I `=` $c_1^{I_1}(T)$ `|` $\cdots$ `|` $c_n^{I_n}(T)$	$t \in \mathcal{T}, c_k \in \mathcal{C}, I \in \mathcal{P}(\mathcal{I}_\mathcal{C}), I_k \in \mathcal{P}(\mathcal{I}_\mathcal{C})$
T	$\rightarrow$	t	$t \in \mathcal{T}$
	$\mid$	t^I	$t \in \mathcal{T}, I \in \mathcal{P}(\mathcal{I}_\mathcal{C})$
	$\mid$	$T * \cdots * T$	
	$\mid$	$I * T * \cdots * T$	$I \in \mathcal{P}(\mathcal{I})$
e	$\rightarrow$	ℓ `:var` x	$x \in \mathcal{X}$
	$\mid$	ℓ `:rcons` $c(e)$	$c \in \mathcal{C}$
	$\mid$	ℓ `:cons` $c^I(e)$	$c \in \mathcal{C}, I \in \mathcal{P}(\mathcal{I}_\mathcal{C})$
	$\mid$	ℓ `:oper` $o\ e_1\ \cdots\ e_n$	$o \in \mathcal{O}, n = arity(o)$
	$\mid$	ℓ `:call` $f^I\ e_1\ \cdots\ e_n$	$f \in \mathcal{F}, n = arity(f), I \in \mathcal{P}(\mathcal{I}_\mathcal{F})$
	$\mid$	ℓ `:let` $(x_1, e_1), \ldots, (x_n, e_n)$ `in` e	$x_k \in \mathcal{X}$
	$\mid$	ℓ `:rmatch` e `with` $c_1(x_1)$ `->` e_1 `|` $\cdots$ `|` $c_n(x_n)$ `->` e_n	$c_k \in \mathcal{C}, x_k \in \mathcal{X}$
	$\mid$	ℓ `:match` e `with` $c_1^{i_1}(x_1)$ `->` e_1 `|` $\cdots$ `|` $c_n^{i_n}(x_n)$ `->` e_n	$c_k \in \mathcal{C}, i_k \in \mathcal{I}_\mathcal{C}, x_k \in \mathcal{X}$
	$\mid$	$\ell : \Pi_m^n(e)$	$1 \leq m \leq n$
	$\mid$	$\ell : \pi_m^n(I, e)$	$1 \leq m \leq n, I \in \mathcal{P}(\mathcal{I}_\mathcal{P})$
	$\mid$	$\ell : (e_1, \ldots, e_n)$	$n \geq 0$
	$\mid$	$\ell : (I, e_1, \ldots, e_n)$	$n \geq 0, I \in \mathcal{P}(\mathcal{I}_\mathcal{P})$

- $f^I, I \in \mathcal{P}(\mathcal{I}_\mathcal{F})$ correspond à la fusion des clones de f dont les index sont I. Nous avons $\forall i \in I\ Class^i = I$.

- `rtype` t `=` $c_1(T)$ `\|` $\cdots$ `\|` $c_n(T), t \in \mathcal{T}, c_k \in \mathcal{C}$ correspond à la définition du type t provenant du programme source. Il décrit le type des constructeurs généralisés.

- `type` t^I `=` $c_1^{I_1}(T)$ `\|` $\cdots$ `\|` $c_n^{I_n}(T), t \in \mathcal{T}, c_k \in \mathcal{C}, I \in \mathcal{P}(\mathcal{I}_\mathcal{C}), I_k \in \mathcal{P}(\mathcal{I}_\mathcal{C})$ correspond à la définition d'un type spécialisable provenant du type t du programme source. L'ensemble d'index I est la famille des constructeurs de ce type. L'ensemble I_k est la classe des clones de constructeur créant cet élément. Nous avons $\forall i \in \mathcal{I}_k\ Class^i = I_k$ et $\forall i \in \bigcup_{1 \leq k \leq n} I_k\ \ Family^i = I$

- $t^I, t \in \mathcal{T}, I \in \mathcal{P}(\mathcal{I}_\mathcal{C})$ correspond à un type spécialisable. I est la famille des clones de constructeur qui le composent.
- $I * T * \cdots * T, I \in \mathcal{P}(\mathcal{I})$ correspond à un produit spécialisable. I est la famille des produits pouvant construire une valeur de ce type.
- ℓ : `rcons` $c(e)$ est un constructeur généralisé.
- ℓ : `cons` $c^I(e), c \in \mathcal{C}, I \in \mathcal{P}(\mathcal{I}_\mathcal{C})$ est un constructeur spécialisable. I est la classe des clones de constructeur pouvant créer cet élément.
- ℓ : `rmatch` e `with` $\cdots$ correspond à une structure de contrôle résiduelle. Les motifs sont un sous-ensemble des motifs de la conditionnelle d'origine.
- ℓ : `match` e `with` $\cdots$ est une structure de contrôle spécialisée dont les motifs sont construits sur un type spécialisable.
- $\ell : \Pi_m^n(e)$ est une projection généralisée.
- $\ell : \pi_m^n(I, e)$ est une projection spécialisable correspondant à la famille de clones de produit I.
- $\ell : (e_1, \ldots, e_n)$ est un produit généralisé.
- $\ell : (I, e_1, \ldots, e_n)$ est un produit spécialisable. I est la famille des clones de produits créant cette valeur.

En réalité, les types spécialisables `type` t^I ne sont pas construits explicitement car nous pouvons retrouver cette information à partir des fonctions $CArg$ et $PArgs$ et de la description des types originels.

La phase suivante de spécialisation va transformer le programme obtenu et notamment s'occuper des objets *spécialisable* (type, constructeur, conditionnelle spécialisée, produit et projection).

Chapitre 6

Spécialisation du programme cloné

Les chapitres précédents ont exprimé une technique permettant de cloner des fonctions et des constructeurs du programme initial puis de les fusionner/généraliser. Nous avons déterminé pour chaque expression la valeur qui lui est associée. En fait, le programme obtenu, exécuté tel quel, ne serait ni plus ni moins rapide que le programme original. Ce chapitre décrit les transformations de programme qui débouchent sur un programme plus efficace et est, en quelle sorte, l'aboutissement et la justification du travail déjà réalisé par les phases précédentes. Ces transformations sont :

- Les constructeurs spécialisables et les structures de contrôle correspondantes sont simplifiés lorsque la famille et la classe des clones de constructeurs sont identiques. Cela revient à éliminer les constructeurs dont le type est composé de cet unique élément.

- De même, les produits et les projections spécialisables sont simplifiés (voire même éliminés) suivant les arguments constants des familles de clones de produit.

- La description des types associés aux objets spécialisables est généré suivant les principes ci-dessus.

- Les expressions dont la valeur associée est constante sont remplacées par une référence vers une constante. De même, une structure de contrôle dont l'expression testée est constante est remplacée par la branche correspondant à cette constante.

- Les variables et les paramètres inutiles sont éliminés (ceux qui ne sont pas utilisés)

- Certaines fonctions sont dépliées (l'appel à une fonction est remplacée par le corps définissant cette fonction). De même, certaines variables locales sont dépliées (une occurrence d'une variable est remplacée par l'expression qui la définit).

Le résultat est un programme CL (une fois que de nouveaux identificateurs soient produits pour les objets spécialisés).

6.1 Efficacité du programme cloné

Le programme obtenu après la phase de clonage puis celle de fusion/généralisation se comporte de la même manière que le programme initial. En effet, l'opération de clonage de fonction consiste

simplement à dupliquer la définition de la fonction originale. Ainsi, si nous considérons la fonction `append` :

```
let rec append x y = match x with
    h::t -> h::append t y
  | []   -> y ;;
```

L'efficacité est identique à celle de la fonction `append_0` où la fonction `append` est copiée deux fois :

```
let rec append_1 x y = match x with
    h::t -> h::append_0 t y
  | []   -> y ;;
let rec append_0 x y = match x with
    h::t -> h::append_1 t y
  | []   -> y ;;
```

Ces deux définitions sont complètement équivalentes du point de vue de la sémantique dénotationnelle et de la complexité.

L'opération qui consiste à spécialiser les structures de contrôle en éliminant les branches inutiles conserve aussi cette propriété d'équivalence pour toutes les exécutions du programme. En effet, le seul cas où un programme et une version clonée peuvent ne pas être équivalents proviendrait d'une erreur dynamique spécifiant qu'une structure de contrôle teste une valeur ne correspondant à aucun motif (une erreur de *pattern-matching*). Si l'analyse de valeur est correcte, ce cas ne peut pas se produire. L'efficacité reste inchangée si l'on suppose que le temps nécessaire à la sélection de la branche correspondant à une valeur ne dépend ni du nombres de motifs, ni de leur ordre. Dans les programmes CL, cette hypothèse est raisonnable car les motifs ne sont constitués que d'un constructeur. Par conséquent, pour sélectionner une branche, if suffit de tester ce constructeur parmi les constructeurs de ce type (par exemple, grâce à une table associant à un constructeur, l'adresse du code le concernant). Toutefois, cette hypothèse n'est pas vérifiée lorsque le programme compilé parcourt la liste des motifs les uns après les autres en tentant à chaque fois d'unifier les motifs avec la valeur testée. Nous supposerons dans ce travail que cette hypothèse est vraie (ou que la différence induite est négligeable).

L'opération de clonage de constructeur n'apporte, elle aussi, aucune modification de la sémantique ou de l'efficacité si nous analysons le programme après la phase de fusion/généralisation. En effet, cette opération remplace un constructeur du programme initial par une classe de clones de constructeur appartenant à un type spécialisable (une famille de clones de constructeur) similaire au type original. La sémantique du programme est préservée grâce à la phase de fusion/généralisation. La complexité est identique car construire une valeur avec un constructeur du programme original ou bien avec un constructeur spécialisable prend autant de temps (si nous supposons que leur construction ne dépend pas du nombre de constructeur d'un type ou de leur ordre de définition dans la déclaration de ce type). De même, les structures de contrôles spécialisées, avec la même hypothèse que ci-dessus ne sont ni plus, ni moins rapides que celles du programme initial.

En conclusion, le programme initial et le programme obtenu après les phases de clonage et de fusion/généralisation sont équivalents d'un point de vue de la sémantique et de l'efficacité. Nous pourrions ainsi croire que ces phases sont inutiles pour le problème de la spécialisation de programme. En fait, la phase de clonage est très importante car elle détermine, pour chaque clone de fonction et pour chaque expression, les valeurs qui leur sont associées et qui peuvent être différentes pour chaque clone d'une même fonction.

Un mécanisme de spécialisation de programme ne pouvant pas créer plusieurs versions d'une même fonction ne serait pas très puissant. Toutefois, comme les mécanismes décrits dans ce chapitre sont indépendants du clonage, nous pouvons parfaitement appliquer cet algorithme à un programme ne comportant qu'une seule version de chaque fonction. Nous parlons alors d'évaluateurs partiels *monovariants*, par opposition avec les évaluateurs partiels *polyvariants* [Bul84]. La seule condition est que l'analyse des valeurs associées à chaque expression ait eu lieu.

6.2 Transformations utilisées par les évaluateurs partiels

La littérature dans le domaine de l'évaluation partielle utilise essentiellement trois transformations pour améliorer l'efficacité d'un programme [JSS85, JGS93]. La première consiste à remplacer les expressions complètement calculables par leur valeur. La seconde consiste à éliminer les paramètres des fonctions et les variables inutiles et à spécialiser les appels. Enfin, le dépliage de fonctions remplace un appel à une fonction par le corps de cette fonction.

6.2.1 Réductions des expressions constantes

La première transformation consiste à remplacer une expression complètement calculable par sa valeur ou bien par une référence à cette valeur. L'expression disparaît. Considérons par exemple le programme suivant :

```
let rec append x y = match x with
    h::t -> h::append t y
  | []   -> y ;;
let f z = append [1;2] [3;4] ;;
```

L'expression `append [1;2] [3;4]` est calculable. Nous pouvons la réduire et la remplacer par la liste `[1;2;3;4]`. Le programme devient :

```
let f x = [1;2;3;4] ;;
```

Nous pouvons plutôt remplacer cette expression par une référence vers une variable globale. En effet, dans la première version, la liste constante est construite à chaque évaluation de `f` alors qu'avec une référence, cette liste est évaluée qu'une fois :

```
let l = [1;2;3;4] ;;
let f z = l ;;
```

Dans les deux cas, le programme est plus efficace que le programme original car l'opération de concaténation des deux listes `[1;2]` et `[3;4]` a disparu.

Cette transformation s'applique aussi aux structures de contrôle dont l'expression testée est complètement calculable. Dans ce cas, la conditionnelle est remplacée par la branche correspondant à la valeur du test. Par exemple :

```
let rec append x y = match x with
    h::t -> h::append t y
  | []   -> y ;;
let f z = append [] z ;;
```

Le paramètre `x` de `append` est calculable. Sa valeur est la liste vide. Par conséquent, la valeur testée pour la conditionnelle de cette fonction est calculable et nous pouvons la remplacer par la branche `y` correspondant au motif vide :

```
let rec append x y = y ;;
let f z = append [] z ;;
```

6.2.2 Élimination de variables

La seconde transformation consiste à éliminer les paramètres ou les variables locales inutiles. Ainsi, si un paramètre ou une variable locale n'est pas référencé, nous pouvons éliminer ce paramètre ou cette variable et effacer l'expression qui construit sa valeur. Pour un paramètre, ce mécanisme consiste, d'une part, à remplacer un appel vers cette fonction par un appel vers la même fonction mais où l'expression calculant le paramètre éliminable a disparu et, d'autre part, à éliminer le paramètre de la définition de la fonction. Ainsi, sur l'exemple suivant :

```
let rec append x y = match x with
    h::t -> h::append t y
  | []   -> y ;;
let f z = append z [1;2] ;;
```

nous obtenons le programme cloné :

```
let rec append_0 x y = match x with
    h::t -> h::append_1 t y
  | []   -> y ;;
let f_0 z = append_1 z [1;2] ;;
```

La transformation des expressions complètement calculables donne le programme :

```
let c = [1;2] ;;
let rec append_1 x y = match x with
    h::t -> h::append_0 t c
  | []   -> c ;;
let f_0 z = append_1 z c ;;
```

Le paramètre `y` de `append_1` n'est pas utilisé. Nous pouvons l'éliminer. Cela donne le programme :

```
let c = [1;2] ;;
let rec append_1 x = match x with
    h::t -> h::append_1 t
  | []   -> c ;;
let f_0 z = append_1 z ;;
```

Dans le cas des paramètres de fonctions, la transformation porte à la fois sur la définition de cette fonction (élimination d'un paramètre) et sur les appels vers cette fonction (élimination de l'expression qui calcule ce paramètre). Toutefois, dans la plus part des évaluateurs partiels, cette transformation ne porte que sur les paramètres dont la valeur est constante ce qui mélange ce principe d'utilisation du paramètre avec la première règle de transformation des expressions de valeur constante.

Pour le cas des variables locales, deux transformations sont possibles suivant l'utilisation des variables. Si une variable n'est pas référencée dans le corps d'une construction `let ...`, nous pouvons éliminer cette variable ainsi que le code qui la calcule. Ainsi, dans le programme :

```
let f z = let x = [1] in z ;;
```

La variable `x` n'est pas utilisée et nous pouvons l'éliminer. Nous obtenons le programme :

```
let f z = z ;;
```

Lorsque la variable n'apparaît qu'une fois dans chaque branche du corps d'une déclaration de variables locales, nous pouvons remplacer cette référence par l'expression qui définit cette variable. Cette opération s'apparente au dépliage de fonction [BD77, ST84]. Nous l'appellerons *dépliage de variable* [BD91]. Ainsi, dans le programme suivant, la variable `x` n'est utilisée qu'une fois :

```
let f z = let x = [z] in x ;;
```

Nous pouvons remplacer cette variable par le code qui la définit et éliminer la variable locale :

```
let f z = [z] ;;
```

Dans les deux cas ci-dessus, ce mécanisme représente la β-réduction d'un redex $(\lambda \mathtt{x}.e_1)e_2$ que représente la construction `let x =` e_2 `in` e_1. Cette expression se réduit en $e_1[\mathtt{x} \mapsto e_2]$. La restriction sur le nombre d'occurrence de la variable `x` consiste à ne pas effectuer cette β-réduction s'il y a une possibilité de dupliquer le code qui définit cette variable. Ainsi, dans le programme :

```
let f z = let x = [z] in (x,x) ;;
```

La β-réduction donne le programme :

```
let f z = ([z],[z]) ;;
```

La liste `[z]` est construite deux fois alors que dans le programme original elle ne l'est qu'une fois et nous obtenons un programme moins efficace. Ainsi, le dépliage de variables s'appuie sur une *analyse d'occurrences de variable* [BD91].

6.2.3 Dépliage de fonctions

Enfin, la troisième transformation usuelle dans le domaine de l'évaluation partielle consiste à *déplier* [BD77, ST84] la définition des fonctions. Cela consiste à remplacer un appel à une fonction par la définition de cette fonction. Cette opération peut être divisée en deux mécanismes pour résoudre les problèmes de duplication du code déplié [Mog88, BD91]. Le premier consiste à remplacer l'appel à la fonction à déplier par le corps de cette fonction dans laquelle des variables locales correspondant aux paramètres sont insérées. Leur définition reprend les expressions calculant les arguments de la fonction initiale. Ensuite, suivant le principe d'analyse d'occurrences de variable locale, ces variables peuvent être dépliées. Ainsi, dans le programme :

```
let rec append x y = match x with
    h::t -> h::append t y
  | []   -> y ;;
let f z = append [] z ;;
```

Nous obtenons, après clonage, le programme suivant :

```
let rec append_1 x y = match x with
    []   -> y ;;
let f_0 z = append_1 [] z ;;
```

Nous pouvons déplier la fonction `append_1` :

```
let f z =
    let x = [] and y = z in
          match x with
            []   -> y ;;
```

Les variables `x` et `y` correspondent aux paramètres de la fonction `append_0`.
Finalement, après dépliage des variables, nous obtenons le programme :

```
let f z = z
```

En effet, la variable `x` est constante. Par conséquent, la structure de contrôle reçoit une valeur de test constante et elle peut disparaître pour ne conserver que la branche active. De même, la variable `y` n'est utilisée qu'une fois.

La division du dépliage en deux mécanismes, l'un transformant un appel par une définition de variables locales, et le second simplifiant cette définition, se justifie pour la même raison de duplication du code calculant les valeurs des arguments de cette fonction que nous avons mentionner dans le traitement des variables locales. Ce mécanisme est longuement abordé dans [Bon90].

Les méthodes usuelles qui gouvernent le dépliage sont basées sur deux stratégies :

- Un appel à un clone d'une fonction est déplié si et seulement si cette fonction n'est appelée que par cet appel [BD91].
- Tous les appels sont dépliés sauf ceux nécessaires aux appels récursifs. Cette méthode est plus compliquée que la première car des fonctions peuvent être mutuellement définies [Ses88].

6.3 Transformation des valeurs partiellement connues

La littérature sur l'évaluation partielle aborde aussi les mécanismes de transformation de programme pour les expressions dont la valeur est partiellement connue [Mog88, Bon88, Lau91a]. Une paire dont le premier élément est constant mais pas le second ne peut pas être traitée par les principes énoncés ci-dessus. Ce problème s'étend aux valeurs partiellement connues composées de constructeurs. Leur gestion repose sur un autre mécanisme. L'idée consiste à ne conserver que les références vers les éléments dynamiques de cette expression (ceux dont les valeurs ne sont pas connues lors de l'évaluation partielle). Ainsi, avec le programme suivant :

```
let rec append x y = match x with
    h::t -> h::append t y
  | []   -> y ;;
let f z = append [z] [z] ;;
```

Les paramètres `x` et `y` de `append` sont partiellement connus si `z` est dynamique (associé à la valeur $\top$). La phase de clonage donne le programme :

```
let append_0 x y = match x with
```

```
    []   -> y ;;
let append_1 x y = match x with
    h::t -> h::append_0 t y ;;
let f z = append_1 [z] [z] ;;
```

La construction des valeurs partiellement connues est retardée dans le corps de la fonction `f`. Seuls sont données en argument les parties dynamiques qui sont les deux occurrences de la variable `z`. Nous obtenons le programme :

```
let append_0 y_z = [y_z] ;;
let append_1 x_z y_z = x_z::append_0 y_z ;;
let f z = append_1 z z ;;
```

Ici, la construction de la seconde liste `[z]` a été retardée jusque dans le corps de la fonction `append_0`. La première liste a disparu.

Les transformations de programme nécessaires pour traiter les valeurs partiellement connues ne seront pas décrites en détails ici car ces mécanismes ne résolvent pas le problème des valeurs alternatives. En fait, la solution que nous proposons pour transformer les valeurs alternatives s'applique aux valeurs partiellement connues. Un mécanisme spécifique n'est donc pas nécessaire.

6.4 Transformation des valeurs alternatives

L'originalité de ce travail porte sur la notion de valeurs alternatives. Lors de la phase de clonage, chaque expression est associée à une valeur qui n'est pas toujours une valeur partiellement connue mais peut représenter des valeurs construites avec plusieurs clones de constructeurs. Ainsi, le valeur v décrite par :

$$\begin{array}{lcl} v & = & S\{\texttt{Nil\_A}, \texttt{Cons\_B}\} \\ CArg(\texttt{Cons\_B}) & = & P(\top, S\{\texttt{Cons\_C}\}) \\ CArg(\texttt{Cons\_C}) & = & P(\top, S\{\texttt{Nil\_A}, \texttt{Cons\_B}\}) \end{array}$$

représente toutes les listes de longueur paire dont les éléments sont quelconques. La valeur v est construite à partir des clones de constructeur `Nil_A` ou `Cons_B` sans que nous puissions savoir lors de la spécialisation du programme lequel de ces deux constructeurs donnera la valeur réelle. Il est fort probable que les deux cas se présenteront lors de l'exécution du programme spécialisé.

Par conséquent, nous devons trouver des transformations permettant de traiter ces valeurs alternatives. Ici, la distinction entre clone de constructeur généralisé et clone spécialisable est importante. En effet, la conséquence de cette séparation est que les constructeurs dits spécialisables ne sont utilisés que de manière interne par le programme cloné et sans jamais être généralisés. Ainsi, les seules utilisations de ces clones de constructeurs sont les structures de contrôle spécialisables dont les motifs n'ont pas été généralisés. Non seulement nous savons que les valeurs créées par les clones de constructeur spécialisables ne sont utilisées que de manière interne par le programme, mais nous pouvons savoir aussi quelles sont les motifs, et donc les branches, des structures de contrôle correspondant à ces valeurs.

L'information sur l'utilisation des valeurs alternatives est très importante car cela nous permet de transformer à la fois le code qui construit une valeur (un clone de constructeur) et le code qui l'utilise (les structures de contrôle spécialisées). Pour déterminer quelles transformations duales nous allons pouvoir appliquer aux deux types de codes, nous allons analyser les valeurs associées aux expressions de ces deux types de codes.

Considérons une structure de contrôle spécialisée :

$$\begin{array}{l}\texttt{match } e \texttt{ with} \\ \qquad c_1^{I_1}(x_1) \texttt{ -> } e_1 \\ \qquad \ldots \\ \texttt{| } c_n^{I_n} x_n) \texttt{ -> } e_n\end{array}$$

Ici, nous avons explicité chaque branche spécialisée en ajoutant aux motifs du programme initial, la classe des clones de constructeur activant cette branche. Un motif est composé d'un constructeur du programme original $c_i \in \mathcal{C}$ associé à la classe $I_i \in \mathcal{P}(\mathcal{I}_\mathcal{C})$ et de la variable x_i pour récupérer la valeur de l'argument du constructeur.

Trois cas peuvent se produire suivant le nombre de branches de la structure de contrôle :

- Si aucune branche n'existe :

  $$\texttt{match } e \texttt{ with}$$

  La valeur testée peut être $\perp$, la valeur indéfinie, qui signifie que l'expression calculant la valeur de test ne retourne jamais de valeur ou bien ne correspondre à aucun motif du programme original. Nous pouvons remplacer cette structure par l'expression $bottom()$ où $bottom$ est une fonction ne retournant jamais de valeur (ou produisant une erreur lors de son évaluation).

- S'il ne reste qu'une seule branche :

  $$\begin{array}{l}\texttt{match } e \texttt{ with} \\ \qquad c_1^{I_1}(x_1) \texttt{ -> } e_1\end{array}$$

  Nous connaissons à l'avance quel est le code qui va s'exécuter après le calcul de l'expression testée. Dans ce cas, nous aimerions que la structure de contrôle disparaisse puisque le test est inutile. Nous pourrions remplacer la structure par le code correspondant à la seule branche disponible. Toutefois, les structures de contrôle ne servent pas uniquement à déterminer quelle branche doit être exécutée mais aussi à lier la variable x_1 du motif correspondant à la valeur de l'argument du constructeur. Or, pour récupérer cette valeur à partir de la valeur testée, nous devons éliminer ce constructeur ce qui nous oblige à utiliser une structure de contrôle car c'est le seul moyen disponible dans le langage CL. La transformation ci-dessus ne pourrait se mettre en œuvre uniquement dans le cas où la variable x_1 n'est pas utilisée par le corps de la branche restante ce qui limite en pratique son utilisation au cas des valeurs constantes. Par contre, si nous arrivions à ne donner que la valeur correspondant au test mais sans le constructeur, nous pourrions transformer cette structure de contrôle en la définition de variables locales :

  $$\texttt{let } (x_1, e) \texttt{ in } e_1$$

  Ceci est réalisable si, pour tout clone de constructeur pouvant apparaître comme valeur de test de cette structure, nous renvoyons non pas la valeur de son argument avec ce clone mais directement son argument. Cette transformation des clones de constructeur s'appelle *élimination de constructeur* (tag removal) [Lau91a]. Ainsi, en transformant de manière duale les clones de constructeur (qui disparaissent) et les structures de contrôle spécialisées à une seule branche, le mécanisme d'évaluation partielle améliore l'efficacité du programme en éliminant à la fois une opération de création d'un constructeur et un test sur une valeur.

- Avec plus d'une branche : La structure de contrôle doit rester puisque nous avons besoin de tester la valeur pour connaître la branche à exécuter.

Ainsi, les transformations qui portent sur les valeurs alternatives sont, d'une part, l'élimination de constructeurs, qui consiste à remplacer une expression ℓ : `cons` $c(e)$ par e, et, d'autre part, la transformation des structures de contrôle spécialisées ne possédant qu'une branche, en une déclaration de variables locales dont le corps reprend le corps de cette branche.

Toutefois, cette double transformation ne peut pas toujours être réalisée. Supposons en effet que nous devions éliminer un clone de constructeur car il sert de valeur de test d'une structure de contrôle spécialisable. Ce constructeur peut diriger d'autres structures de contrôle spécialisées qui elles peuvent posséder plus d'une branche. D'après le troisième principe énoncé ci-dessus, ce constructeur ne doit pas être éliminé. Dans ce cas, la double transformation ne peut s'appliquer. Par conséquent, nous devons aussi envisager le cas où une structure de contrôle spécialisée ne comportant qu'une branche ne peut pas être simplifiée de cette manière.

En résumé :

- Une classe de clones de constructeur est éliminable si toutes les structures de contrôle spécialisées utilisant ce constructeur comme valeur de sélection, ne comportent qu'une seule branche et sont transformables en définitions locales.

- Une structure de contrôle spécialisée ne comportant qu'une branche est transformable si tous les clones de constructeur pouvant apparaître comme valeur de sélection sont éliminables.

Ce principe n'est pas une définition car les deux critères sont définis de manière mutuelle. Toutefois, nous nous apercevons qu'ils nous donne un moyen de connaître les clones de constructeurs qui ne doivent pas être éliminés et les structures de contrôle qui ne sont pas transformables. Les clones de constructeurs éliminables et les structures de contrôle transformables sont ceux qui satisfont les deux critères ci-dessus. D'un point de vue théorique, cet ensemble de clones de constructeur et de structures de contrôle forment le plus grand ensemble vérifiant les conditions ci-dessus et leur complétion.

En fait, pour des raisons de typage, les critères ci-dessus ne sont pas suffisants pour obtenir un programme correct. Nous devons aussi vérifier que l'élimination des constructeurs n'apporte aucune incohérence entre les types des expressions. En effet, en éliminant un clone de constructeur, nous transformons le type de l'expression correspondant à la création de ce constructeur. Par exemple, en éliminant un constructeur de liste, nous passons du type associé à cette liste à un produit du type des éléments de cette liste et du type de la liste.

Heureusement, la phase de fusion/généralisation nous donne les informations suffisantes pour connaître quels sont les clones de constructeur éliminables et pour nous assurer que les transformations n'introduisent aucune incohérence du types des expressions.

En effet, durant cette phase, nous avons rassemblé les clones de constructeurs en classes (fonction $Class$) et en familles (fonction $Family$). Les classes définissent de nouveaux constructeurs, un par classe. Elles regroupent plusieurs clones de constructeur et traduisent la fusion de ces clones en un seul objet. Les familles rassemblent les classes pour former des types.

La complétion des clones de constructeur en classes et des classes en familles traduit justement la cohérence des types des expressions. Ainsi, le critère sur les structures de contrôle transformables se traduit par un critère sur les classes et les familles des clones de constructeur.

Soit c^i un clone de constructeur spécialisable. Il est éliminable si $Class^i = Family^i$.

Une structure de contrôle spécialisable est éliminable si la famille des clones de constructeur servant de test ne comporte qu'une seule classe.

Au niveau du typage, cela revient à simplifier les types qui ne comportent qu'un seul constructeur et à transformer les structures de contrôle dont le test possède ce type par une définition d'une variable locale.

Nous avons maintenant un critère bien défini pour éliminer les clones de constructeur et transformer les structures de contrôle. Cette condition sur la famille des clones est compatible avec les deux critères. En effet, considérons un clone de constructeur c^i tel que $Class^i = Family^i$. Alors, tous les clones de constructeur pouvant apparaître comme valeur de test d'une structure de contrôle spécialisée où c^i sert lui-même de valeur de test ne peuvent appartenir qu'à la classe $Class^i$ de ce clone c^i. Par conséquent, toutes ces structures de contrôle n'ont qu'une seule branche (après la phase de fusion) correspondant à cette classe et tous les clones de constructeur apparaissant comme valeur de test d'une de ces structures correspondent à la même classe et donc au même constructeur.

6.5 Transformations des produits et des projections

La section précédente décrivait les transformations duales entre clones de constructeur spécialisables et structures de contrôle spécialisées. Nous pouvons de même transformer conjointement les produits et les projections.

Considérons par exemple un produit de deux éléments $\ell : (e_1, e_2)$, apparaissant dans un clone de fonction. Nous pouvons distinguer les cas suivants :

- Si les valeurs associées à e_1 et e_2 sont constantes alors l'expression est constante et nous pouvons remplacer cette expression par une constante (ou une référence vers une valeur globale constante).
- Si seule une de ces deux valeurs est constante, nous pouvons éliminer ce produit pour ne conserver que la composante non-constante. Cette opération fait disparaître le produit comme dans la section précédente un constructeur était éliminé.
- Si les deux composantes ne sont pas constantes, le produit doit rester inchangé.

Les transformations duales sur la projection `fst` qui renvoie la première composante d'un produit de deux éléments donnent, suivant les cas :

- Si la valeur associée à l'argument de cette projection est un produit de deux valeurs dont la première est constante alors l'expression est remplacée par cette constante (ou par une référence vers cette valeur). La projection disparaît.
- Si la seconde composante est constante mais pas la première, le mécanisme de transformation des produits a éliminé ce produit pour ne conserver que la composante non-constante. Par conséquent, nous devons éliminer cette projection et la remplacer par l'expression calculant l'argument de la projection.
- Dans le dernier cas où les deux composantes ne sont pas constantes, la projection reste inchangée.

Dans le cas des produits de plus de deux composantes, la méthode consiste à ne retenir que les composantes non-constantes pour obtenir des produits d'arité inférieure. Le produit disparaît s'il ne reste qu'une seule composante (non-constante). La transformation duale des projections se déduit automatiquement.

Toutefois, comme pour les clones de constructeur, nous devons limiter ces transformations aux clones de produit spécialisable (ceux tel que *RProd* est vrai). En effet, les autres peuvent être généralisés ou rendu comme résultat de la fonction spécialisée. Par contre, la phase de fusion/généralisation

nous assure que les produits spécialisables ne sont utilisés que de manière interne et nous pouvons connaître toutes les projections du programme pouvant les utiliser.

Comme pour les constructeurs, nous ne devons pas travailler sur un clone de produit mais sur les éléments de sa famille. Les transformations sur les clones de produit doivent être identiques pour tous les éléments d'une même classe. Soit i l'index d'un clone de produit spécialisable ($RProd^i$ est faux). Les valeurs v_k associées à chacun des arguments de tous les clones $Family^i$ de la famille de ce produit sont les bornes supérieures des valeurs des arguments des clones :

$$v_k = \bigsqcup_{j \in Family^i} PArgs^j(k)$$

Les arguments dont la valeur est constante sont éliminés (de même que les expressions qui les ont construites). Si toutes ces valeurs sont constantes, l'expression produit disparaît (cas des expressions constantes). S'il ne reste qu'une seule composante non-constante, le produit est remplacé par l'expression calculant cette composante.

Soient une projection $\ell : \pi_n^m(e)$ et v la valeur associée à son argument. Alors :

- Si v vaut $\top(.)$, $\bot$ ou $P(I,\ldots)$ tel que $\exists i \in I | RProd^i$, alors cette projection a été généralisée et doit rester inchangée.

- Dans le cas contraire, tous les produits d'index I sont spécialisables et appartiennent à la même famille. Nous pouvons appliquer la transformation duale de celle sur les produits : soit $v_k, k = 1,\ldots,n$ les valeurs définies comme ci-dessus. Elles représentent les valeurs des composantes des clones de produits associés à la valeur v.

- Si v_m est constante, l'expression est elle-même constante.

- Si toutes les valeurs v_k sont constantes sauf v_m alors les produits ont disparu (remplacés par cette composante) et nous devons remplacer la projection par e.

- Sinon, la projection est remplacée par une autre projection $\pi_{n'}^{m'}$. Nous devons calculer les nouvelles valeurs de n' et m'. Le premier n' correspond aux nombres de valeurs v_k non-constantes et m' à la place de l'argument projeté parmi ces expressions dont la valeur est non-constante.

Chapitre 7

Conclusion et travaux futurs

En conclusion, nous allons résumer les chapitres précédents et présenter les extensions possibles de ce travail.

7.1 Clonage de fonctions

Les deux premiers chapitres qui suivent l'introduction décrivent une notion de clonage de fonctions et plus généralement de clonage d'objets d'un programme. Ce mécanisme élémentaire de transformation de programme consiste à dupliquer les objets en leur ajoutant un index. Ainsi, cloner une fonction d'un programme revient à ajouter un index aux occurrences de l'identificateur de cette fonction. Chaque clone est défini par une expression clonant la définition de la fonction initiale. De même, le clonage des constructeurs ajoute un index aux constructeurs. Les structures de contrôles dont les motifs sont clonés introduisent des clones de branches de conditionnelle.

Nous avons retenu cette notion simple de clonage comme base de notre évaluateur partiel et la seule opération vraiment significative de ce processus. En effet, les transformations de programmes utilisées par les évaluateurs partielles peuvent être décomposées en transformations plus simples. En particulier, la notion de clonage apparaît comme un principe irréductible et crucial.

7.2 Valeurs alternatives

Dans cette optique du clonage, les valeurs alternatives semblent une extension naturelle des domaines utilisés par les évaluateurs partiels actuels. Elles étendent la notion de valeurs partiellement connues en utilisant des grammaires hors-contexte. Ainsi, nous pouvons décrire une valeur provenant de plusieurs sources comme le résultat d'une structure de contrôle dont la valeur testée n'est pas connue. Ces valeurs alternatives peuvent aussi modéliser des listes de longueur paire, des listes dont la longueur n'est pas connue mais dont les éléments le sont, des listes formées de parties connues et d'autres inconnues,...

En particulier, les valeurs alternatives semblent bien adaptées au problème posé par les environnements et les types universelles utilisés par les interpréteurs.

7.3 Notion d'événement

La troisième notion introduite dans ce travail porte sur le mécanisme qui permet de cloner un programme. Après avoir introduit les valeurs alternatives, nous avons montré que les mécanismes

actuels des évaluateurs partiels ne pouvaient appréhender correctement ce nouveau type de valeur. Nous avons dû trouver un nouveau mécanisme de clonage qui repose sur la notion d'événement.

Un événement peut être grossièrement vu comme la consommation d'un clone de constructeur par un clone de structure de contrôle. Cette notion se propage, dans le programme cloné, par une notion d'environnement d'événements (de la même façon qu'un environnement des variables est attaché à l'évaluation d'une expression) et une notion d'événements associés aux clones de constructeur (qui correspond à une spécialisation des constructeurs). Les événements traduisent la volonté de connaître quels sont les objets qui ont activé une expression du programme cloné. Ils dénotent une vision dynamique du processus de clonage par rapport à la vision traditionnelle s'appuyant sur les valeurs des arguments d'un clone pour l'identifier avec les autres clones.

7.4 Correction du clonage

Suite à ce processus de clonage, nous devons nous intéresser à la correction du programme cloné par rapport au programme original. Il est apparu qu'un certain nombre de problèmes se posent qui sont résolus par une phase de fusion/généralisation. De manière opposée au clonage, ce mécanisme fusionne plusieurs objets (par exemple des clones de fonctions) et généralise à leur type original certains constructeurs (par exemple ceux pouvant apparaître dans le résultat de la fonction à spécialiser).

Ce processus regroupe les clones de constructeur en classes et en familles représentant respectivement de nouveaux constructeurs et leurs types. De même, les clones de produit sont regroupés en familles et les clones de fonction en classes.

7.5 Spécialisation du programme cloné

La phase suivante consiste à rassembler toutes les informations des phases précédentes pour créer un programme plus efficace que le programme original. Cela passe par la spécialisation des type spécialisables (résolvant ainsi le problème du domaine universel des interprètes et de leur gestion des environnements), le remplacement d'une expression dont la valeur est constante par une référence vers cette valeur, la simplification des produits (qui ne comportent plus que les composantes non-constantes), l'élimination des expressions, des paramètres de fonctions et des variables locales inutiles et, enfin, le dépliage de fonctions et de définitions de variables locales.

Toutes ces transformations résultent de transformations locales assez indépendantes du clonage mais portent en elles toutes les raisons de cloner les programmes. En effet, cette première opération n'apporte aucune modification de la complexité d'un programme. Mais, elle autorisent les transformations ultérieures améliorant l'efficacité d'un programme.

7.6 Développements

Ce travail s'inscrit dans un objectif plus ambitieux de créer un évaluateur partiel pour un langage fonctionnel à la ML. Plus précisément, nous avons l'objectif de travailler sur CAML [WL93, Ler93], un de ses dialectes. De nombreux points restent à éclaircir pour achever ce travail. Nous devons notamment passer d'un langage du premier ordre à un langage d'ordre supérieur et introduire la notion de polymorphisme. Une seconde voie va consister à traiter les constructions impératives, les effets de bord et les valeurs mutables.

7.6.1 Fonctions d'ordre supérieur

Les valeurs alternatives semblent bien adaptées aux fermetures. En effet, il suffirait d'insérer dans le domaine une construction pour ce type d'objets. Les transformations pouvant être appliquées sont aussi faciles à trouver. Elles consistent, pour une abstraction, à la remplacer par un constructeur d'un type somme. L'argument de ce constructeur représente alors l'environnement local de la fermeture. De manière duale, une application est remplacée par une structure de contrôle sélectionnant une branche grâce au constructeur associé à une abstraction. Le corps de ces branches est constitué du corps des abstractions correspondantes.

Toutefois, ce principe reste délicat à mettre en œuvre pour obtenir un programme toujours plus efficace que le programme original. De plus, cela passe par la création de nouveaux types et nous devons fournir leur déclaration ce qui nécessite de connaître le type des valeurs de l'environnement d'une abstraction.

7.6.2 Polymorphisme

Les problèmes posés par le polymorphisme semblent vraiment difficiles lorsque le langage est d'ordre supérieur. En effet, dans une version du premier ordre telle que CL, nous pensons qu'il serait facile d'étendre le mécanisme d'évaluation partielle car les clones de constructeur pouvant apparaître à un endroit du programme doivent appartenir au même type du programme original. Nous n'avons donc pas de problème d'incompatibilité de deux instances d'un même type dans une expression d'une fonction.

Par contre, avec un langage d'ordre supérieur, nous ne pouvons utiliser la méthode de transformation décrite ci-dessus. Considérons par exemple la fonction `map` qui applique une fonction à tous les éléments d'une liste et qui renvoie la liste des résultats de ces applications. Nous pouvons imaginer que cette fonction est appelée avec une fonction sur les entiers (par exemple le successeur) et aussi avec une fonction sur des listes d'entier (représentants par exemple des ensembles d'entiers). Nous ne pouvons pas utiliser la transformation de la section précédente car, la fonction `map` induite devrait, en même temps, travailler sur des entier *et* sur des listes d'entiers qui ne sont pas compatibles du point de vue des types. Toutefois, une solution consisterait à associer aux clones leur type et à tenir compte de cette valeur dans le clonage (comme dans [Mog89]). Un travail important reste à fournir.

7.6.3 Valeurs mutables

Dans CAML, des types spéciaux appelés *mutables* permettent d'introduire des expressions avec effets de bord et permettent une programmation impérative. Nous avons l'objectif d'introduire ces types dans notre évaluateur partiel et nous pensons que la notion d'événement permettra d'aborder le problème difficile des effets de bord dans la spécialisation de programme de manière plus satisfaisante que dans les évaluateurs partiels actuels (qui gèlent les expressions avec effets de bord [Bon90]).

7.7 Remarques finales

Une mise en œuvre LaMix a été développée. Cette expérience impose un certain nombre de choix dans les algorithmes utilisés. En effet, la présentation des différentes phases laisse une certaine liberté dans l'ordre dans lequel les calculs sont réalisés. D'une manière plus générale, certaines informations qui n'ont pas été présentées ici, permettent d'obtenir un mécanisme plus rapide d'évaluation partielle.

Cette expérience nous a permis de cerner quelques problèmes. Ainsi, la phase de clonage a tendance à créer de nombreuses versions d'un même constructeur. La phase de fusion/généralisation pourrait tenir compte de ce fait pour forcer certains clones de constructeur à fusionner. De plus, l'expérience nous a permis de trouver un domaine d'événements plus simple que celui présenté dans ce travail et qui a donné les mêmes résultats qu'une version plus complexe. La rapidité de LaMix a ainsi pu être grandement améliorée.

Les annexes de ce travail présente LaMix et les expériences réalisées avec ce système.

Nous espérons que ce travail pourra servir à comprendre mieux des mécanismes d'optimisation de programme et que les notions introduites se trouveront être utiles en pratique.

Deuxième partie

Évaluation partielle et réseaux d'interaction

Chapitre 1

Évaluation partielle et réseaux d'interaction

Cette partie décrit une expérience qui a consisté à appliquer le processus d'évaluation partielle aux *réseaux d'interaction* [Laf90, Gir87]. Il résulte de travaux effectués au cours d'un D.E.A. [Bec90] et leur continuation en début de thèse [Bec91b, Bec91a, Bec92].

Comme les principes qui ont débouché sur les notions présentées dans la première partie résultent, en grande partie, de cette tentative, nous avons désiré insérer cette partie plus ancienne.

En effet, la notion de spécialisation de constructeurs et des structures de contrôle ressemble au mécanisme d'abréviation dans les réseaux d'interaction. Cette analogie se retrouve dans [Bec93] sur un langage distribué où les opérations de communication (envoie et réception de message) sont spécialisées suivant la partie connues des messages (un peu comme dans [Mog93]).

La notion d'événement introduite dans la première partie trouve aussi son analogue dans les réseaux d'interaction avec l'application d'une règle d'interaction. Comme une structure de contrôle crée un événement en rencontrant un clone de constructeur, l'application d'une règle dans les réseaux fait interagir deux agents (l'un pouvant être la conditionnelle et l'autre, un constructeur).

De plus, la logique linéaire [Gir87] et les réseaux d'interaction [Laf90] s'intéressant de près à la consommation de ressources, cela nous a conduit à la notion d'environnement d'événements (quelles sont les ressources nécessaires à l'activation d'une expression?) puis à la comparaison entre eux.

Nous avons l'intention de reprendre le travail sur l'évaluation partielle des réseaux d'interaction en introduisant les concepts de valeurs alternatives et d'événement puisqu'ils semblent bien s'adapter aux réseaux. Cela permettra de rendre complètement automatique le mécanisme d'introduction d'abréviation dans les réseaux d'interaction.

Ce chapitre, après une section consacrée à la description des réseaux d'interaction, introduit la notion d'abréviation. Puis, des extensions de ce mécanisme sont présentés pour résoudre certains problèmes spécifiques aux réseaux d'interaction comme celui posé par les opérateurs de duplication et d'effacement. La dernière section conclue et donne quelques perspectives de recherche sur ce domaine.

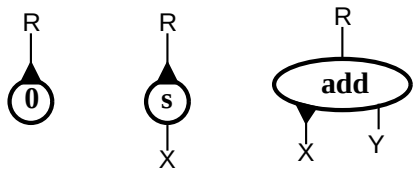


FIG. 1.1 – *Représentation des nombres unaires*

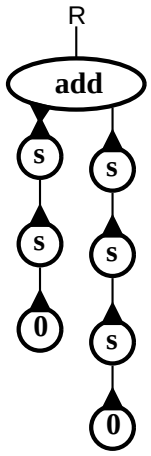


FIG. 1.2 – *Réseau représentant* $2 + 3$

1.1 Les réseaux d'interaction

Les réseaux d'interaction, issus, d'une part de la réécriture et d'autre part de la logique linéaire, se présentent sous la forme de graphes étiquetés couplés à des règles de réécriture. Les articles [Baw86, Laf90, Laf91] présentent ces systèmes en détail.

1.1.1 Les réseaux

Les graphes se décomposent en nœuds et en arêtes appelés respectivement *agents* et *liens*. Chaque agent appartient à une classe le caractérisant. Elle comporte principalement une étiquette, un *port principal* et le nombre de *ports auxiliaires* des agents de cette classe. Un réseau se construit en utilisant des agents et en connectant deux à deux les ports de ceux-ci par des liens. Les ports peuvent éventuellement rester libres. Ils sont alors reliés à une variable libre. Un réseau peut ainsi se voir comme un ensemble de boîtes préfabriquées possédant des bornes reliées deux à deux par des fils.

Les nombres en représentation unaire vont être codés au moyen des deux constructeurs S et 0 comportant respectivement un et aucun port auxiliaire.

L'opérateur d'addition *Add* comporte trois ports, deux pour les arguments et un pour le résultat.

1.1.2 Règles d'interaction

Une fois définies les classes d'agents, nous pouvons introduire des règles de réécriture. Le principe d'interaction considère que seuls deux agents peuvent réagir et qu'ils doivent être reliés par leur

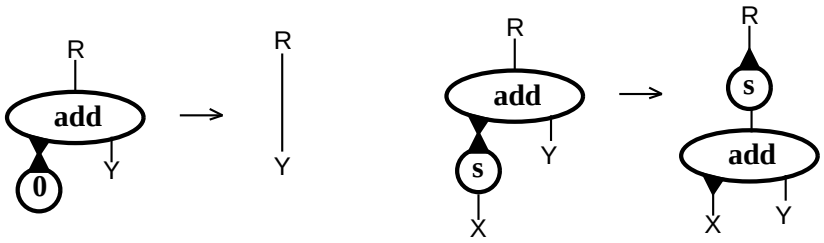


FIG. 1.3 – *Règles d'interaction entre Add, S et* 0

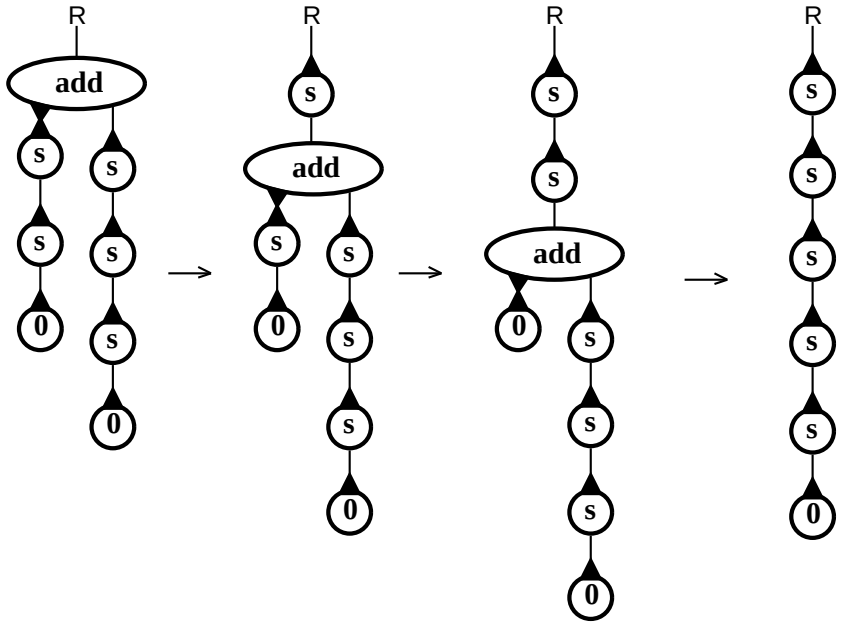


FIG. 1.4 – *Réductions successives de* $2 + 3$

port principal. Ainsi, le membre gauche d'une règle ne comporte que deux agents liés par leur port principal. Le membre droit pourra être tout réseau réduit et possédant les mêmes variables libres que le membre gauche.

Le mécanisme de réduction consiste à substituer, dans un réseau, deux agents par le membre droit de la règle définissant l'interaction entre eux.

A cette définition minimale des réseaux d'interaction viennent se greffer des mécanismes restrictifs. Ainsi, chaque port sera associé à un type. Un lien entre deux ports n'est correct que si le type des deux ports sont complémentaires. Il est en effet stupide qu'une liste puisse se lier à l'opérateur d'addition ou que deux agents S puissent interagir. De plus, comme les réseaux sont des graphes, certains cycles peuvent apparaître. Il est nécessaire de distinguer les "bons" cycles des "mauvais" et de vérifier qu'une règle n'introduise pas de "mauvais" cycle si le réseau de départ n'en comportait pas. Pour plus de précision sur ces aspects, se reporter aux articles d'Yves Lafont [Laf90, Laf91].

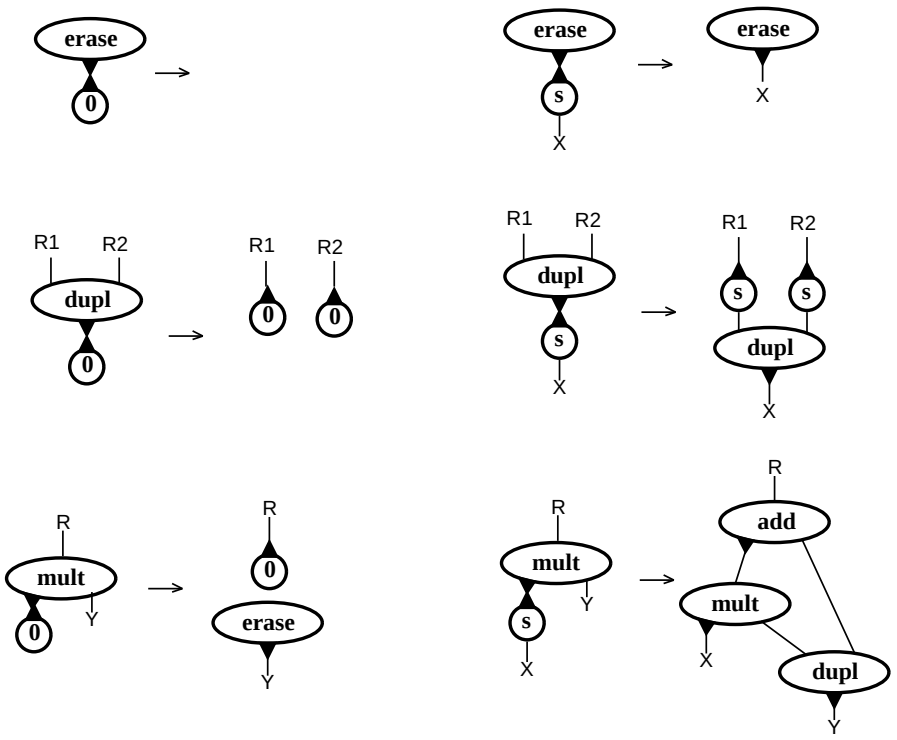


FIG. 1.5 – *Règles pour Mult, Dupl et Erase*

1.1.3 L'arithmétique unaire

Nous avons déjà vu les règles définissant l'addition. Il est à noter que le port principal de *Add* se trouve placé sur un des arguments. La symétrie de cet opérateur est ainsi brisée. Ces règles proviennent directement des définitions mathématiques :

$$S(x) + y = S(x + y)$$

$$0 + y = y$$

La définition de la multiplication est :

$$S(x) * y = y + x * y$$

$$0 * y = 0$$

Si nous regardons ces deux formules, nous nous apercevons que les occurrences de la variable y dans le membre droit et le membre gauche de chaque règle sont différentes. Nous ne pouvons pas les utiliser directement. En fait, y doit être dupliqué dans la première formule et effacé dans la seconde. Comme cela n'est pas automatique dans les réseaux d'interaction, deux nouveaux opérateurs sont introduits, *Dupl* créant deux copies de son argument et *Erase* effaçant le sien.

Ces agents n'ont pas d'équivalent en programmation fonctionnelle car ce mécanisme est implicite (partage et ramasse-miettes).

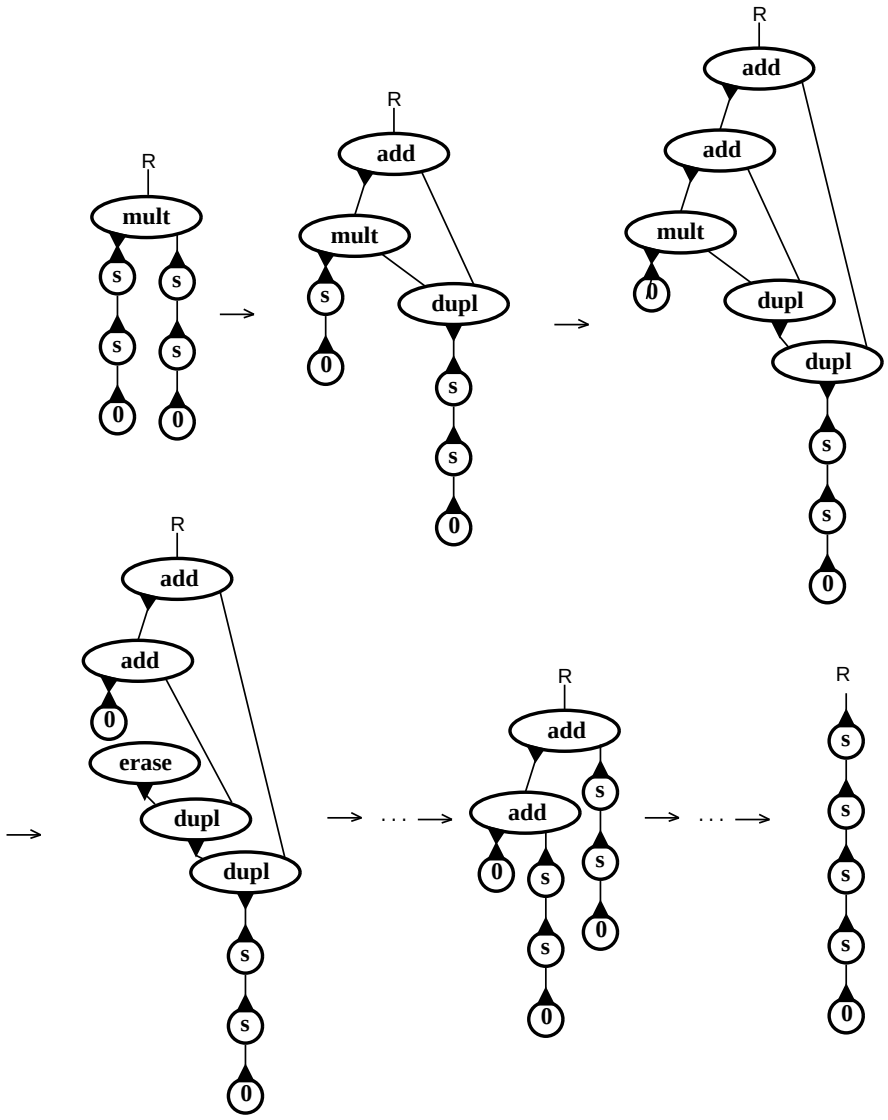


FIG. 1.6 – *Réductions de* $2 * 2$

1.1.4 Propriétés des réseaux d'interaction

Le principal avantage des réseaux par rapport à la réécriture de graphes est la propriété de confluence forte qu'ils vérifient. En fait, l'interaction entre deux agents n'influence en aucune façon les autres couples. Il n'y a jamais de paire critique. Par contre, la terminaison du mécanisme de réduction n'est pas assurée dans le cas général bien que l'on puisse la démontrer pour certains programmes comme l'arithmétique unaire. Le principe d'interaction étant local, les réductions peuvent facilement être exécutées en parallèle.

Contrairement à la programmation classique, rien ne distingue constructeurs et destructeurs. cela pose des problèmes d'interprétation car il n'est pas aisé de distinguer un processus d'une donnée. En particulier, il est difficile d'avoir une notion intuitive de la forme d'un résultat.

1.2 Les abréviations

L'informatique se trouve confrontée à un choix antinomique entre deux conceptions de la programmation. Universalité et efficacité s'affrontent sans que l'on puisse les réconcilier. Un programme général a plus d'applications qu'un programme spécialisé, mais il est plus lent. Pour des raisons de preuves et de temps de conception, la construction des programmes est hiérarchisée. La programmation consiste alors, à chaque génération, à définir un programme comme la mise en relation d'autres programmes déjà définis. Elle n'est plus qu'un jeu d'assemblage. Cet assemblage tend à spécialiser chaque brique qui le compose. Pour résoudre ce dilemme, une solution consiste à évaluer partiellement ces assemblages en tenant compte des liens qui existent entre les différentes composantes. Par exemple, si nous avons besoin d'une fonction calculant le successeur d'un entier, nous pouvons soit créer un opérateur spécifique soit la définir comme l'addition d'un nombre et de 1. Le fait de lier l'addition et le nombre 1, spécialise ces deux entités.

Si nous essayons d'appliquer cette idée aux réseaux d'interaction, nous nous trouvons devant le premier problème de déterminer ce qu'est un programme. En effet, l'exécution d'un processus consiste à réduire un réseau au moyen de règles d'interaction. L'approche classique considère un processus comme la donnée d'un programme et d'un ou plusieurs arguments. Cela consiste dans les réseaux d'interaction à lier un réseau correspondant au programme à d'autres réseaux reflétant les arguments. L'évaluation partielle porte alors sur le premier réseau. Une seconde approche considère que le programme est confondu avec les règles ou un sous-ensemble de règles et qu'un processus lié à ce programme est toute réduction d'un réseau quelconque construit avec les agents définis par ces règles.

Nous considérerons ici que, en partant de règles de base (par exemple, celles de l'addition et de la multiplication), nous créons des boîtes par identification avec un réseau de la génération précédente. Ainsi, les définitions seront hiérarchisées. L'opération consistant à substituer une boîte par le réseau le définissant sera appelée *développement* ou *expansion* et le mécanisme inverse *abréviation*. L'évaluation normale d'un réseau comportant ces entités consiste, dans une première étape, à développer récursivement les abréviations puis, lorsque nous avons obtenu un réseau construit avec les agents de base, à le réduire. Nous pouvons alors interpréter le résultat en l'abrégeant.

Alors que le mécanisme d'expansion ne pose pas de problème, la traduction inverse (abréviation) n'est pas, en général, confluante. Ainsi, la dernière phase du mécanisme d'évaluation reste assez floue. Nous supposerons que nous avons défini une relation entre un réseau composé des agents de base et un réseau d'abréviation. En fait, les définitions engendrent une relation d'équivalence entre réseaux dont les classes sont finies si le membre droit des définitions comporte au moins un agent ou une boîte.

Le mécanisme d'abréviations consiste à ajouter de nouvelles règles pour les boîtes introduites.

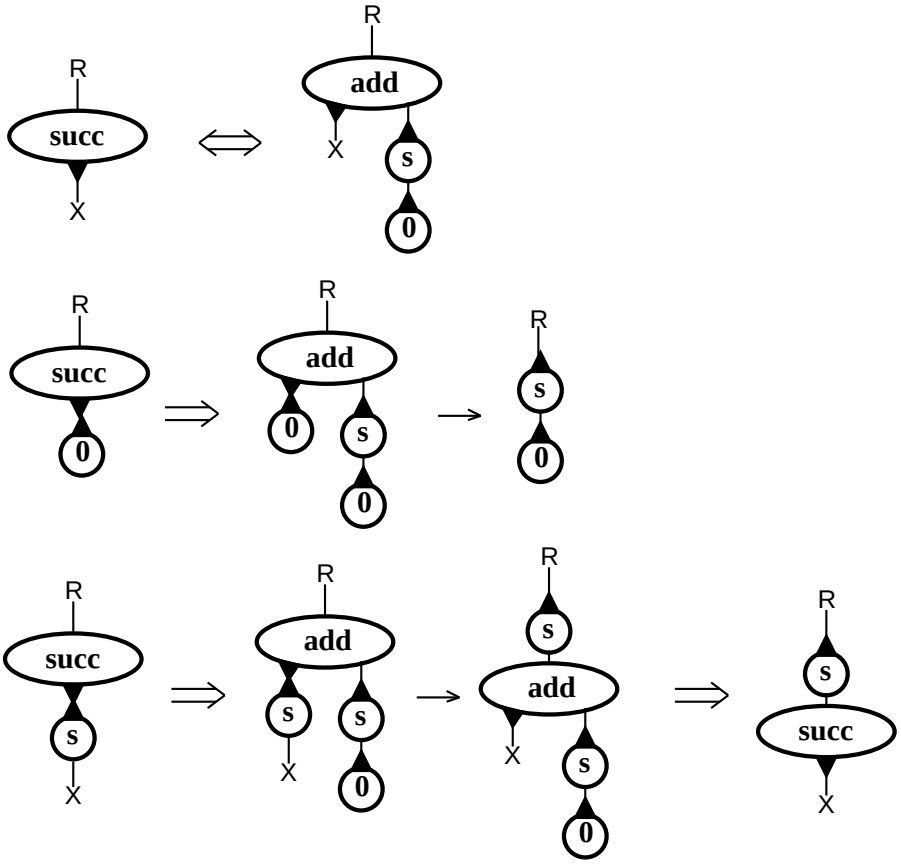


FIG. 1.7 – *Calcule des règles de Succ*

Comme les règles d'interaction reposent sur les agents, nous devons identifier les boîtes à des agents. Ceci introduit des contraintes assez sévères sur le type de réseau que l'on peut abréger. Ces réseaux devront être connexes, comporter au moins un agent, ne posséder qu'un agent dont le port principal est relié à une variable libre, et être irréductible.

Chaque définition engendre un nouvel agent qui va interagir avec les agents qui interagissaient avec l'agent principal de cette abréviation (qui est l'agent du réseau développé ayant son port principal libre). Pour chaque couple, nous devons introduire une nouvelle règle, donc calculer son membre droit. Il est le résultat de la réduction du réseau composé du réseau développé et de l'agent interagissant avec l'abréviation, par les anciennes règles. Ainsi, si nous définissons `Succ=Add(S 0,Res)`, l'agent *Succ* va interagir avec S et 0. Cela débouchera sur deux règles définissant l'interaction de *Succ* avec S et 0.

Le membre droit de la règle entre *Succ* et S peut s'abréger. Une deuxième étape de l'évaluation partielle va tenter d'abréger les membres droits des nouvelles règles. En fait, cette opération peut être réalisée sur les membres droits de toutes les règles et avec toutes les abréviations déjà définies. Comme cette étape comporte des ambiguïtés, nous avons été obligé de faire un choix qui peut ne pas être optimal dans tous les cas.

Ce mécanisme d'évaluation partielle s'apparente au pliage/dépliage introduit par Sato et Tamaki [ST84]. L'efficacité du système d'abréviations porte, d'une part sur l'opération de traduction préliminaire (développements des boîtes devenus inutiles) et d'autre part sur le nombre de réductions nécessaires pour aboutir à un réseau irréductible.

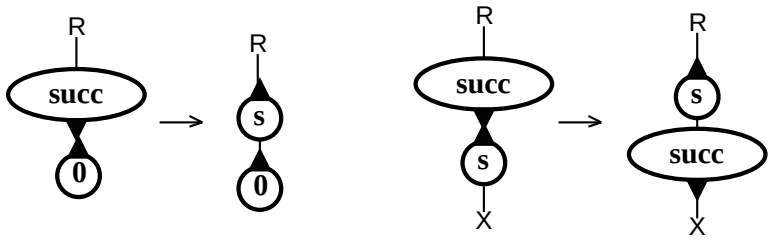


FIG. 1.8 – *Règles de Succ*

1.2.1 Confluence et terminaison

Nous avons vu que les définitions engendrent une relation d'équivalence sur les réseaux dont les classes sont finies. Les éléments d'une classe ont la même interprétation sémantique. Malheureusement, les classes d'équivalence ne sont pas conservées par réduction. La confluence n'est pas assurée modulo cette relation d'équivalence. Ainsi, deux réseaux équivalents peuvent après une étape de réduction ne pas confluer vers deux réseaux équivalents. La figure 1.9 en est un exemple.

Toutefois, si les deux réseaux équivalents convergent vers deux réseaux irréductibles, alors ces derniers appartiennent à la même classe. Cette propriété résulte du caractère hiérarchique des définitions. En effet, chaque réseau peut être développé en un réseau ne comportant que des agents de base (comme cela était réalisé pour exécuter de façon normale les réseaux comportant des boîtes). Nous pouvons démontrer qu'à chaque réduction d'un réseau, correspond un nombre fini et non nul de réductions du réseau développé et que si un réseau est réductible alors tous les réseaux appartenant à la même classe le sont. Ainsi, la confluence modulo les classes d'équivalence est assurée pour les réseaux convergents. De plus, cette équivalence conserve la propriété de terminaison.

1.3 Équivalence de comportement

Les abréviations ne sont pas toujours suffisantes pour satisfaire vraiment le programmeur. Ainsi, certains agents se comportent comme l'identité et sont donc superflus. D'autres agents réalisent le même calcul. Il serait intéressant de les identifier. De plus, certaines propriétés algébriques sont parfois utiles. Ainsi, nous pourrions considérer que l'addition est commutative et associative.

Nous discuterons, dans ce chapitre, d'une technique permettant d'identifier deux agents et, d'une manière plus générale, des classes qui étaient distinctes pour la relation d'équivalence induite par les définitions d'abréviations.

1.3.1 Cas de l'identité

Le premier exemple d'élimination d'agent porte sur les opérateurs se comportant comme l'identité. Dans cette optique, nous pouvons introduire l'abréviation spéciale I pour un unique lien.

Le calcul des règles de I est plus compliqué que pour les abréviations classiques. En effet, dans le chapitre précédent, nous pouvions connaître les agents interagissant avec elle en cherchant ceux qui interagissaient avec l'agent principal du réseau développé. Ici, seul le type des liens est significatif. Un second problème porte sur le membre droit des règles. Comme tout lien peut s'abréger en le remplaçant par I, nous pouvons dans le membre droit de toute règle (s'il n'est pas vide) rajouter autant de I que l'on désire. Les classes d'équivalence que ces définitions engendrent deviennent infinies. Malgré cette difficulté, le système composé des définitions ordinaires et de cette opérateur

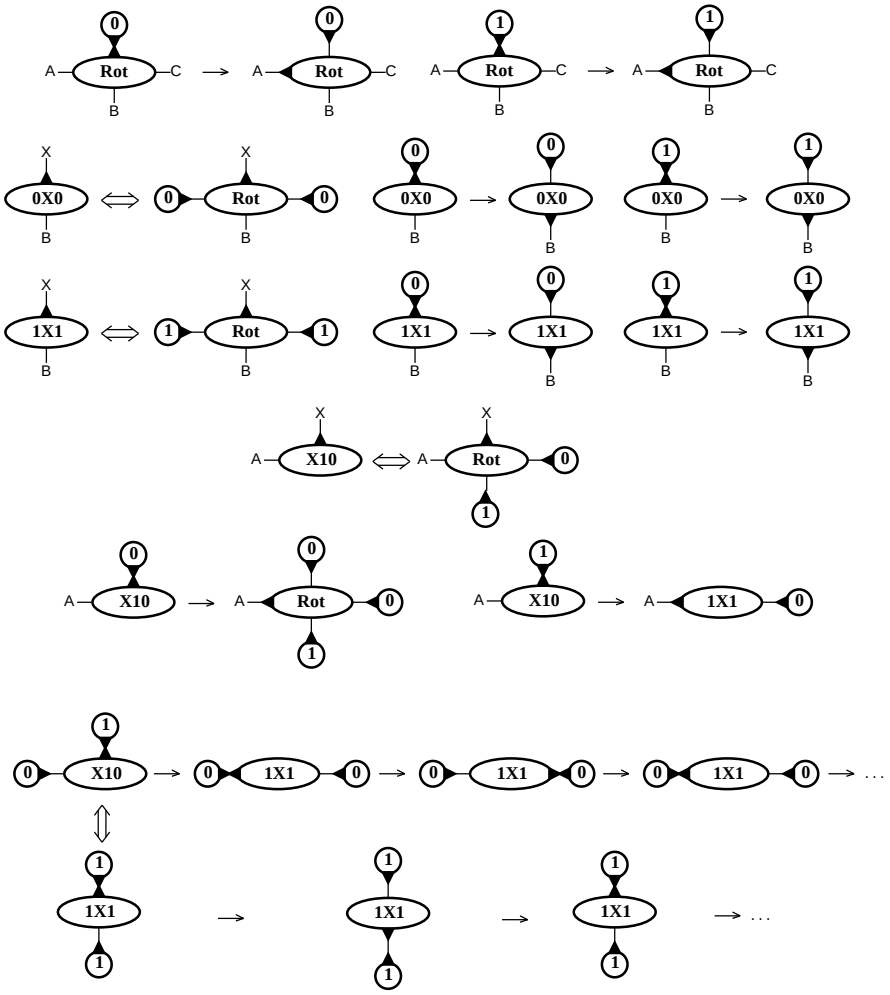


FIG. 1.9 – *Exemple de non-confluence modulo les définitions*

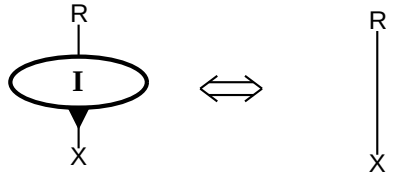


FIG. 1.10 – *Abréviation spéciale I*

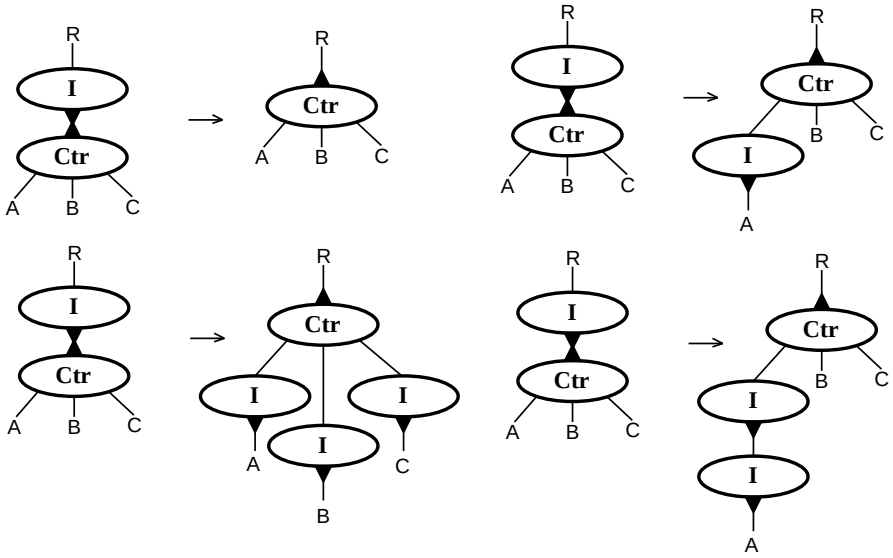


FIG. 1.11 – *Exemples de règles pour I*

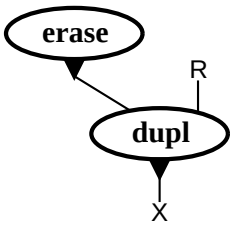


FIG. 1.12 – *Exemple de réseau identité*

a les mêmes propriétés qu'au chapitre précédent (Cela signifie que I n'empêche pas deux agents d'interagir s'ils le faisaient dans le réseau initial).

Une fois les règles fixées, nous pouvons essayer d'identifier un agent avec I. Les agents qui disparaissent après avoir interagi, ceux qui descendent l'arborescence des constructeurs, ceux ayant un comportement encore un peu plus compliqué ou mixte se comportent comme l'identité.

Ainsi, le couple d'agent $Dupl/Erase$ est inutile. Il apparaît, par exemple, lors d'une multiplication ou toutes les fois qu'un argument est dupliqué mais que l'une des copie n'est pas utilisée. Si nous pouvions l'identifier avec I, nous pourrions le remplacer par l'identité puis le développer, ce qui le ferait disparaître.

1.3.2 Fusion de deux agents

Le mécanisme d'équivalence de comportement consiste à identifier le membre droit des règles des deux opérateurs à identifier. Pour que cette opération reste correcte du point de vu sémantique, nous ne devons identifier que des destructeurs qui doivent disparaître au cours de la réduction. De plus, la convergence n'aura effectivement lieu que si tous les arguments d'un processus sont donnés. Ici, les deux agents seront considérés comme semblables s'ils représentent un processus ayant des arguments et rendant un ou plusieurs résultats tout en disparaissant au cours du calcul.

Dans un premier temps, nous ne considérerons qu'une seule identification. Deux agents seront

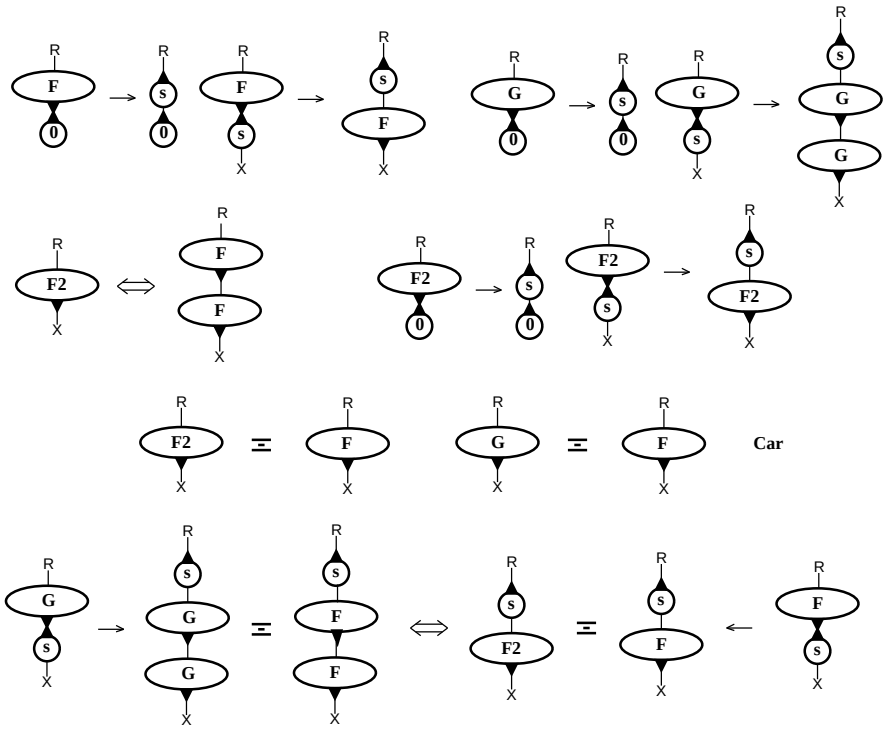


FIG. 1.13 – *Exemple d'équivalence incorrecte*

considérés comme équivalents si pour chaque règle d'interaction avec un même troisième agent, les membres droits des deux règles sont équivalentes. Le problème consiste à définir ce que sont deux réseaux équivalents. Une première approche déclarera que deux réseaux sont équivalents s'ils le sont vis-à-vis de la relation d'équivalence engendrée par les définitions et les identifications déjà introduites ainsi que la nouvelle. Malheureusement, cette théorie est trop générale et engendre des incohérences. Ainsi, un réseau convergeant peut être équivalent à un réseau qui boucle indéfiniment.

La figure 1.13 décrit un exemple d'équivalence abusive. F est la fonction constante égale à 1, G boucle pour tout argument supérieur à zéro. On démontre, dans un premier temps, que F et $F \circ F$ donnent le même résultat en introduisant l'abréviation `F2 = F.F`. Il est alors aisé de démontrer que F et G sont équivalents bien que le premier donne 1 quel que soit son argument alors que le second boucle.

Pour remédier à ce problème, des contraintes sur l'équivalence doivent être introduites. Une solution consiste à n'utiliser que l'équivalence engendrée par la nouvelle identification. Dans ce cas, on retrouve la propriété de terminaison des abréviations pures. D'autres voies restent à explorer dans cette théorie et notamment savoir, dans le cas général, lorsque deux réseaux équivalents convergent vers des réseaux irréductibles, si les deux réseaux irréductibles trouvés sont équivalents.

1.4 Les abréviations généralisées

Nous avons vu que les réseaux pouvant être abrégés devaient posséder certaines propriétés. Ainsi, le réseau développé devait être irréductible, connexe, comporter au moins un agent et ne posséder qu'un port principal libre. Nous allons discuter ici des possibilités d'extension de cette théorie si on abandonne ces contraintes. Ces conditions sont importantes pour diverses raisons dont

la principale repose sur le fait que le réseau développé et l'agent l'abrégeant doivent interagir de la même manière lorsqu'ils sont placés dans le même environnement.

- Ainsi, les réseaux réductibles ne peuvent s'abréger car un agent tout seul ne peut se modifier lui même. Cela remet en cause le principe d'interaction.
- De même, si un réseau comporte un port principal libre qui ne correspond pas au port principal de l'abréviation, il pourra réagir si on lie ce port au port principal d'un autre agent alors que l'abréviation ne le pourra pas.
- Comme on peut démontrer que chaque réseau connexe, comportant au moins un agent et sans cycle vicieux possède au moins un port principal libre, cela élimine les abréviations non-connexes.
- Le dernier cas à traiter porte sur les réseaux constitués d'un unique lien reliant deux variables. Ce réseau se comporte comme l'identité. Le chapitre précédent a déjà abordé cette question.

Ainsi, si nous abandonnons provisoirement ce dernier problème, il apparaît que les contraintes sur les réseaux abrégeables sont liées au fait qu'un réseau peut interagir mais pas l'agent l'abrégeant. Cela était nécessaire lorsque nous nous intéressions à tous les réseaux pouvant être construit avec les agents de base et les abréviations. Toutefois, les réseaux réellement exécutables n'ont pas n'importe quelle forme. Ainsi, nous pouvons interpréter un réseau comme un opérateur dont certains liens portent les arguments et d'autres le ou les résultats qu'il fournit. Dans ce cas, nous pouvons supposer que l'abréviation n'apparaîtra jamais dans un résultat. Cette méthode qui "oublie" certaines possibilités d'interaction est correcte car cela consiste à séquencialiser artificiellement le mécanisme de réduction.

Il peut sembler étrange, dans le cadre de l'évaluation partielle, de réduire les possibilités d'interaction en introduisant des abréviations étendues. Le but principal de ce mécanisme n'est pas lié directement à la spécialisation de programme mais elles permettent de démontrer certaines propriétés attachées aux agents. L'utilisation conjuguée de ces abréviations et de l'équivalence de comportement démontre, en effet, certaines propriétés algébriques telles que l'associativité et la commutativité de l'addition et de la multiplication, l'associativité de la concaténation de deux listes, l'équivalence des représentations unaires et binaires des entiers et de leur opérateurs...

Une fois ces propriétés démontrées, nous pouvons les utiliser dans le mécanisme d'évaluation partielle.

1.5 Conclusion sur les réseaux

L'évaluation partielle des réseaux d'interaction consiste, à partir d'un programme (une liste de règles d'interaction), à créer de nouveaux agents représentant une spécialisation d'un agent lorsque ses ports auxiliaires sont connectés à d'autres agents. Cette définition appelée *abréviation* va engendrer de nouvelles règles d'interaction qui permettent de définir ce nouvel agent.

Des extensions de ce mécanisme résolvent le problème de la gestion explicite des duplications et des effacements dans les réseaux (problème n'apparaissant pas dans la programmation fonctionnelle puisque les valeurs sont partagées).

Dans ce travail, les abréviations sont introduites par l'opérateur. Les règles qui les définissent sont alors automatiquement calculées. Un travail futur va consister à rendre ce mécanisme complètement automatique en utilisant les principes abordés dans la première partie de ce travail. Nous pensons que l'introduction de valeurs alternatives et d'événement pourra créer automatiquement de nouvelles abréviations.

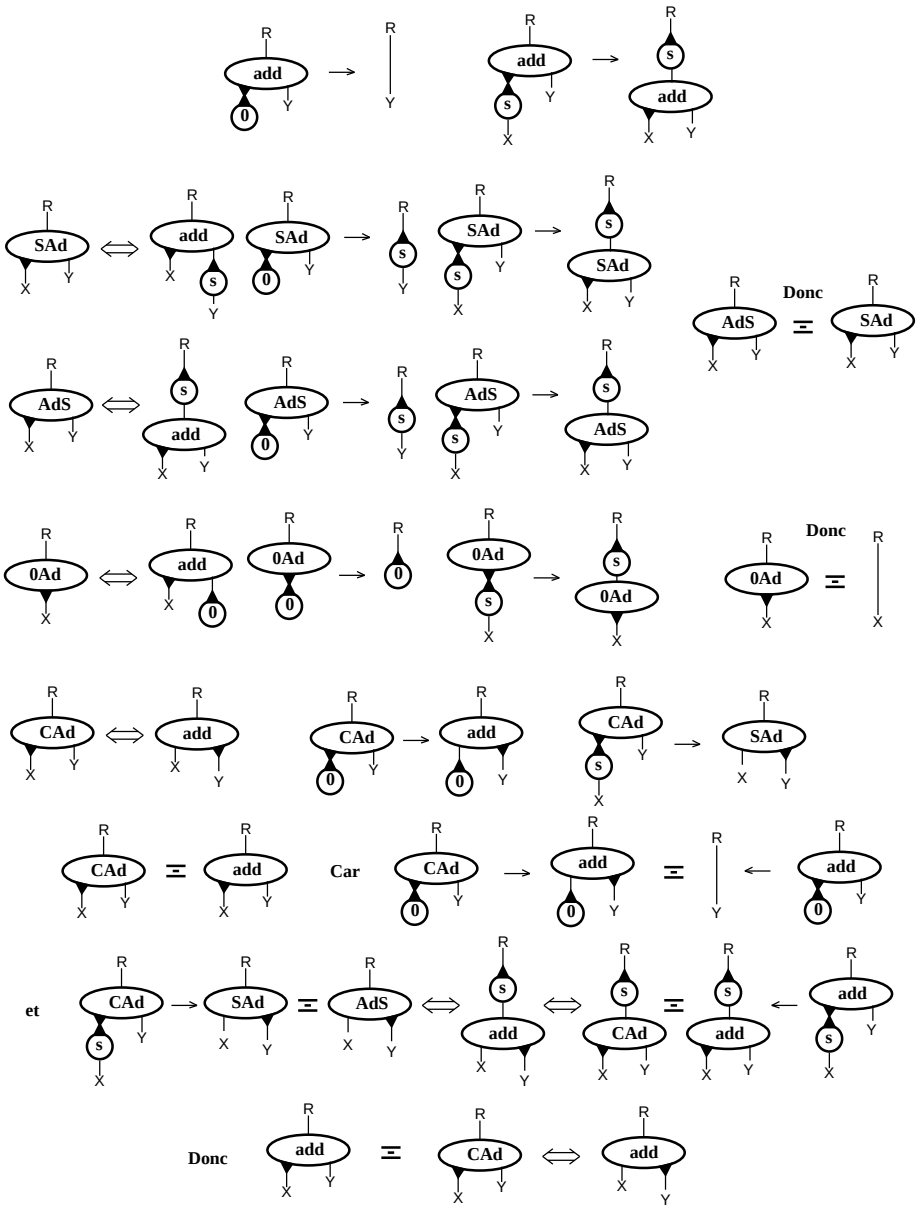


FIG. 1.14 – *La commutativité de l'addition*

1.5.1 Ajout d'événements

Comme pour la programmation fonctionnelle, nous pouvons parler d'événement. Un événement résulte simplement de l'utilisation d'une règle d'interaction entre deux agents comme un événement provient de la rencontre entre une structure de contrôle et un clone de constructeur dans le clonage de programmes CL. Ainsi, si, lors de la réduction d'un réseau, nous utilisons une règle entre deux agents, nous pouvons dire que les agents créés par cette réduction (ceux du membre droit de la règle) proviennent de cet événement et des événements qui ont créés les deux agents qui interagissent.

Cette idée s'insère très bien dans la notion de réseau spécialisé. En effet, nous pouvons ajouter à chaque agent utilisé au cours du mécanisme de spécialisation une information supplémentaire indiquant quels sont les événements qui ont participé à sa création et avoir ainsi une notion d'environnement d'événements.

Troisième partie

LaMix : Une implantation

Annexe A

LaML : un petit langage

Ces annexes présentent une implantation des idées introduites dans la première partie de ce travail. Nous avons appelé LaMix cette réalisation.

Le langage accepté par cet évaluateur partiel est un sous ensemble du langage CAML. Nous avons préféré utiliser un langage déjà existant plutôt que de travailler sur CL car ceci permet de tester les exemples directement. Nous avons appelé LaML ce sous-ensemble de CAML.

La syntaxe d'un programme LaML ressemble a celle de CAML. Toutefois, de nombreuses restrictions ont été nécessaires pour qu'il soit équivalent à CL.

- Ce langage ne comporte pas de fonction d'ordre supérieur et le type des expressions doit être monomorphe. Par conséquent, si un programme utilise des abstractions comme les listes, nous devons déclarer un type concret par version monomorphe de liste (une liste d'entiers, de chaînes de caractère...). Les fonctions polymorphes sur ces objets devront être dupliquées (par exemple, une fonction `append` sur les listes d'entiers, une sur les listes de chaînes de caractères...). Les fonction comme `map` devront être spécialisées *à la main* pour donner autant de versions qu'il n'y a de fonctions données en argument à `map`. Ces deux restrictions devraient disparaître lorsque LaMix pourra prendre en compte le polymorphisme et les fonctions d'ordre supérieur.

- LaML est purement fonctionnel. Il n'y a aucun effet de bord.

- Les motifs sont restreints à un seul constructeur. Nous pouvons étendre ceci à des motifs complexes par traduction des structures de contrôle en plusieurs conditionnelles imbriquées (par exemple, comme dans [Aug85]).

- Les motifs sur les produits ne sont pas reconnus. Cela nous a obligé à utiliser des projections dans CL. Dans LaML, un certain nombre d'identificateurs sont lié à ces projections. Ainsi, `fst` et `snd` sont les deux projections sur une paire. Pour les triplets, les fonctions `fst3`, `snd3` et `trd3` correspondent aux trois projections. Enfin, l'identificateur `proj_i_n` fournit la projection sur la i-ième composante d'un produit de n valeurs.

- LaML ne comporte pas de type de base comme les entiers ou les chaînes de caractères. Ils doivent être codés par des types abstraits (par exemple des entiers unaires). De même, il n'y a aucune primitive. Des fonctions comme le test d'égalité doivent être définies.

- Les identificateurs des types, des constructeurs et des fonctions doivent être uniques. En effet, les programmes LaML sont traduit en CL avant d'être spécialisés. Or, CL n'accepte qu'une seule définition par identificateur global.

A.1 Syntaxe de LaML

A.1.1 Notations

La description de la syntaxe utilise les conventions suivantes :

- Les symboles non-terminaux sont en italique, comme *ceci*.
- Les caractères terminaux sont tapés en caractères *machine à écrire*, comme `ceci`.
- Une suite de caractères s'écrit avec trois petits points, comme ceci : `A···Z`.
- Une composante entre accolades $\{\cdots\}$ dénote zéro ou plusieurs répétitions de cette composante.
- De même, une composante entre crochets $[\cdots]$ dénote une composante optionnelle.
- Les alternatives entre plusieurs composantes sont séparées par des barres verticales |.

A.1.2 Conventions lexicales

Espaces

Les caractères suivant sont considérés comme des espaces : *espace*, *nouvelle ligne*, *tabulation* et *retour chariot*. Les espaces sont ignorés mais permettent de séparer les identificateurs et les mots-clés.

Commentaires

Des commentaires peuvent être introduits par les deux caractères `(*` et terminés par `*)`. Les commentaires imbriqués sont correctement traités.

Identificateurs

$$\begin{array}{lcl} \textit{ident} & ::= & \textit{letter}\ \{\textit{letter} \mid \textit{digit} \mid \texttt{\_}\} \\ \textit{letter} & ::= & \texttt{A}\cdots\texttt{Z} \mid \texttt{a}\cdots\texttt{z} \\ \textit{digit} & ::= & \texttt{0}\cdots\texttt{9} \end{array}$$

A.1.3 Mots-clés

Les identificateurs suivants ne peuvent pas être utilisés :

```
and     in      let     match   of
rec     type    with    fst     snd
fst3    snd3    trd3    p_i_n   unit
```

A.1.4 Valeurs

Les valeurs manipulées par les programmes LaML sont soit des produits, soit des constructeurs (constants ou non-constants).

Produits

Le produit de n valeurs est noté $(e_1, \ldots, e_n)$. Les projections sur la i-ème composante correspond à l'opérateur `p_`i`_`n. Les mots-clés `fst` et `snd` correspondent aux deux projections d'une paire (`p_1_2` et `p_2_2`). Pour un triplet, les projections sont `fst3`, `snd3` et `trd3`. Le produit vide (noté `()`) ne possède pas de projection.

Le type produit est noté $t_1 * \cdots * t_n$, sauf le cas particulier où n est nul qui est désigné par le mot-clé `unit`.

Constructeurs

LaML permet de déclarer des types sommes. Cela permet de définir de nouveaux constructeurs. Ceux-ci peuvent être soit constants, comme les valeurs booléennes, soit possédant un argument. L'accès à l'argument d'une valeur construite avec un constructeur non-constant est accessible par filtrage (avec la construction `match ... with ...`).

A.1.5 Déclaration de types sommes

Les constructeurs d'un type somme doivent être déclarés par la construction `type ...`. La syntaxe de cette définition est la suivante :

type-definition ::= `type` *typedef* {`and` *typedef*}
typdef ::= *ident* `=` *constr-decl* {`|` *constr-decl*}
constr-decl ::= *ident*
| *ident* `of` *typeexpr*
typeexpr ::= `unit`
| *typeexpr* {`*` *typeexpr*}
| *ident*
| `(` *typeexpr* `)`

A.2 Expressions

expr ::= *ident* {*expr*}
| *expr* `,` *expr* {`,` *expr*}
| `()`
| `(` *expr* `)`
| `match` *expr* `with` *cases*
| `let` *binding* {`and` *binding*} `in` *expr*
cases ::= *case* {`|` *case*}
case ::= pattern `->` *expr*
pattern ::= *ident*
| *ident* `_`
| *ident ident*
binding ::= *ident* `=` *expr*

A.2.1 Identificateurs

L'expression *ident* {*expr*} correspond soit à une variable ou un paramètre, soit à l'application d'une fonction à un ou plusieurs arguments, soit l'utilisation d'un constructeur (constant ou non)

ou bien à l'utilisation d'une projection.

A.2.2 Produits

expr, *expr*{ , *expr*} permet de construire un n-uplet tandis que l'expression `()` renvoie la valeur *unit*.

A.2.3 Expressions conditionnelles

```
match expr with
        pattern_1 -> expr_1
      | ...
      | pattern_n -> expr_n
```

Cette expression calcule *expr* puis, évalue la branche correspondant au constructeur calculé (*pattern*$_i$ est constitué d'un constructeur). Pour les constructeurs non-constants, la variable du motif est liée à la valeur de l'argument du constructeur. Le symbole _ permet de ne pas utiliser de variable lorsque l'argument d'un constructeur non-constant n'est pas utilisé.

LaML ne possède pas la construction `if` e_1 `then` e_2 `else` e_3 qui est équivalente à l'expression :

```
match e_1 with
        true  -> e_2
      | false -> e_3
```

De plus, les constructeurs `true` et `false` et le type `bool` ne sont pas prédéfinis.

A.2.4 Définitions locales

```
let var_1 = expr_1
and ...
and var_n = expr_n
  in expr
```

Cette construction calcule les n expressions *expr*$_1$,...,*expr*$_n$ puis évalue *expr* dans un environnement où les n variables sont liées aux n valeurs calculées.

A.3 Déclaration de fonctions

Les fonctions (et les constantes globales) sont introduites en utilisant les constructions `let ...` et `let rec ...`. La première permet d'introduire des fonctions non-récursives tandis que la deuxième autorise les définitions récursives.

fun-definition ::= `let` [`rec`] *fundef* {`and` *fundef*}
fundef ::= *ident* {*ident*} `=` *expr*

La déclaration :

```
let fun_1 var_1^1 ... var_1^{k_1} = expr_1
and ...
and fun_n var_n^1 ... var_n^{k_n} = expr_n
```

définie les n fonctions *fun*$_1$,...,*fun*$_n$ ayant k_i paramètres. Lorsque la liste des paramètres est vide, la construction définie une constante globale.

A.4 Déclarations globales

Un programme LaML est une suite de déclarations de types sommes et de fonctions globales séparées par ;; :

$$
\begin{array}{lcl}
\textit{program} & ::= & \{\textit{phrase}\ ;;\} \\
\textit{phase} & ::= & \textit{type-definition} \\
 & | & \textit{fun-definition}
\end{array}
$$

Annexe B

Installation et utilisation de LaMix

LaMix est un programme Caml-Light [Ler93] écrit de manière modulaire. Cette version tourne sous la version 0.6 de Caml-Light.

B.1 Installation

LaMix est distribué dans le format source, comme un fichier `tar` compressé de nom `lamix1.tar.Z`.

B.1.1 Récupération des sources

Pour extraire les fichiers sources, il faut placer ce fichier dans un répertoire puis exécuter :

```
zcat lamix1.tar.Z | tar -xBf
```

Cette commande installe LaMix en créant un répertoire `lamix1` dans lequel les fichiers sources ainsi que quelques exemples sont créés.

B.1.2 Compilation de LaMix

Une fois les sources installés, les modules de LaMix doivent être compilés. Ce sont des fichiers Caml-Light d'extension `.ml` ainsi qu'un fichier pour l'analyseur lexicale de LaMl, d'extension `.mll` et l'analyseur syntaxique (`.mly`).

La compilation des modules est réalisée en lançant le script `compile.sh`. La commande :

```
sh compile.sh
```

permet de compiler les modules.

B.2 Utilisation de LaMix

B.2.1 Options

Lors de la phase de spécialisation, LaMix comporte deux stratégies de dépliage des fonctions.

- Par défaut, toutes les fonctions qui ne sont appelées qu'à un endroit du programme résiduel sont dépliées.
- Les fonctions qui ne sont appelées qu'à un endroit du programme résiduel sont dépliées si leur corps ne contient pas de structure de contrôle résiduelle. Cette option permet de produire un programme résiduel plus lisible que l'option par défaut.

B.2.2 Lancement de LaMix

Deux méthodes permettent d'exécuter LaMix. Soit en lançant une commande Unix, soit en lançant Caml-Light, en chargeant ses modules puis en exécutant des commandes Caml.

B.2.3 Commande shell

La commande `lamix` permet de lancer LaMix sans utiliser l'interpréteur Caml-Light. La syntaxe de cette commande est la suivante :

```
lamix [-f/u] function file_in file_out
```

Avec `function`, le nom de la fonction qui sera spécialisée, `file_in` le fichier LaML source dans lequel `function` est définie et `file_out` est le nom du programme LaML résiduel.

L'option `-f` permet de conserver la définition des fonctions dépliables mais dont le corps comporte au moins une structure de contrôle. Cette option permet de produire un programme résiduel plus lisible (bien que moins efficace).

L'option `-u` (par défaut) déplie toutes les fonctions pouvant l'être. Ceci est au détriment de la lisibilité du programme résiduel car de nombreuse structures de contrôle peuvent être imbriquées.

B.2.4 Avec Caml-Light

La seconde possibilité pour utiliser LaMix consiste à lancer Caml-Light (tapez `camllight`), à charger les modules de LaMix (tapez `include"lamix";;`) et à exécuter les fonctions de spécialisation en interactif.

```
$camllight
>       Caml Light version 0.6

#include"lamix";;
```

On dispose alors des fonctions suivantes :

- `lamix : string -> string -> string -> unit`. L'expression `lamix fun_id file_in file_out` spécialise la fonction `fun_id` définie par le programme LaML `file_in`. Le programme résiduel est écrit dans le fichier `file_out`.
- `lamix_stdout : string -> string -> unit`. L'expression `lamix_stdout fun_id file_in` spécialise la fonction `fun_id` définie par le programme LaML `file_in` et écrit le résultat sur la sortie standard.
- `set_fold_rmatch : unit -> unit` positionne l'option `-f` comme pour la commande shell.
- `set_unfold_rmatch : unit -> unit` positionne l'option `-u` (par défaut au lancement de LaMix) comme pour la commande shell.

Annexe C

Exemples de spécialisation

Cet annexe décrit les résultats expérimentaux obtenus avec LaMix. Cette expérience a permit de trouver un domaine plus simple d'événements et de cerner quelques problèmes qui devront être abordés dans une version future de LaMix.

Les programmes résiduels ont été obtenus avec l'option `-f` de la commande `lamix` qui ne déplie pas les fonctions dont le corps comporte au moins une structure de contrôle résiduelle (ce qui permet d'améliorer la lisibilité du programme résiduel)

Nous présentons des tests de comparaison entre les temps d'exécution des programmes initiaux et des programmes résiduels. Ces tests ont été effectués sur une station SPARC IPC en lançant le shell script `experiments.sh` inclus dans la distribution de LaMix. Les programmes résiduels ont été obtenus en dépliant le maximum de fonctions (option `-u` de LaMix). Les résultats, exprimés en secondes, ont été obtenus en exécutant les fonctions plusieurs fois (100000 fois pour les fonctions sur `append`).

C.1 Spécialisation sur des listes

Cette section rapporte les résultats obtenus avec la spécialisation de la fonction `append` qui concatène deux listes. Puisque LaML est monomorphe, ces listes sont des listes d'entiers unaires.

C.1.1 Le programme `append.ml`

Le programme source pour tous les exemples de spécialisation de `append` est le suivant :

```
(* Examples on nat lists *)
type nat = Z | S of nat ;;
type nat_list = N | C of nat * nat_list ;;

let rec append x y = match x with
    N   -> y
  | C v -> C(fst v,append (snd v) y) ;;

let c = C(S Z,C(Z,N)) ;;

let f1 x = append c x ;;
let f2 x = append x c ;;
let f3 x y = append (append c x) y ;;
```

```
let f4 x y = append c (append x y) ;;
let f5 x y = append x (append c y) ;;
let f6 x y = append (append x c) y ;;
```

La fonction `append` concatène deux listes d'entiers. L'identificateur `c` est lié à une liste constante de deux éléments. Les six fonctions `f1`,...,`f6` définissent des fonctions diverses de concaténation de listes.

C.1.2 Programmes résiduels

`f1` : Concaténation d'une liste constante et d'un liste inconnue

La fonction `f1` concatène la liste constante `c` à la liste donnée en argument. Sa spécialisation retourne le programme résiduel :

```
type nat = Z | S of nat
and nat_list = N | C of (nat * nat_list) ;;

let cte_1 = S Z ;;

let rec f1 x = C(cte_1,C(Z,x)) ;;
```

Ce programme définie la fonction `f1` de manière non-récursive en dépliant les fonctions `append` spécialisées lors du clonage.

`f2` : Concaténation d'une liste et d'un liste constante

La fonction `f2` appelle `append` avec pour deuxième argument une liste constante. Le programme résiduel ressemble à la fonction initiale `append` mais avec un seul argument (puisque le second est connu) :

```
type nat = Z | S of nat
and nat_list = N | C of (nat * nat_list) ;;

let cte_1 = C(S Z,C(Z,N)) ;;

let rec append_0 x = match x with
     N -> cte_1
   | C v -> C(fst v,append_0(snd v)) ;;

let rec f2 x = append_0 x ;;
```

Le cas terminal de `append_0` renvoie la liste constante `cte_1`.
Ces deux spécialisations de `append` correspondent bien aux définitions attendues (bien que nous aurions pu renommer `append_0` par `f2` dans le second programme).

`f3` : `append (append c x) y`

La troisième fonction `f3` concatène trois listes dont la première est constante. Le programme résiduel est :

```
type nat = Z | S of nat
```

```
and nat_list = N | C of (nat * nat_list) ;;

let cte_1 = S Z ;;

let rec append_0 x y = match x with
     N -> y
   | C v -> C(fst v,append_0 (snd v) y) ;;

let rec f3 x y = C(cte_1,C(Z,append_0 x y)) ;;
```

Cette nouvelle définition élimine un des appels à `append`. La fonction `append_0` est identique à la fonction initiale.

`f4: append c (append x y)`

En utilisant l'associativité de `append`, nous pouvons définir la même fonction par l'expression `append c (append x y)`.

Le programme résiduel est identique au précédent programme :

```
type nat = Z | S of nat
and nat_list = N | C of (nat * nat_list) ;;

let cte_1 = S Z ;;

let rec append_0 x y = match x with
     N -> y
   | C v -> C(fst v,append_0 (snd v) y) ;;

let rec f4 x y = C(cte_1,C(Z,append_0 x y)) ;;
```

Bien que les deux programmes résiduels soient identiques, le programme d'évaluation partielle est plus complexe sur le premier exemple que sur le second. En effet, le premier doit créer plus de clones de `append` que le second (correspondant aux deux appels à `append` avec comme premier argument une liste constante puis une liste débutant par une partie connue).

`f5: append x (append c y)`

Lorsque la liste constante est la seconde liste, les deux expressions possibles pour leur concaténation ne donnent pas le même résultat. Le programme résiduel pour la spécialisation de l'expression `append x (append c y)` lorsque `c` est constant est :

```
type nat = Z | S of nat
and nat_list = N | C of (nat * nat_list) ;;

let cte_1 = S Z ;;

let rec append_0 x y = match x with
     N -> y
   | C v -> C(fst v,append_0 (snd v) y) ;;

let rec f5 x y = append_0 x (C(cte_1,C(Z,y))) ;;
```

```
f6 : append (append x c) y
```

Avec l'expression `append (append x c) y`, nous obtenons :

```
type nat = Z | S of nat
and nat_list = N | C of (nat * nat_list) ;;

type nat_list_0 = C_B | C_A of (nat * nat_list_0) ;;

let cte_1 = S Z ;;

let rec append_0 x = match x with
     N -> C_B
   | C v -> C_A(fst v,append_0(snd v)) ;;

let rec append_1 x y = match x with
     C_B -> C(cte_1,C(Z,y))
   | C_A v -> C(fst v,append_1 (snd v) y) ;;

let rec f6 x y = append_1 (append_0 x) y ;;
```

Le programme obtenu n'est pas optimal car une liste intermédiaire est construite alors que la version précédente renvoie directement la liste résultat. Cet exemple montre les limites de l'évaluation partielle comme mécanisme de transformations de programme. Ici, l'associativité de `append` serait utile pour arriver à un programme optimal.

Cet exemple est intéressant car il introduit une spécialisation du type `nat_list`. Ce nouveau type `nat_list_0` comporte deux constructeurs, `C_B` et `C_A` dont le premier est constant mais pas le second. Le constructeur `C_B` correspond à la liste constante `C(S Z,C(Z,N))` du programme initial. `C_A` vient du constructeur de liste de la fonction `append` correspondant à l'appel `(append x c)`. Ce type décrit les listes de longueur inconnue mais se terminant pas la liste constante `C(S Z,C(Z,N))`. Cette liste apparaît comme le résultat de `append_0` et est utilisée par `append_1`.

C.1.3 Banc d'essai

Les comparaisons des temps d'exécution entre programmes sources et programmes résiduels ont été réalisées sur les six fonctions de concaténation avec trois types de données statiques. La première est une liste composée de deux entiers, la deuxième, une liste de dix éléments et la dernière de cinquante éléments. De même, la donnée dynamique représente une liste de deux, dix et cinquante

éléments.

Fonction spécialisée	Donnée statique	Donnée dynamique	Programme source (secondes)	Programme résiduel (secondes)	Gain
`append c x`	2	2	6.1	2.2	2.8
`append c x`	2	10	6.1	2.2	2.8
`append c x`	2	50	6.1	2.2	2.8
`append c x`	10	2	20.4	4.7	4.3
`append c x`	10	10	20.4	4.7	4.3
`append c x`	10	50	20.4	4.7	4.3
`append c x`	50	2	91.5	18.4	5.0
`append c x`	50	10	91.6	18.5	5.0
`append c x`	50	50	91.9	18.6	4.9
`append x c`	2	2	6.1	5.2	1.2
`append x c`	2	10	20.7	17.7	1.2
`append x c`	2	50	92.9	78.7	1.2
`append x c`	10	2	6.1	5.2	1.2
`append x c`	10	10	20.7	17.7	1.2
`append x c`	10	50	93.3	79.3	1.2
`append x c`	50	2	6.1	5.2	1.2
`append x c`	50	10	20.8	17.8	1.2
`append x c`	50	50	93.3	79.2	1.2
`append (append c x) x`	2	2	14.9	7.5	2.0
`append (append c x) x`	2	10	30.6	21.6	1.4
`append (append c x) x`	2	50	107.3	93.3	1.2
`append (append c x) x`	10	2	43.6	10.3	4.2
`append (append c x) x`	10	10	59.2	24.5	2.4
`append (append c x) x`	10	50	131.4	102.0	1.3
`append c (append x x)`	2	2	11.3	7.5	1.5
`append c (append x x)`	2	10	25.5	21.6	1.2
`append c (append x x)`	2	50	97.3	93.3	1.0
`append c (append x x)`	10	2	25.9	10.2	2.5
`append c (append x x)`	10	10	41.2	24.5	1.7
`append c (append x x)`	10	50	118.6	102.0	1.2
`append x (append c x)`	2	2	11.3	7.3	1.5
`append x (append c x)`	2	10	25.8	22.0	1.2
`append x (append c x)`	2	50	97.3	93.0	1.0
`append x (append c x)`	10	2	25.6	10.0	2.6
`append x (append c x)`	10	10	40.3	24.2	1.7
`append x (append c x)`	10	50	118.7	101.8	1.2
`append (append x c) x`	2	2	14.9	11.5	1.3
`append (append x c) x`	2	10	44.1	37.9	1.2
`append (append x c) x`	2	50	190.3	172.6	1.1
`append (append x c) x`	10	2	29.1	13.8	2.1
`append (append x c) x`	10	10	59.2	42.0	1.4
`append (append x c) x`	10	50	216.6	187.0	1.2

Le premier jeu d'essai concernant l'expression `append c x` permet de voir que le gain pour `c` et `x` grands est de cinq. Cela donne une idée de la différence d'efficacité lorsque nous remplaçons une fonction récursive sur une structure de données connue par l'expression équivalente des fonctions dépliées.

Le second jeu d'essai sur `append x c` donne une idée du gain (1,2) obtenu lorsqu'un paramètre d'une fonction disparaît (car il est constant).

Les jeux suivants se placent entre ces deux résultats (ils utilisent les deux types de transformations ci-dessus).

C.2 Spécialisation d'un programme de filtrage

Cette seconde section rapporte les résultats obtenus en spécialisant un programme de filtrage de motifs. Nous avons testé deux versions de ce type de programme. Ces programmes sont adaptés de programmes de [MHD94]

C.2.1 Programme de filtrage simple

Le premier programme vérifie que deux motifs sont syntaxiquement égaux. Les deux motifs sont des arbres binaires dont les feuilles sont des naturels. Le programme descend simultanément les deux arborescences et s'arrête lorsque les sous-motifs sont différents :

```
(* A simple pattern matching program *)
(* pmatch pattern1 pattern2 -> Bool *)

type Bool = True | False ;;
(* Nat == unary numbers *)
type Nat = S of Nat | Z ;;
(* Patterns *)
type NatTree = NatLeaf of Nat | NatNode of NatTree * NatTree ;;

(* equal on identifiers *)
let rec eq_nat x y = match x with
     S xx -> (match y with S yy -> eq_nat xx yy | Z -> False)
   | Z    -> (match y with S yy -> False        | Z -> True ) ;;

let rec pm p1 p2 = match p1 with
    NatLeaf i -> (match p2 with
           NatLeaf j -> pm_leaf i j
         | NatNode _ -> False)
  | NatNode xx -> (match p2 with
           NatLeaf _  -> False
         | NatNode yy -> pm_node xx yy)

and pm_leaf i j = eq_nat i j

and pm_node xx yy = match pm (snd xx) (snd yy) with
     False    -> False
   | True     -> pm (fst xx) (fst yy) ;;

let pmatch p1 p2 = pm p1 p2 ;;

let pat = NatNode(NatNode(NatLeaf(S(S(Z))),NatLeaf(S(Z))),
                  NatNode(NatLeaf(S(S(S(Z)))),NatLeaf(Z))) ;;

let f y = pmatch pat y ;;
```

Spécialisation

Nous pouvons spécialiser ce programme lorsque l'un des deux motifs est connu (le motif `pat` ci-dessus). La fonction `f` appelle la fonction de filtrage avec pour premier paramètre, un motif constant. LaMix renvoie le programme résiduel suivant :

```
type Bool = True | False
and Nat = S of Nat | Z
and NatTree = NatLeaf of Nat | NatNode of (NatTree * NatTree) ;;

type Bool_2 = False_M | False_L | True_C
and Bool_1 = False_K | False_J | False_I | False_H | False_G
           | False_F | False_E | False_D | True_B
and Bool_0 = False_C | False_B | False_A | True_A ;;

let rec eq_nat_0 y = match y with S yy -> False_L | Z -> True_C ;;

let rec pm_0 p2 = match p2 with
     NatLeaf j -> eq_nat_0 j | NatNode _ -> False_M ;;

let rec eq_nat_1 y = match y with S yy -> False_G | Z -> True_B ;;

let rec eq_nat_2 y = match y with S yy -> eq_nat_1 yy | Z -> False_H ;;

let rec eq_nat_3 y = match y with S yy -> eq_nat_2 yy | Z -> False_I ;;

let rec eq_nat_4 y = match y with S yy -> eq_nat_3 yy | Z -> False_J ;;

let rec pm_1 p2 = match p2 with
     NatLeaf j -> eq_nat_4 j | NatNode _ -> False_K ;;

let rec pm_node_0 yy = match pm_0(snd yy) with
     False_M -> False_F | False_L -> False_E | True_C -> pm_1(fst yy) ;;

let rec pm_2 p2 = match p2 with
     NatLeaf _ -> False_D | NatNode yy -> pm_node_0 yy ;;

let rec eq_nat_5 y = match y with S yy -> False_A | Z -> True_A ;;

let rec eq_nat_6 y = match y with S yy -> eq_nat_5 yy | Z -> False_B ;;

let rec pm_3 p2 = match p2 with
     NatLeaf j -> eq_nat_6 j | NatNode _ -> False_C ;;

let rec eq_nat_7 y = match y with S yy -> False | Z -> True ;;

let rec eq_nat_8 y = match y with S yy -> eq_nat_7 yy | Z -> False ;;

let rec eq_nat_9 y = match y with S yy -> eq_nat_8 yy | Z -> False ;;
```

```
let rec pm_4 p2 = match p2 with
     NatLeaf j -> eq_nat_9 j | NatNode _ -> False ;;

let rec pm_node_1 yy = match pm_3(snd yy) with
     False_C -> False | False_B -> False | False_A -> False
   | True_A -> pm_4(fst yy) ;;

let rec pm_5 p2 = match p2 with
     NatLeaf _ -> False | NatNode yy -> pm_node_1 yy ;;

let rec pm_node_2 yy = match pm_2(snd yy) with
     False_K -> False | False_J -> False | False_I -> False
   | False_H -> False | False_G -> False | False_F -> False
   | False_E -> False | False_D -> False
   | True_B -> pm_5(fst yy) ;;

let rec pm_6 p2 = match p2 with
     NatLeaf _ -> False | NatNode yy -> pm_node_2 yy ;;

let rec f y = pm_6 y ;;
```

Ce programme a éliminé complètement le motif constant. De nombreuses versions des fonctions originales sont apparues. Elles correspondent aux diverses spécialisation des fonctions initiales par rapport à la structure arborescente du motif constant.

Toutefois, nous voyons que LaMix a aussi créé de nouveaux types en spécialisant le type des booléens. Ceci se traduit par un nombre injustifié de cas pour les structures de contrôle sur ces types. Ainsi, la fonction `pm_node_2` comportent 8 cas pour les clones de `False` avec la même expression associée (`False`). Ceci permet d'introduire une remarque sur la qualité du code produit par LaMix. Comme le mécanisme de clonage ne s'occupe pas des valeurs associées aux clones de constructeur (leur argument) lorsqu'il compare deux clones, il est amené à distinguer des clones correspondant à une même valeur (ici la valeur `False`). Cela entraîne un nombre parfois élevé de cas pour des structures de contrôle spécialisées.

Dans l'exemple ci-dessus, LaMix devrait pouvoir s'apercevoir que ces clones de `False` correspondent exactement à la même valeur et qu'il ne faut pas les distinguer. Comme ce mécanisme ressemble à celui de *fusion de clones*, nous pensons ajouter un mécanisme, lors de la phase de fusion/généralisation pour forcer ces clones de constructeurs à être identifiés.

Modification du programme de filtrage

Pour résoudre ce problème, nous pouvons modifier légèrement le programme source. Pour forcer les valeurs booléennes à fusionner, deux paramètres constants égaux à `True` et `False` sont ajoutés aux fonctions. Le programme de filtrage modifié est alors :

```
(* A simple pattern matching program *)
(* pmatch pattern1 pattern2 -> Bool *)
(* With global booleans *)

type Bool = True | False ;;
```

```
(* Nat == unary numbers *)
type Nat = S of Nat | Z ;;
(* Patterns *)
type NatTree = NatLeaf of Nat
             | NatNode of NatTree * NatTree ;;

(* equal on identifiers *)
let rec eq_nat x y tt ff = match x with
     S xx -> (match y with S yy -> eq_nat xx yy tt ff | Z -> ff)
   | Z    -> (match y with S yy -> ff                 | Z -> tt) ;;


let rec pm p1 p2 tt ff = match p1 with
    NatLeaf i -> (match p2 with
           NatLeaf j -> pm_leaf i j tt ff
         | NatNode _ -> ff)
  | NatNode xx -> (match p2 with
           NatLeaf _  -> ff
         | NatNode yy -> pm_node xx yy tt ff)

and pm_leaf i j tt ff = eq_nat i j tt ff

and pm_node xx yy tt ff = match pm (snd xx) (snd yy) tt ff with
     False    -> ff
   | True     -> pm (fst xx) (fst yy) tt ff ;;

let pmatch p1 p2 = pm p1 p2 True False ;;

let pat = NatNode(NatNode(NatLeaf(S(S(Z))),NatLeaf(S(Z))),
                  NatNode(NatLeaf(S(S(S(Z)))), NatLeaf(Z))) ;;

let f y = pmatch pat y ;;
```

Spécialisation du filtreur modifié

Nous obtenons le programme résiduel suivant :

```
type Bool = True | False
and Nat = S of Nat | Z
and NatTree = NatLeaf of Nat | NatNode of (NatTree * NatTree) ;;

let rec eq_nat_0 y = match y with S yy -> False | Z -> True ;;

let rec pm_0 p2 = match p2 with
     NatLeaf j -> eq_nat_0 j | NatNode _ -> False ;;

let rec eq_nat_1 y = match y with S yy -> False | Z -> True ;;

let rec eq_nat_2 y = match y with S yy -> eq_nat_1 yy | Z -> False ;;
```

```
let rec eq_nat_3 y = match y with S yy -> eq_nat_2 yy | Z -> False ;;

let rec eq_nat_4 y = match y with S yy -> eq_nat_3 yy | Z -> False ;;

let rec pm_1 p2 = match p2 with
     NatLeaf j -> eq_nat_4 j | NatNode _ -> False ;;

let rec pm_node_0 yy = match pm_0(snd yy) with
     False -> False | True -> pm_1(fst yy) ;;

let rec pm_2 p2 = match p2 with
     NatLeaf _ -> False | NatNode yy -> pm_node_0 yy ;;

let rec eq_nat_5 y = match y with S yy -> False | Z -> True ;;

let rec eq_nat_6 y = match y with S yy -> eq_nat_5 yy | Z -> False ;;

let rec pm_3 p2 = match p2 with
     NatLeaf j -> eq_nat_6 j | NatNode _ -> False ;;

let rec eq_nat_7 y = match y with S yy -> False | Z -> True ;;

let rec eq_nat_8 y = match y with S yy -> eq_nat_7 yy | Z -> False ;;

let rec eq_nat_9 y = match y with S yy -> eq_nat_8 yy | Z -> False ;;

let rec pm_4 p2 = match p2 with
     NatLeaf j -> eq_nat_9 j | NatNode _ -> False ;;

let rec pm_node_1 yy = match pm_3(snd yy) with
     False -> False | True -> pm_4(fst yy) ;;

let rec pm_5 p2 = match p2 with
     NatLeaf _ -> False | NatNode yy -> pm_node_1 yy ;;

let rec pm_node_2 yy = match pm_2(snd yy) with
     False -> False | True -> pm_5(fst yy) ;;

let rec pm_6 p2 = match p2 with
     NatLeaf _ -> False | NatNode yy -> pm_node_2 yy ;;

let rec f y = pm_6 y ;;
```

Dans cette version, la spécialisation du type `Bool` a disparu car, lors de la phase de fusion/généralisation, les constructeurs `True` et `False` ont été généralisés (ils peuvent apparaître comme résultat de `f`).

Ce programme ressemble étrangement à un programme testant l'égalité d'une structure à une structure constante. C'est une sorte de spécialisation de la fonction générique d'égalité structurelle

de Caml-Light lorsque l'un de ses paramètres est constant.

Toutes les fonctions des deux programmes résiduels (sauf `f`) peuvent être dépliées puisque les appels ne sont pas récursifs. En utilisant l'option `-u` de LaMix, nous obtenons une seule fonction.

C.2.2 Filtrage complet

Le second exemple de programme de filtrage est plus conforme à ce type de problèmes car, au lieu de rendre une valeur booléenne, ce programme renvoie une substitution de variables :

```
(* A pattern matching program returning a substitution *)
(* CPS version *)

(* Nat == unary numbers *)
type Nat = S of Nat | Z ;;

(* Values *)
type Value   = Atom of Nat | Cons of Value * Value ;;

(* Patterns *)
type Pattern = PVar of Nat
             | PAtom of Nat
             | PCons of Pattern * Pattern ;;

(* Substitution *)
type Subst = CSubst of (Nat * Value) * Subst | NSubst ;;

(* PM result *)
type Result = Failure | Success of Subst ;;

type Cont = End | Do of (Pattern * Value) * Cont ;;

(* equal on atoms *)
let rec eq_atom x y subst cont = match x with
     S xx -> (match y with
                   S yy -> eq_atom xx yy subst cont
                 | Z    -> Failure)
   | Z    -> (match y with
                   S yy -> Failure
                 | Z    -> pm_cont subst cont )

and pm pat val subst cont = match pat with
    PVar i  -> pm_cont (CSubst((i,val),subst)) cont
  | PAtom i -> (match val with
                    Atom j -> eq_atom i j subst cont
                  | Cons _ -> Failure)
  | PCons i -> (match val with
                    Atom _ -> Failure
                  | Cons j -> pm_cons i j subst cont)
```

```
and pm_cons pp vv subst cont =
        pm (snd pp) (snd vv) subst (Do((fst pp,fst vv),cont))

and pm_cont subst cont = match cont with
     End   -> Success subst
   | Do c  -> pm (fst(fst c)) (snd(fst c)) subst (snd c) ;;

let pmatch p1 p2 = (pm p1 p2 NSubst End) ;;

let pat = PCons(PCons(PVar(S(S(Z))),PAtom(S(Z))),
                PCons(PAtom(S(S(S(Z)))),PVar(Z))) ;;

let f y = pmatch pat y ;;
```

Ce programme de filtrage prend, comme premier argument, un motif représentant un arbre binaire dont les feuilles sont soit un atome, soit une variable, et comme second argument, un arbre. Il renvoie alors un message indiquant que le filtrage a échoué (valeur `Failure`) ou bien une substitution des variables du motifs.

Nous pouvons spécialiser cette fonction lorsque le motif est connue (la valeur `pat` ci-dessus). La fonction `f` appelle la fonction de filtrage avec un motif constant. Son évaluation partielle avec LaMix donne le programme résiduel suivant :

```
type Nat = S of Nat | Z
and Value = Atom of Nat | Cons of (Value * Value)
and Pattern = PVar of Nat | PAtom of Nat | PCons of (Pattern * Pattern)
and Subst = CSubst of ((Nat * Value) * Subst) | NSubst
and Result = Failure | Success of Subst
and Cont = End | Do of ((Pattern * Value) * Cont) ;;

let cte_1 = S(S Z) ;;

let rec eq_atom_0 y subst cont = match y with
     S yy -> Failure | Z -> Success(CSubst((cte_1,cont),subst)) ;;

let rec eq_atom_1 y subst cont = match y with
     S yy -> eq_atom_0 yy subst cont | Z -> Failure ;;

let rec pm_0 val subst cont = match val with
     Atom j -> eq_atom_1 j subst cont | Cons _ -> Failure ;;

let rec pm_1 val subst = match val with
     Atom _ -> Failure | Cons j -> pm_0 (snd j) subst (fst j) ;;

let rec eq_atom_2 y subst cont = match y with
     S yy -> Failure | Z -> pm_1 cont subst ;;

let rec eq_atom_3 y subst cont = match y with
     S yy -> eq_atom_2 yy subst cont | Z -> Failure ;;
```

```
let rec eq_atom_4 y subst cont = match y with
     S yy -> eq_atom_3 yy subst cont | Z -> Failure ;;

let rec eq_atom_5 y subst cont = match y with
     S yy -> eq_atom_4 yy subst cont | Z -> Failure ;;

let rec pm_2 val subst cont = match val with
     Atom j -> eq_atom_5 j subst cont | Cons _ -> Failure ;;

let rec pm_3 val cont = match val with
     Atom _ -> Failure
   | Cons j -> let cont = (fst j,cont)
                in pm_2 (fst cont) (CSubst((Z,snd j),NSubst)) (snd cont) ;;

let rec pm_4 val = match val with
     Atom _ -> Failure | Cons j -> pm_3 (snd j) (fst j) ;;

let rec f y = pm_4 y ;;
```

Ce programme a éliminé toutes les références au motif constant. De plus, le type initial pour les continuations (`Cont`) a été transformé en produit. Les constructeurs `End` et `Do` ont disparu.

Comme pour l'exemple précédent, les fonctions résiduelles comportant au moins une structure de contrôle n'ont pas été dépliées, pour une meilleur compréhension. Elles pourraient être dépliées ce qui donnerait une définition de `f` sans appel de fonction.

C.2.3 Banc d'essai

La comparaison entre les temps d'exécution des programmes sources et des programmes résiduels concerne les trois programmes de filtrage : `pm1.ml` est le programme de filtrage simple, `pm1_bis.ml` est le même programme dans lequel les valeurs booléennes sont données en argument des fonctions et `pm2.ml` est le programme de filtrage renvoyant une substitution des variables du motif en cas de succès. Ces programmes sont spécialisés par rapport à deux motifs constants. De même, les programmes sont testés avec deux valeurs différentes :

Filteur	Motif	Donnée dynamique	Programme source (secondes)	Programme résiduel (secondes)	Gain
pm1.ml	((2,1),(3,0))	((2,1),(3,0))	34.2	11.3	3.0
pm1.ml	((2,1),(3,0))	(((2,1),(3,0)),((2,1),(3,0)))	9.0	4.0	2.3
pm1.ml	(((2,1),(3,0)),((2,1),(3,0)))	((2,1),(3,0))	9.0	4.0	2.3
pm1.ml	(((2,1),(3,0)),((2,1),(3,0)))	(((2,1),(3,0)),((2,1),(3,0)))	69.1	23.1	3.0
pm1_bis.ml	((2,1),(3,0))	((2,1),(3,0))	47.3	11.1	4.3
pm1_bis.ml	((2,1),(3,0))	(((2,1),(3,0)),((2,1),(3,0)))	11.7	3.9	3.0
pm1_bis.ml	(((2,1),(3,0)),((2,1),(3,0)))	((2,1),(3,0))	11.7	3.9	3.0
pm1_bis.ml	(((2,1),(3,0)),((2,1),(3,0)))	(((2,1),(3,0)),((2,1),(3,0)))	96.7	22.7	4.3
pm2.ml	((x2,1),(3,x0))	((2,1),(3,0))	41.1	9.0	4.6
pm2.ml	((x2,1),(3,x0))	(((2,1),(3,0)),((2,1),(3,0)))	16.2	3.7	4.4
pm2.ml	(((x2,1),(3,x0)),((2,x1),(3,0)))	((2,1),(3,0))	12.4	3.5	3.5
pm2.ml	(((x2,1),(3,x0)),((2,x1),(3,0)))	(((2,1),(3,0)),((2,1),(3,0)))	87.9	18.6	4.7

Le gain est de l'ordre de 3 pour le programme de filtrage simple (en cas de succès). Il passe à $4,3$ pour la version modifiée. Cela s'explique par le fait que durant la spécialisation, les paramètres `tt`

et `ff` des fonctions disparaissent puisqu'ils sont constants (dans la première version, ces arguments n'étaient pas présents).

La version renvoyant une substitution en cas de succès du filtrage donne un gain un peu plus grand, de l'ordre de $4,6$ en cas de succès.

C.3 Spécialisation de TP

Nous avons présenté un petit langage à la Pascal. Cette section décrit les essais de la spécialisation d'un interpréteur de ce langage pour un programme TP donné.

C.3.1 Interpréteur TP standard

Le programme LaML suivant définit un interpréteur TP standard :

```
(* Tiny Pascal : a small pascal interpretor *)

type Bool = True | False ;;

(* Identifiers are unary numbers *)
type Ident =  N of Ident | E ;;

(* Nat == unary numbers *)
type Nat = S of Nat | Z ;;

(* List of numbers *)
type NatList = CNat of Nat * NatList | NNat ;;

(* Environments for identifiers *)
type VarEnv = CVarEnv of Ident * Nat * VarEnv | NVarEnv ;;

(* Expressions *)
type Expr = Zero
          | Var of Ident
          | Succ of Expr
          | Pred of Expr ;;

(* Commands *)
type Com = Block of ComList
         | If of Expr * Com * Com
         | While of Expr * Com
         | Let of Ident * Expr
         | Proc of Ident
         | Read of Ident
         | Write of Expr

(* Lists of commands *)
and ComList = CCom of Com * ComList | NCom

(* Lists of procedures definitions *)
```

```
and DeclList = CDecl of Ident * Com * DeclList | NDecl ;;

let fst3 (x,y,z) = x
and snd3 (x,y,z) = y
and trd3 (x,y,z) = z ;;

(* Error : loop forever *)
let rec error x = error x ;;

(* equal on identifiers *)
let rec eq_Ident x y = match x with
     N xx -> (match y with N yy -> eq_Ident xx yy | E -> False)
   | E    -> (match y with N yy -> False          | E -> True ) ;;

(* Find the definition of a procedure *)
let rec get_p pid defs = match defs with
     CDecl d -> (match eq_Ident pid (fst3 d) with
                    True  -> (snd3 d)
                  | False -> get_p pid (trd3 d))
   | NDecl   -> error() ;;

(* Find the definition of a procedure from the program *)
let get_proc_com pid prog = get_p pid (fst prog) ;;

(* Creat an empty environment *)
let make_empty_env reads = (NVarEnv,reads,NNat) ;;

(* Get the next read value *)
let get_read_val env = match snd3 env with
    CNat v -> fst v
  | NNat   -> error() ;;

(* Pop the first read value *)
let pop_read_vals env = match snd3 env with
    CNat v -> (fst3 env,snd v,trd3 env)
  | NNat   -> error() ;;

(* Push a value in the write buffer *)
let push_write_val v env = (fst3 env,snd3 env,CNat(v,trd3 env)) ;;

(* Get the write buffer from an environment *)
let get_writes env = trd3 env ;;

(* Update the value associated to an identifier *)
let rec upd vid n vars = match vars with
     CVarEnv d -> (match eq_Ident vid (fst3 d) with
                      True  -> CVarEnv(vid,n,(trd3 d))
                    | False -> CVarEnv(fst3 d,snd3 d,upd vid n (trd3 d)))
```

```
   | NVarEnv   -> CVarEnv(vid,n,NVarEnv) ;;

(* Update the value associated to an identifier in an environment *)
let update_var vid n env = (upd vid n (fst3 env),snd3 env,trd3 env) ;;

(* Get the value associated to an identifier *)
let rec get id vars = match vars with
     CVarEnv d -> (match eq_Ident id (fst3 d) with
                      True  -> (snd3 d)
                    | False -> get id (trd3 d))
   | NVarEnv   -> error() ;;

(* Get the value associated to an identifier from an environment *)
let get_var id env = get id (fst3 env) ;;

(* Compute an expression with an environment *)
let rec run_e expr env = match expr with
     Zero   -> Z
   | Var v  -> get_var v env
   | Succ e -> S(run_e e env)
   | Pred e -> (match run_e e env with
                    Z   -> error()
                  | S v -> v) ;;

(* Execute a command into a given environment *)
(* and return the modified environment *)
let rec run_c prog com env = match com with
     Block c -> run_cs prog c env
   | If c    -> (match run_e (fst3 c) env with
                    S _ -> run_c prog (snd3 c) env
                  | Z   -> run_c prog (trd3 c) env)
   | While c -> (match run_e (fst c) env with
                    S _ -> run_c prog com (run_c prog (snd c) env)
                  | Z   -> env)
   | Let c   -> update_var (fst c) (run_e (snd c) env) env
   | Proc c  -> run_c prog (get_proc_com c prog) env
   | Read c  -> update_var c (get_read_val env) (pop_read_vals env)
   | Write c -> push_write_val (run_e c env) env

(* Execute a list of commands (a block) *)
and run_cs prog coms env = match coms with
     CCom c  -> run_cs prog (snd c) (run_c prog (fst c) env)
   | NCom    -> env ;;

(* Execute a program with a read list of values *)
(* an return the write buffer *)
(* A program is a list of procedure definition and a command *)
let exec prog reads =
```

```
let env = make_empty_env reads in
 let new_env = run_c prog (snd prog) env
  in get_writes new_env ;;
```

La fonction `exec` prend comme arguments un programme TP composé d'une liste de définitions de procédures et d'une instruction, et une liste de valeurs. Elle renvoie une liste de valeurs. La liste de valeurs donnée en argument correspond aux valeurs qui sont successivement lues par les instructions `read`. La liste renvoyée correspond aux valeurs écrites par `write`.

L'exécution d'une command TP (par la fonction `run_c`) prend une commande et un environnement et renvoie l'environnement modifié. Un environnement est composé d'une liste de valeurs non encore lues, d'une liste de valeurs écrites par `write` et d'une structure liant les variables (globales) du programme TP à leur valeur actuelle.

Les évaluateurs partiels basés sur une phase d'analyse des temps de liaison (tel Similix ou Scheme) spécialisent mal les structures représentant les environnements dans ce type d'interpréteurs (sans continuations). En effet, ces structures sont renvoyées comme résultat de structures de contrôle dont la valeur de test n'est pas connue lors de la spécialisation.

Spécialisation d'un programme TP

Considérons le programme TP suivant qui calcule la somme de deux nombres unaires :

```
program ADD1;
  procedure add;       (* x + y -> z *)
    begin
      z := y;
      while x do
        begin
          x := pred(x);
          z := succ(z)
        end
    end;

  begin
    read(x);
    read(y);
    add;
    write(z)
  end
```

Nous pouvons essayer LaMix sur ce programme en ajoutant la définition de la fonction `ADD1` :

```
(* Example 1 : ADD1 : Nat -> Nat -> Nat *)
(*  Using a while loop *)
let x = N(E) ;;
let y = N(N(E)) ;;
let z = N(N(N(E))) ;;
let add = N(N(N(N(E)))) ;;

let proc_add1 =
  Block(
```

```
   CCom(Let(z,Var y),
   CCom(While(Var x,
          Block(
            CCom(Let(x,Pred(Var x)),
            CCom(Let(z,Succ(Var z)),
            NCom))) ),
   NCom))) ;;

let proc_body1 =
  Block(
   CCom(Read x,
   CCom(Read y,
   CCom(Proc add,
   CCom(Write(Var z),
   NCom))))) ;;

let prog1 = (CDecl(add,proc_add1,
              NDecl),proc_body1) ;;

let ADD1 x y = match exec prog1 (CNat(x, CNat(y, NNat))) with
                 CNat w -> fst w
               | NNat   -> error() ;;
```

La fonction `ADD1` calcule la somme de ses deux arguments (des nombres en représentation unaire). Le programme TP exécute une boucle `while`.

Le programme résiduel définissant `ADD1` est alors le suivant :

```
type Bool = True | False
and Ident = N of Ident | E
and Nat = S of Nat | Z
and NatList = CNat of (Nat * NatList) | NNat
and VarEnv = CVarEnv of (Ident * Nat * VarEnv) | NVarEnv
and Expr = Zero | Var of Ident | Succ of Expr | Pred of Expr
and Com = Block of ComList | If of (Expr * Com * Com)
 | While of (Expr * Com) | Let of (Ident * Expr)
 | Proc of Ident | Read of Ident | Write of Expr
and ComList = CCom of (Com * ComList) | NCom
and DeclList = CDecl of (Ident * Com * DeclList) | NDecl ;;

type VarEnv_1 = CVarEnv_D of (Nat * (Nat * Nat))
              | CVarEnv_C of (Nat * (Nat * Nat))
and VarEnv_0 = CVarEnv_B of (Nat * (Nat * Nat))
             | CVarEnv_A of (Nat * (Nat * Nat)) ;;

let rec bottom x = bottom x ;;

let rec get_0 vars = match vars with
     CVarEnv_D d -> snd(snd d) | CVarEnv_C d -> snd(snd d) ;;
```

```
let rec get_1 vars = match vars with
     CVarEnv_D d -> fst d | CVarEnv_C d -> fst d ;;

let rec get_2 vars = match vars with
     CVarEnv_B d -> snd(snd d) | CVarEnv_A d -> snd(snd d) ;;

let rec get_3 vars = match vars with
     CVarEnv_D d -> fst d | CVarEnv_C d -> fst d ;;

let rec run_e_0 env = match get_3 env with Z -> bottom() | S v -> v ;;

let rec upd_0 n vars = match vars with
     CVarEnv_D d -> CVarEnv_B(n,snd d) | CVarEnv_C d -> CVarEnv_A(n,snd d) ;;

let rec upd_1 n vars = match vars with
     CVarEnv_B d -> CVarEnv_C(fst d,(fst(snd d),n))
   | CVarEnv_A d -> CVarEnv_C(fst d,(fst(snd d),n)) ;;

let rec run_c_0 env = match get_1 env with
     S _ -> run_c_0(let env = upd_0 (run_e_0 env) env
                    in upd_1 (S(get_2 env)) env)
   | Z -> env ;;

let rec ADD1 x y =
  let reads = (x,y)
   in snd(let env = run_c_0(CVarEnv_D(fst reads,(snd reads,snd reads)))
          in (env,get_0 env)) ;;
```

Dans ce programme, nous pouvons constater que le programme TP a complètement disparu. Les environnements sur les variables du programme TP ont été remplacés par les types `VarEnv_1` et `VarEnv_0`. Cette opération a permis de simplifier ces structures en éliminant les variables et en ne laissant qu'un seul constructeur par environnement (un des quatre constructeurs `CVarEnv_A`, `CVarEnv_B`, `CVarEnv_C` et `CVarEnv_D`). La boucle `while` du programme TP a été remplacée par la fonction récursive terminale `run_c_0`.

Les fonctions comportant une structure de contrôle résiduelle n'ont pas été dépliées. Cela concerne toutes les fonctions sauf `run_c_0` et `ADD1`. Nous pourrions les déplier ce qui donnerait un programme où seules sont définies les deux fonctions `run_c_0` et `ADD1`.

La correspondance entre les fonctions du programme résiduel et du programme initial est la suivante :

- Les fonctions `get_1` et `get_3` correspondent à la fonction `get` initiale spécialisée pour la variable `x`.
- La fonction `upd_0` spécialise `upd` pour cette même variable.
- Les fonctions `get_0`, `get_2` et `upd_1` correspondent aux fonctions `get` et `upd` pour la variable `z`.
- `run_e_0` provient de la spécialisation de `run_e` pour l'expression `pred(x)`.

- `bottom` remplace la fonction `error` qui, initialement, était activée lorsqu'une erreur se produisait dans l'exécution du programme TP. Par exemple lors de l'évaluation du prédécesseur de zéro. En effet, LaML ne permet pas de générer une erreur (comme les exceptions en ML). L'interpréteur simule ce cas en entrant dans une boucle infinie. Comme cette boucle ne rend jamais de résultat, LaMix a trouvé que le résultat de cet appel est $\perp$ et la phase de spécialisation a remplacé cet appel par un appel à la fonction `bottom`.

- `run_c_0` correspond à `run_c` pour la commande `while` du programme TP d'addition.

- `ADD1` correspond à la fonction originale `ADD1`.

Nous pouvons émettre quelques critiques par rapport aux buts que nous nous étions fixés :

- Les environnements ont été remplacés par des structures de données plus simples. Toutefois, LaMix n'a pas complètement remplacé ces structures par des produits puisqu'il reste les quatre constructeurs `CVarEnv_A` à `CVarEnv_D`. Si nous analysons en détail l'utilisation des valeurs construites, d'une part avec `CVarEnv_A` et `CVarEnv_B` et d'autre part avec `CVarEnv_C` et `CVarEnv_D`, nous nous apercevons que les expressions sont identiques. En effet, les branches des conditionnelles des clones de `get` et `upd` sont syntaxiquement égales. Par exemples, les deux branches de `get_1` sont `fst d`. Par conséquent, LaMix aurait pu fusionné les deux classes de clones correspondant à `CVarEnv_C` et `CVarEnv_D`. Cela aurait permis d'éliminer ces constructeurs et de rendre le programme résiduel plus efficace.

- Les fonctions `get_0` et `get_2` d'une part et `get_1` et `get_3` sont identiques. LaMix a différencié inutilement ces fonctions rendant un programme plus gros que nécessaire.

- Certaines optimisations locales auraient pu simplifier certaines expressions. Ainsi, dans `ADD1`, le produit `let reads = (x,y)` est utilisé trois fois dans cette fonction : `fst reads`, et deux `snd reads`. Nous aurions pu remplacer ces trois expressions par une référence directe aux variables `x` et `y`. De plus la paire `(env,get_0 env)` est construite alors que seule sa seconde composante est utilisée. Une version plus efficace de `ADD1` serait donc :

```
let ADD1 x y = get_0(run_c_0(CVarEnv_D(x,(y,y)))) ;;
```

Un programme où nous aurions tenu compte de ces remarques serait le programme suivant :

```
let rec bottom x = bottom x ;;

let rec run_e_0 env = match fst env with Z -> bottom() | S v -> v ;;

let rec run_c_0 env = match fst env with
     S _ -> run_c_0(let env = ((run_e_0 env),snd env)
                    in (fst env,(fst(snd env),S(snd(snd env)))))
   | Z -> env ;;

let rec ADD1 x y = snd(snd(run_c_0(x,(y,y)))) ;;
```

Cette version plus optimale pourrait encore être améliorée.

- Les fonction pourraient être curryfiées. Les parties des produits provenant des environnements seraient ainsi directement accessibles (et non plus en utilisant des projections).

- Le test de `run_e_0` est inutile car la fonction `run_c_0` testait déjà la même valeur. Comme `run_e_0` n'est exécuté que dans le cas où le constructeur est `S`, la branche `Z -> bottom()` ne peut jamais être empruntée. En fait, les deux structures de contrôle de `run_c_0` et `run_e_0` pourraient être fusionnées.
- La valeur associée à la variable `y` du programme TP n'est jamais utilisée. Une analyse des valeurs réellement utilisées par le programme pourrait découvrir cela et décider que cette valeur ne doit pas apparaître. De même la valeur correspondant à `x` dans le résultat de `run_c_0` n'est pas utilisé. Nous pourrions renvoyer uniquement la valeur correspondant à `z`.
- Les fonctions `ADD1` et `run_c_0` pourraient fusionner.

Ces remarques déboucheraient sur le programme optimal :

```
let rec ADD1 x z = match x with S d -> ADD1 d (S z) | Z -> z ;;
```

Ces différentes remarques nous ont conduit à trouver des solutions à ces différents problèmes. Ainsi, nous pensons résoudre le problème de la présence des constructeurs `CVarEnv_A` ... `CVarEnv_D` en modifiant la phase de fusion/généralisation. Certaines classes de constructeurs seront forcées à fusionner lorsqu'elles décrivent des valeurs identiques (comme c'est le cas pour les constructeurs ci-dessus). Un phase d'optimisations locales pourra être ajoutée pour résoudre les problèmes de référence à des composantes de produits. Une analyse de l'utilisation des valeurs calculées pourraient améliorer les programmes résiduels (comme dans [Bec94]). Le problème de la fusion de structures de contrôle semble délicate à mettre en œuvre facilement. Cela reste, toutefois, un objectif important de nos travaux futures.

Ainsi, cet exemple nous a permis de constater que les objectifs initiaux (disparition des environnements et spécialisation de types) n'ont pas été complètement atteints bien que le programme résiduel ait réussi à simplifier ces structures.

Spécialisation avec une procédure récursive

Le programme TP précédent utilise une boucle `while` pour calculer la somme de deux entiers. Nous pourrions aussi utiliser une procédure récursive :

```
program ADD3;
  procedure add;       (* x + y -> y *)
    begin
      if x then
        begin
          x := pred(x);
          add
          y := succ(y)
        end
    end;

  begin
    read(x);
    read(y);
    add;
    write(y)
  end
```

La fonction ADD3 utilise ce programme:

```
(* Example 3 : ADD3 : Nat -> Nat -> Nat *)
(*  Using a non-terminal recursive procedure *)
let x = N(E) ;;
let y = N(N(E)) ;;
let z = N(N(N(E))) ;;
let add = N(N(N(N(E)))) ;;

let proc_add3 =
  Block(
   CCom(If(Var x,
          Block(
            CCom(Let(x,Pred(Var x)),
            CCom(Proc add,
            CCom(Let(y,Succ(Var y)),
            NCom)))),
          Block(NCom) ),
    NCom)) ;;

let proc_body3 =
  Block(
   CCom(Read x,
   CCom(Read y,
   CCom(Proc add,
   CCom(Write(Var y),
   NCom))))) ;;

let prog3 = (CDecl(add,proc_add3,
              NDecl),proc_body3) ;;

let ADD3 x y = match exec prog3 (CNat(x, CNat(y, NNat))) with
                  CNat w -> fst w
                | NNat   -> error() ;;
```

LaMix renvoie un programme résiduel très similaire au programme obtenu avec une boucle while:

```
type Bool = True | False
and Ident = N of Ident | E
and Nat = S of Nat | Z
and NatList = CNat of (Nat * NatList) | NNat
and VarEnv = CVarEnv of (Ident * Nat * VarEnv) | NVarEnv
and Expr = Zero | Var of Ident | Succ of Expr | Pred of Expr
and Com = Block of ComList | If of (Expr * Com * Com)
 | While of (Expr * Com) | Let of (Ident * Expr)
 | Proc of Ident | Read of Ident | Write of Expr
 and ComList = CCom of (Com * ComList) | NCom
and DeclList = CDecl of (Ident * Com * DeclList) | NDecl ;;
```

```
type VarEnv_0 =
   CVarEnv_D of (Nat * Nat)
 | CVarEnv_C of (Nat * Nat)
 | CVarEnv_B of (Nat * Nat)
 | CVarEnv_A of (Nat * Nat) ;;

let rec bottom x = bottom x ;;

let rec get_0 vars = match vars with
     CVarEnv_D d -> snd d | CVarEnv_C d -> snd d
   | CVarEnv_B d -> snd d | CVarEnv_A d -> snd d ;;

let rec get_1 vars = match vars with
     CVarEnv_D d -> fst d | CVarEnv_C d -> fst d ;;

let rec get_2 vars = match vars with
     CVarEnv_D d -> snd d | CVarEnv_C d -> snd d
   | CVarEnv_B d -> snd d | CVarEnv_A d -> snd d ;;

let rec get_3 vars = match vars with
     CVarEnv_D d -> fst d | CVarEnv_C d -> fst d ;;

let rec run_e_0 env = match get_3 env with Z -> bottom() | S v -> v ;;

let rec upd_0 n vars = match vars with
     CVarEnv_D d -> CVarEnv_C(n,snd d)
   | CVarEnv_C d -> CVarEnv_C(n,snd d) ;;

let rec upd_1 n vars = match vars with
     CVarEnv_D d -> CVarEnv_B(fst d,n)
   | CVarEnv_C d -> CVarEnv_A(fst d,n)
   | CVarEnv_B d -> CVarEnv_B(fst d,n)
   | CVarEnv_A d -> CVarEnv_A(fst d,n) ;;

let rec run_c_0 env = match get_1 env with
     S _ -> let env = run_c_0(upd_0 (run_e_0 env) env)
            in upd_1 (S(get_2 env)) env
   | Z -> env ;;

let rec ADD3 x y = snd(let env = run_c_0(CVarEnv_D(x,y))
                       in (env,get_0 env)) ;;
```

Dans ce programme, seules deux variables composent l'environnement (`x` et `y`). De plus, le premier programme est récursif terminal tandis que celui-ci ne l'est pas. Cela provient du fait que dans le premier programme, la variable `z` est modifiée avant l'appel récursif tandis que celui-ci la modifie au retour de cet appel. Les types `VarEnv_0` et `VarEnv_1` du premier programme résiduel forment ici un seul type `VarEnv_0`. Nous retrouvons les mêmes critiques que dans le programme d'addition précédent.

Une version complètement optimisé donnerait la définition suivante :

```
let rec ADD3 x z = match x with S d -> S(ADD3 d z) | Z -> z ;;
```

C.3.2 Interpréteur avec continuations

Lors du chapitre introduisant le clonage, nous avons souligner les problèmes rencontrés par les évaluateurs partiels pour gérer correctement les structures comme les environnements. Deux alternatives se présentaient. Soit le programme est mal spécialisé (comme la version de l'interpréteur standard) car les structures partiellement connues sont renvoyées par des structures de contrôle résiduelles (comme celle de la fonction `run_c_0` dans les deux exemples ci-dessus). Soit nous utilisons une version du programme à spécialiser avec continuations où ces structures sont bien gérer mais où nous avons des problèmes de terminaison de l'évaluation partielle.

Avec LaMix, nous n'avons pas de problème de terminaison puisque les algorithmes terminent toujours. Nous pouvons nous demander si une version de l'interpréteur TP dans le style des continuations pourrait améliorer les programmes résiduels obtenus.

Voici le programme permettant d'interpréter un programme TP en utilisant les continuations :

```
(* Tiny Pascal : a small pascal interpretor *)
(* Version using continuation passing style *)

type Bool = True | False ;;

(* Identifiers are unary numbers *)
type Ident =  N of Ident | E ;;

(* Nat == unary numbers *)
type Nat = S of Nat | Z ;;

(* List of numbers *)
type NatList = CNat of Nat * NatList | NNat ;;

(* Environments for identifiers *)
type VarEnv = CVarEnv of Ident * Nat * VarEnv | NVarEnv ;;

(* Expressions *)
type Expr = Zero
          | Var of Ident
          | Succ of Expr
          | Pred of Expr ;;

(* Commands *)
type Com = Block of ComList
         | If of Expr * Com * Com
         | While of Expr * Com
         | Let of Ident * Expr
         | Proc of Ident
         | Read of Ident
         | Write of Expr
```

```
(* Lists of commands *)
and ComList = CCom of Com * ComList | NCom

(* Lists of procedures definitions *)
and DeclList = CDecl of Ident * Com * DeclList | NDecl ;;

let fst3 (x,y,z) = x
and snd3 (x,y,z) = y
and trd3 (x,y,z) = z ;;

(* Error : loop forever *)
let rec error x = error x ;;

(* equal on identifiers *)
let rec eq_Ident x y = match x with
     N xx -> (match y with N yy -> eq_Ident xx yy | E -> False)
   | E    -> (match y with N yy -> False          | E -> True ) ;;

(* Find the definition of a procedure *)
let rec get_p pid defs = match defs with
     CDecl d -> (match eq_Ident pid (fst3 d) with
                    True  -> (snd3 d)
                  | False -> get_p pid (trd3 d))
   | NDecl   -> error() ;;

(* Find the definition of a procedure from the program *)
let get_proc_com pid prog = get_p pid (fst prog) ;;

(* Creat an empty environment *)
let make_empty_env reads = (NVarEnv,reads,NNat) ;;

(* Get the next read value *)
let get_read_val env = match snd3 env with
    CNat v -> fst v
  | NNat   -> error() ;;

(* Pop the first read value *)
let pop_read_vals env = match snd3 env with
    CNat v -> (fst3 env,snd v,trd3 env)
  | NNat   -> error() ;;

(* Push a value in the write buffer *)
let push_write_val v env = (fst3 env,snd3 env,CNat(v,trd3 env)) ;;

(* Get the write buffer from an environment *)
let get_writes env = trd3 env ;;
```

```
(* Update the value associated to an identifier *)
let rec upd vid n vars = match vars with
     CVarEnv d -> (match eq_Ident vid (fst3 d) with
                     True  -> CVarEnv(vid,n,(trd3 d))
                   | False -> CVarEnv(fst3 d,snd3 d,upd vid n (trd3 d)))
   | NVarEnv   -> CVarEnv(vid,n,NVarEnv) ;;

(* Update the value associated to an identifier in an environment *)
let update_var vid n env = (upd vid n (fst3 env),snd3 env,trd3 env) ;;

(* Get the value associated to an identifier *)
let rec get id vars = match vars with
     CVarEnv d -> (match eq_Ident id (fst3 d) with
                     True  -> (snd3 d)
                   | False -> get id (trd3 d))
   | NVarEnv   -> error() ;;

(* Get the value associated to an identifier from an environment *)
let get_var id env = get id (fst3 env) ;;

(* Concatenate two lists of commands *)
let rec append_com x y = match x with
     CCom c  -> CCom(fst c,append_com (snd c) y)
   | NCom    -> y ;;

(* Compute an expression with an environment *)
let rec run_e expr env = match expr with
     Zero   -> Z
   | Var v  -> get_var v env
   | Succ e -> S(run_e e env)
   | Pred e -> (match run_e e env with
                   Z   -> error()
                 | S v -> v) ;;

(* Execute a command "com" into a given environment *)
(* then, execute the reste of commands "coms" *)
let rec run_c prog com coms env = match com with
     Block c -> run_cs prog (append_com c coms) env
   | If c    -> (match run_e (fst3 c) env with
                   S _ -> run_cs prog (CCom(snd3 c,coms)) env
                 | Z   -> run_cs prog (CCom(trd3 c,coms)) env)
   | While c -> (match run_e (fst c) env with
                   S _ -> run_cs prog (CCom(snd c,CCom(com,coms))) env
                 | Z   -> run_cs prog coms env)
   | Let c   -> run_cs prog coms (update_var (fst c) (run_e (snd c) env) env)
   | Proc c  -> run_cs prog (CCom(get_proc_com c prog,coms)) env
   | Read c  -> run_cs prog coms (update_var c (get_read_val env)
                                               (pop_read_vals env))
```

```
   | Write c -> run_cs prog coms (push_write_val (run_e c env) env)

(* Execute a list of commands (the continuation) *)
and run_cs prog coms env = match coms with
     CCom c  -> run_c prog (fst c) (snd c) env
   | NCom    -> get_writes env ;;

(* Execute a program with a read list of values *)
(* an return the write buffer *)
(* A program is a list of procedure definition and a command *)
let exec prog reads =
   let env = make_empty_env reads
    in run_cs prog (CCom(snd prog,NCom)) env ;;
```

Ici, l'exécution d'une commande est suivie par l'exécution du reste des commandes en ajoutant un argument supplémentaire à la fonction `run_c`.

Spécialisation d'une boucle `while`

Nous pouvons reprendre l'exemple du calcule de la somme de deux entiers unaires avec le programme TP suivant :

```
program ADD1;
  procedure add;       (* x + y -> z *)
    begin
      z := y;
      while x do
        begin
          x := pred(x);
          z := succ(z)
        end
    end;

  begin
    read(x);
    read(y);
    add;
    write(z)
  end
```

La fonction `ADD1` permet de lancer l'interpréteur TP avec le programme ci-dessus :

```
(* Example 1 : ADD1 : Nat -> Nat -> Nat *)
(*  Using a while loop *)

let x = N(E) ;;
let y = N(N(E)) ;;
let z = N(N(N(E))) ;;
let add = N(N(N(N(E)))) ;;
```

```
let proc_add1 =
  Block(
   CCom(Let(z,Var y),
   CCom(While(Var x,
          Block(
            CCom(Let(x,Pred(Var x)),
            CCom(Let(z,Succ(Var z)),
            NCom))) ),
   NCom))) ;;

let proc_body1 =
  Block(
   CCom(Read x,
   CCom(Read y,
   CCom(Proc add,
   CCom(Write(Var z),
   NCom))))) ;;

let prog1 = (CDecl(add,proc_add1,
              NDecl),proc_body1) ;;

let ADD1 x y = match exec prog1 (CNat(x, CNat(y, NNat))) with
                  CNat w -> fst w
                | NNat   -> error() ;;
```

Le programme résiduel obtenu pour la spécialisation de `ADD1` est le suivant :

```
type Bool = True | False
and Ident = N of Ident | E
and Nat = S of Nat | Z
and NatList = CNat of (Nat * NatList) | NNat
and VarEnv = CVarEnv of (Ident * Nat * VarEnv) | NVarEnv
and Expr = Zero | Var of Ident | Succ of Expr | Pred of Expr
and Com = Block of ComList | If of (Expr * Com * Com)
 | While of (Expr * Com) | Let of (Ident * Expr)
 | Proc of Ident | Read of Ident | Write of Expr
and ComList = CCom of (Com * ComList) | NCom
and DeclList = CDecl of (Ident * Com * DeclList) | NDecl ;;

type NatList_0 = CNat_B of Nat | CNat_A of Nat ;;

let rec bottom x = bottom x ;;

let rec run_e_0 env = match fst env with Z -> bottom() | S v -> v ;;

let rec run_c_0 env = match fst env with
     S _ -> let env = (run_e_0 env,snd env)
             in run_c_0(fst env,(fst(snd env),S(snd(snd env))))
   | Z -> snd(env,CNat_A(snd(snd env))) ;;
```

```
let rec run_e_1 env = match fst env with Z -> bottom() | S v -> v ;;

let rec run_c_1 env = match fst env with
     S _ -> let env = (run_e_1 env,snd env)
            in run_c_0(fst env,(fst(snd env),S(snd(snd env))))
   | Z -> snd(env,CNat_B(snd(snd env))) ;;

let rec ADD1 x y = match let reads = (x,y)
                         in run_c_1(fst reads,(snd reads,snd reads)) with
                  CNat_B w -> w | CNat_A w -> w ;;
```

Ce programme a réussi à éliminer les clones de constructeur de `CVarEnv` qui étaient présents dans les programmes résiduels précédents. Par contre, nous retrouvons le même problème de structures de données équivalentes avec les constructeurs `CNat_A` et `CNat_B`.

Un problème qui n'existait pas avec les deux programmes résiduels précédents est apparu. Les fonctions `run_c_1` et `run_c_0` pourraient être fusionnées. La fonction `run_c_1` correspond à la première fois que le programme exécute la boucle `while` et `run_c_0` aux étapes suivantes. Ce problème [WCRS91] se retrouve souvent dans les évaluateurs partiels *en ligne* en comparaison avec les version *hors ligne*. Nous pourrions ajouter un mécanisme à LaMix pour détecter ce genre de situations.

Spécialisation d'une procédure récursive

Nous pouvons aussi spécialiser le second programme TP d'addition qui utilise une procédure récursive non-terminale :

```
program ADD3;
  procedure add;       (* x + y -> y *)
    begin
      if x then
        begin
          x := pred(x);
          add
          y := succ(y)
        end
    end;

  begin
    read(x);
    read(y);
    add;
    write(y)
  end
```

La fonction `ADD3` à spécialiser est définie par :

```
(* Example 3 : ADD3 : Nat -> Nat -> Nat *)
(*  Using a non-terminal recursive procedure *)
```

```
let x = N(E) ;;
let y = N(N(E)) ;;
let z = N(N(N(E))) ;;
let add = N(N(N(N(E)))) ;;

let proc_add3 =
  Block(
   CCom(If(Var x,
          Block(
            CCom(Let(x,Pred(Var x)),
            CCom(Proc add,
            CCom(Let(y,Succ(Var y)),
            NCom)))),
          Block(NCom) ),
   NCom)) ;;

let proc_body3 =
  Block(
   CCom(Read x,
   CCom(Read y,
   CCom(Proc add,
   CCom(Write(Var y),
   NCom))))) ;;

let prog3 = (CDecl(add,proc_add3,
              NDecl),proc_body3) ;;

let ADD3 x y = match exec prog3 (CNat(x, CNat(y, NNat))) with
                  CNat w -> fst w
                | NNat   -> error() ;;
```

Le programme résiduel obtenu est assez intéressant car il introduit une spécialisation utile d'un type de donnée :

```
type Bool = True | False
and Ident = N of Ident | E
and Nat = S of Nat | Z
and NatList = CNat of (Nat * NatList) | NNat
and VarEnv = CVarEnv of (Ident * Nat * VarEnv) | NVarEnv
and Expr = Zero | Var of Ident | Succ of Expr | Pred of Expr
and Com = Block of ComList | If of (Expr * Com * Com)
 | While of (Expr * Com) | Let of (Ident * Expr)
 | Proc of Ident | Read of Ident | Write of Expr
and ComList = CCom of (Com * ComList) | NCom
and DeclList = CDecl of (Ident * Com * DeclList) | NDecl ;;

type ComList_0 = CCom_B | CCom_A of ComList_0
and NatList_0 = CNat_B of Nat | CNat_A of Nat
and VarEnv_0 = CVarEnv_C of (Nat * Nat)
```

```
              | CVarEnv_B of (Nat * Nat)
              | CVarEnv_A of (Nat * Nat) ;;

let rec bottom x = bottom x ;;

let rec get_0 vars = match vars with
     CVarEnv_C d -> fst d | CVarEnv_B d -> fst d ;;

let rec get_1 vars = match vars with
     CVarEnv_C d -> snd d | CVarEnv_B d -> snd d | CVarEnv_A d -> snd d ;;

let rec get_2 vars = match vars with
     CVarEnv_C d -> snd d | CVarEnv_B d -> snd d | CVarEnv_A d -> snd d ;;

let rec upd_0 n vars = match vars with
     CVarEnv_C d -> (fst d,n)
   | CVarEnv_B d -> (fst d,n)
   | CVarEnv_A d -> (fst d,n) ;;

let rec upd_1 n vars = match vars with
     CVarEnv_C d -> CVarEnv_A(fst d,n)
   | CVarEnv_B d -> CVarEnv_A(fst d,n)
   | CVarEnv_A d -> CVarEnv_A(fst d,n) ;;

let rec run_cs_0 coms env = match coms with
     CCom_B -> let env = upd_0 (S(get_1 env)) env
                in snd(env,CNat_A(snd env))
   | CCom_A c -> run_cs_0 c (upd_1 (S(get_2 env)) env) ;;

let rec get_3 vars = match vars with
     CVarEnv_C d -> fst d | CVarEnv_B d -> fst d ;;

let rec run_e_0 env = match get_3 env with Z -> bottom() | S v -> v ;;

let rec upd_2 n vars = match vars with
     CVarEnv_C d -> CVarEnv_B(n,snd d) | CVarEnv_B d -> CVarEnv_B(n,snd d) ;;

let rec run_c_0 coms env = match get_0 env with
     S _ -> run_c_0 (CCom_A coms) (upd_2 (run_e_0 env) env)
   | Z -> run_cs_0 coms env ;;

let rec run_e_1 env = match fst env with Z -> bottom() | S v -> v ;;

let rec run_c_1 env = match fst env with
     S _ -> run_c_0 CCom_B (CVarEnv_C(run_e_1 env,snd env))
   | Z -> snd(env,CNat_B(snd env)) ;;

let rec ADD3 x y = match run_c_1(x,y) with CNat_B w -> w | CNat_A w -> w ;;
```

Dans ce programme, nous retrouvons le même problème de profusion inutile de clones de constructeurs (`CVarEnv_A`, `CVarEnv_B` et `CVarEnv_C` d'une part et `CNat_A` et `CNat_B` d'autre part). Par contre, le nouveau type de données `ComList_0` a été créé pour les deux types de continuations possibles. Le constructeur `CCom_B` correspond à une continuation composée d'une seule instruction `y := succ(y);` tandis que `CCom_A` à une continuation comportant l'instruction `y := succ(y);` suivie de ce même type de continuation. Ici, ce type de données est utilisé par la fonction `run_cs_0` pour exécuter toutes les instructions `y := succ(y);` qui attendent d'être exécutées après la fin de la première boucle (celle de `run_c_0`).

Une version optimisée de ce programme donnerait la définition suivante :

```
type ComList_0 = CCom_B | CCom_A of ComList_0 ;;

let rec run_cs_0 coms y = match coms with
     CCom_B -> S y | CCom_A c -> run_cs_0 c (S y) ;;

let rec run_c_0 coms x y = match x with
     S d -> run_c_0 (CCom_A coms) d y | Z -> run_cs_0 coms y ;;

let rec ADD x y = match x with
     S d -> run_c_0 CCom_B d y | Z -> y ;;
```

Cette version montre clairement les deux boucles du programme résiduel. La première, celle de `run_c_0`, construit une suite de `CCom_A` suivie d'un `CCom_B` dont la longueur est égale à la longueur de l'entier `x` moins un. La seconde boucle parcourt cette liste et ajoute un successeur à `y` pour chaque élément de la suite de `CCom_A` plus un pour `CCom_B`. Nous retrouvons ici la caractéristique du style par continuation où la pile des appels récursifs est remplacée par une gestion explicite d'accumulateurs. Ce mauvais résultat (tout relatif) provient plus de l'utilisation des continuations et de la manière dont le programme TP est écrit que du mécanisme d'évaluation partielle qui reste une méthode de transformation de programme non universelle. L'utilisation d'une technique de déforestation [Wad88, FW88] permettrait d'obtenir un programme optimal.

C.3.3 Banc d'essai

Des comparaisons des temps d'exécution des programmes sources et résiduels ont été effectuées sur les deux interpréteurs TP : `tp.ml` est le programme standard tandis que `tpc.ml` utilise les continuations. Les données statiques spécialisant ces programmes correspondent à trois programmes TP différents pour calculer la somme de deux entiers unaires : `ADD1` utilise une boucle `while`, `ADD2` un appel récursif terminal et `ADD3` un appel récursif non-terminal. Les arguments dynamiques prennent

les valeurs 2, 10 et 50 :

Interpréteur	Fonction spécialisée	Donnée dynamique	Programme source (secondes)	Programme résiduel (secondes)	Gain
`tp.ml`	`ADD1 x x`	2	9.0	0.5	18.0
`tp.ml`	`ADD1 x x`	10	29.3	1.8	16.3
`tp.ml`	`ADD1 x x`	50	130.6	8.3	15.7
`tp.ml`	`ADD2 x x`	2	7.7	0.4	19.3
`tp.ml`	`ADD2 x x`	10	27.8	1.6	17.4
`tp.ml`	`ADD2 x x`	50	128.8	7.6	16.9
`tp.ml`	`ADD3 x x`	2	7.7	0.4	19.3
`tp.ml`	`ADD3 x x`	10	28.0	1.7	16.5
`tp.ml`	`ADD3 x x`	50	134.5	8.0	16.8
`tpc.ml`	`ADD1 x x`	2	9.8	0.3	32.7
`tpc.ml`	`ADD1 x x`	10	31.8	1.0	31.8
`tpc.ml`	`ADD1 x x`	50	141.6	4.5	31.5
`tpc.ml`	`ADD2 x x`	2	8.8	0.4	22.0
`tpc.ml`	`ADD2 x x`	10	31.6	1.6	19.8
`tpc.ml`	`ADD2 x x`	50	145.0	7.9	18.4
`tpc.ml`	`ADD3 x x`	2	8.8	0.4	22.0
`tpc.ml`	`ADD3 x x`	10	31.5	1.9	16.6
`tpc.ml`	`ADD3 x x`	50	145.5	9.6	15.2

Le gain pour l'interpréteur standard va de $15,7$ à $17,4$ pour la somme de 50 et 50. Il est de l'ordre de 16 et est relativement homogène pour les trois programmes TP.

Pour l'interpréteur avec continuations, ce gain varie entre $15,2$ et $31,5$ pour la somme de 50 et 50. Le très bon résultat obtenu en spécialisant le premier programme TP (utilisant une boucle `while`) s'explique par le fait que la structure des environnements a complètement disparu alors que les autres programmes résiduels n'ont pas réussi à les éliminer entièrement.

En fait, les résultats de la spécialisation des deux interpréteurs donnent un gain de l'ordre de 16 pour tous les programmes résiduels où l'environnement n'a pas été complètement éliminé et de l'ordre de 32 lorsqu'il a disparu.

C.4 Spécialisation d'un analyseur syntaxique

Ce dernier exemple décrit la spécialisation d'un programme qui teste si une suite de caractères appartient à une grammaire hors-contexte. Ce programme provient de l'article [Mog93].

La fonction `parse` prend pour arguments la description d'une grammaire hors-contexte et une liste de symboles. Elle renvoie `True` lorsque la liste appartient au langage décrit par la grammaire et `False` dans le cas contraire. Le programme définissant `parse` est :

```
(* Parser of a list of symbols using a CF grammar *)

type Bool = True | False ;;

(* Symbols = unary numbers *)
type Ident =  N of Ident | E ;;

(* Lists of symbols *)
type IdentList = CIdent of Ident * IdentList | NIdent ;;
```

```
(* Grammars *)
type Gram = EMPTY
          | REC
          | SYMB of Ident
          | SEQ of Gram * Gram
          | ALT of Gram * Gram ;;

(* List of parsing commands *)
type ToDo = DONE
          | DO of Gram * ToDo ;;

(* Equality of symbols *)
let rec eq_ident x y = match x with
     N xx -> (match y with N yy -> eq_ident xx yy | E -> False)
   | E    -> (match y with N yy -> False          | E -> True ) ;;

(* Parse the stream using a grammar expression *)
(* g = the current grammar expression *)
(* s = the input stream *)
(* g0 = the global grammar (used by recursion *)
(* todo = the rest of the parsing job *)
let rec p g s g0 todo = match g with
      EMPTY -> nowdo todo s g0
    | REC   -> p g0 s g0 todo
    | SYMB v -> (match s with
                   CIdent p -> (match eq_ident v (fst p) with
                                    True  -> nowdo todo (snd p) g0
                                  | False -> False)
                 | NIdent -> False)
    | SEQ v -> p (fst v) s g0 (DO(snd v,todo))
    | ALT v -> (match p (fst v) s g0 todo with
                  True  -> True
                | False -> p (snd v) s g0 todo)

(* Continue the parsing job *)
and nowdo todo s g = match todo with
     DONE -> (match s with
                 CIdent _ -> False
               | NIdent   -> True)
   | DO v -> p (fst v) s g (snd v) ;;

(* Parse a stream using a grammar *)
let parse g s = p g s g DONE ;;
```

Les grammaires reconnues par cet analyseur syntaxique sont construites avec les cinq schémas suivants :

- `EMPTY` : la grammaire vide.
- `SYMB(s)` : le symbole `s` (un symbole terminal).

- `SEQ(g1,g2)` : la concaténation de deux langages décrits par `g1` et `g2`.
- `ALT(g1,g2)` : l'alternative entre deux langages décrits par `g1` et `g2`.
- `REC` : une référence récursive à la grammaire entière.

Ainsi, le grammaire définie par l'expression :

```
ALT(SEQ(SYMB("("),SEQ(REC,SEQ(SYMB(")"),REC))),EMPTY)
```

décrit le langage des expressions bien parenthésées (composées uniquement des deux symboles (et)).

C.4.1 Spécialisation de `parse`

Avec la grammaire ci-dessus, nous pouvons spécialiser l'analyseur syntaxique en définissant la fonction `is_well_balanced` :

```
(* Grammar for well balanced expressions *)
(* N E == "("   N N E == ")" *)
let g = ALT(SEQ(SYMB(N E),SEQ(REC,SEQ(SYMB(N(N E)),REC))),EMPTY) ;;

(* Check if a stream is well balanced *)
let is_well_balanced s = parse g s ;;
```

Comme LaML ne dispose pas du type des caractères, nous avons remplacé les symboles par des entiers unaires. Le programme résiduel retourné par LaMix est :

```
type Bool = True | False
and Ident = N of Ident | E
and IdentList = CIdent of (Ident * IdentList) | NIdent
and Gram = EMPTY | REC | SYMB of Ident
 | SEQ of (Gram * Gram) | ALT of (Gram * Gram)
and ToDo = DONE | DO of (Gram * ToDo) ;;

type Bool_1 = False_B | True_B
and Bool_0 = False_A | True_A
and ToDo_0 = DO_A of ToDo_0 | DONE_A ;;

let rec eq_ident_0 y = match y with N yy -> False_B | E -> True_B ;;

let rec eq_ident_1 y = match y with N yy -> eq_ident_0 yy | E -> False_B ;;

let rec eq_ident_2 y = match y with N yy -> False_A | E -> True_A ;;

let rec eq_ident_3 y = match y with N yy -> eq_ident_2 yy | E -> False_A ;;

let rec eq_ident_4 y = match y with N yy -> eq_ident_3 yy | E -> False_A ;;

let rec nowdo_0 todo s = match todo with
     DONE_A -> (match s with CIdent _ -> False | NIdent -> True)
   | DO_A v -> p_0 s v
```

```
and p_1 s todo = match p_2 s todo with
     True -> True | False -> nowdo_0 todo s

and p_2 s todo = match s with
     CIdent p -> (match eq_ident_1(fst p) with
                     True_B -> p_1 (snd p) (DO_A todo) | False_B -> False)
   | NIdent -> False

and p_0 s todo = match s with
     CIdent p -> (match eq_ident_4(fst p) with
                     True_A -> p_1 (snd p) todo | False_A -> False)
   | NIdent -> False ;;

let rec is_well_balanced s = p_1 s DONE_A ;;
```

Ce programme résiduel semble à priori difficile à comprendre. La correspondance entre ses fonctions et le programme initial est la suivante :

- Les constructeurs `DO_A` et `DONE_A` correspondent à des spécialisations de `DO` et `DONE` pour des sous-grammaires de la grammaire initiale. Ils forment un type isomorphe aux entiers unaires (`DONE_A` pour le zéro et `DO_A` pour le successeur) et correspondent au nombre de parenthèses ouvrantes qui n'ont pas encore été fermées.

- La fonction `p_1` correspond à l'appel à `p` avec pour grammaire, la grammaire tout entière. Le paramètre `todo` correspond au nombre de parenthèses ouvrantes qui doivent être fermées. Elle correspond à l'appel initial de `is_well_balanced` ou bien à l'interprétation du constructeur `REC` (deux occurrences dans la grammaire des expressions bien parenthésées). L'appel à `p_1` dans `p_2` correspond au `REC` compris entre les deux parenthèses. Celui de `p_0`, au `REC` après la parenthèse fermante. Cette fonction renvoie `True` si le flux d'entrée correspond à une expression bien parenthésée avec un excès de parenthèses fermantes correspondant à l'argument `todo`.

- La fonction `p_2` reconnaît une parenthèse ouvrante dans le flux d'entrée, ajoute un élément sur la pile des parenthèses ouvrantes non fermées et appel récursivement `p_1`. Cette fonction renvoie `True` lorsque le flux d'entrée commence par une parenthèse ouvrante et que le reste correspond à une expression bien parenthésée avec un excès de parenthèses fermantes égal au successeur de l'argument `todo`.

- `nowdo_0` est appelé par `p_1` lorsque la première alternative de la grammaire ne peut pas être utilisée et que l'analyseur syntaxique tente d'utiliser la seconde (`EMPTY` qui est toujours valide). Il doit alors tenter de réduire la pile des parenthèses ouvrantes non fermées en appelant `p_0`.

- La fonction `p_0` reconnaît une parenthèse fermante puis appelle `p_1` (le deuxième `REC` de la grammaire).

- Les clones de `eq_ident` reconnaissent un caractère. `eq_ident_1` reconnaît une parenthèse ouvrante et `eq_ident_4`, une parenthèse fermante.

Le programme obtenu est tout à fait satisfaisant. Il a réussi à éliminer toute référence à la grammaire. Le type `ToDo_0` forme une spécialisation du type `ToDo` vraiment intéressante car ce type

est isomorphe aux entiers unaires et une valeur correspond exactement au nombre de parenthèses ouvrantes *en l'air*.

Pour ne pas compliquer la lisibilité du programme résiduel, nous n'avons pas déplié les fonctions comportant au moins une conditionnelle. En fait, toutes les fonctions sauf `is_well_balanced` et `p_1` pourraient être dépliées. Les types `Bool_0` et `Bool_1` ont été créés. Étant isomorphes aux booléen, LaMix aurait pu utiliser le type `Bool` à leur place.

Le programme résiduel ne ressemble pas à un programme que pourrait écrire un programmeur pour reconnaître le langage des expressions bien parenthésées. Cela est du au fait que l'évaluation partielle est une méthode qui ne modifie pas l'algorithme utilisé par un programme.

En comparaison avec les résultats de [Mog93], le programme résiduel produit par LaMix a été obtenu directement alors que dans l'article de Mogensen, l'analyseur syntaxique devait être modifié. Notamment, l'auteur devait introduire une opération explicite de généralisation du type `ToDo`. L'utilisation d'un évaluateur partiel en ligne et la gestion originale des clones (par la notion d'événement) permet de résoudre ce problème sans réécrire le programme à spécialiser.

C.4.2 Banc d'essai

Les essais ont été réalisés sur deux programmes d'analyse syntaxique. Le premier `gram.ml` utilise les constructeurs `True` et `False` dans les fonctions tandis que `gram_bis.ml` ajoute deux paramètres supplémentaires aux fonctions dont la valeur correspond à ces deux constructeurs. Cette modification permet d'empêcher la création d'une multitude de clones de ces deux constructeurs. Ces analyseurs syntaxiques ont été spécialisés avec deux grammaires : `well_balanced` correspond à la grammaire des mots composés de parenthèses et qui sont bien parenthésés et `if_then_else` correspond aux expressions conditionnelles composées d'atomes `v`, de `i ... t ...` et de `i ... t ... e ...` :

Analyseur syntaxique	Grammaire	Donnée dynamique	Programme source (secondes)	Programme résiduel (secondes)	Gain
`gram.ml`	`well_balanced`	()()	9.4	2.3	4.1
`gram.ml`	`well_balanced`	(((()(((()(())))))))	46.3	11.3	4.1
`gram.ml`	`well_balanced`	((((((((((()))))))(((((((((( ))))))))))(((((())))))))))	103.9	25.1	4.1
`gram_bis.ml`	`well_balanced`	()()	12.2	2.3	5.3
`gram_bis.ml`	`well_balanced`	(((()(((()(())))))))	60.0	11.3	5.3
`gram_bis.ml`	`well_balanced`	((((((((((()))))))(((((((((( ))))))))))(((((())))))))))	134.8	25.2	5.3
`gram_bis.ml`	`if_then_else`	ivtv	7.7	2.0	3.9
`gram_bis.ml`	`if_then_else`	iivtvevtivtvev	49.8	11.0	4.5
`gram_bis.ml`	`if_then_else`	iivtvevtivtveiivtvevtivtvev	98.4	21.8	4.5

C.5 Conclusion sur LaMix

Les quelques exemples d'utilisation de LaMix ont permis d'expérimenter les idées décrites dans ce travail. Nous avons pu voir que les objectifs consistant à gérer les environnements et à créer de nouveaux types dans les programmes interprétatifs ont été globalement atteints. La méthode de la gestion du clonage par la notion d'événement semble bien adaptée à ce type de programmes.

Pourtant, ces expérimentations ont aussi montré certaines insuffisances dans les mécanismes de fusion et de spécialisation. LaMix avait pour but d'essayer les concepts introduits ici et reste sujet à de nombreuses améliorations.

C.5.1 Problème de fusion

Ainsi, la profusion de clones de constructeur a parfois conduit LaMix a différencier des structures de données représentant la même valeur. Certaines structures de contrôle possèdent trop de branches et certains types spécialisés portent trop de clones de constructeur. Cela a parfois des effets néfastes lorsque le type spécialisé comporte plusieurs constructeurs alors que leur fusion pourrait générer un programme plus efficace. Ce cas s'est présenté pour la spécialisation des interpréteurs TP où le type `VarEnv_0` possédait deux constructeurs isomorphes :

```
type VarEnv_0 = CVarEnv_B of (Nat * (Nat * Nat))
              | CVarEnv_A of (Nat * (Nat * Nat)) ;;
```

Cela est aussi vrai dans l'exemple du programme de reconnaissance de motifs où plusieurs type spécialisés ont été introduits pour les booléens :

```
type Bool_2 = False_M | False_L | True_C
and Bool_1 = False_K | False_J | False_I | False_H | False_G
           | False_F | False_E | False_D | True_B
and Bool_0 = False_C | False_B | False_A | True_A ;;
```

Dans ce dernier cas, cela ne nuit pas à l'efficacité du programme résiduel mais rend les structures de contrôle plus imposantes et ralentit la phase de clonage.

Pour résoudre ce problème, nous pensons modifier la phase de fusion/généralisation. LaMix imposerait alors la fusion de clones de constructeur lorsqu'ils représenteraient la même valeur. Peut-être devrions nous aussi modifier la phase de clonage pour que cette opération de fusion ait lieu plus tôt dans le mécanisme d'évaluation partielle en imposant que certains clones de constructeur doivent activer la même branche spécialisée d'une conditionnelle (comme les clones de booléens ci-dessus).

C.5.2 Optimisations locales et analyse de l'utilisation des valeurs

Un certain nombre de transformations locales pourraient être appliquées pour rendre les programmes résiduels plus efficaces. Ainsi, un programme comme `fst(v1,v2)` pourrait se simplifier en `v1`. Ceci nécessite de modifier la phase de spécialisation.

Une phase supplémentaire analysant l'utilisation des valeurs qui sont créés pourrait aussi apporter quelques améliorations. Ainsi, avec la même expression que ci-dessus `fst(v1,v2)`, cette phase pourrait s'apercevoir que la seconde composante de la paire n'est jamais utilisée et donc qu'il n'est pas nécessaire de la calculer. Dans ce cas, si la variable `v2` ne sert qu'à créer cette paire, nous pouvons ne pas calculer sa valeur (par exemple si c'est une variable locale définie avec un `let`, ce code pourrait disparaître).

Ceci pourra aussi déboucher sur des programmes aussi bien traités que par la supercompilation [Tur79, Tur86] et le *calcul partiel généralisé* [FN88, FNT91, Tak91].

Bibliographie

[AJ89] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Sixteenth ACM Symp. on Principles of Programming Languages*, pages 293–302, New York, 1989. ACM Press.

[Aug85] Lennart Augustsson. Compiling lazy pattern-matching. In J.-P. Jouannaud, editor, *Conference on Functional Programming and Computer Architecture, LNCS 201*, pages 368–381. Springer-Verlag, 1985.

[Baw86] A. Bawden. Connection graphs. In *Proc. ACM Symp. on Lisp and Functional Programming*, pages 258–265, 1986.

[BD77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of A.C.M.*, 24(1):44–67, January 1977.

[BD91] A. Bondorf and O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[Bec90] D. Bechet. Les abréviations dans les réseaux d'interactions. Rapport de D.E.A., E.N.S. Paris, 1990.

[Bec91a] D. Bechet. Évaluation partielle dans les réseaux d'interactions. Rapport de magistère mmfai, E.N.S. Paris, 1991.

[Bec91b] D. Bechet. Problèmes d'évaluation partielle dans les réseaux d'interaction. Notes de travail, 1991.

[Bec92] D. Bechet. Partial evaluation of interaction nets. In M. Billaud et al., editors, *WSA '92, Static Analysis, Bordeaux, France, September 1992. Bigre vols 81–82, 1992*, pages 331–338. Rennes: IRISA, 1992.

[Bec93] D. Bechet. Partial evaluation and distributed systems. Draft, 1993.

[Bec94] D. Bechet. Spliting partial evaluation for a better support of partially static structures. Draft, 1994.

[BEJ87] D. Bjørner, A.P. Ershov, and N.D. Jones, editors. *Workshop Compendium, Workshop on Partial Evaluation and Mixed Computation, Gl. Avernæs, Denmark, October 1987.* Department of Computer Science, Technical University of Denmark, Lyngby, Denmark, 1987.

[BJ93] A. Bondorf and J. Jørgensen. Efficient analyses for realistic off-line partial evaluation. Technical Report 93/4, DIKU, University of Copenhagen, Denmark, 1993.

[Bon88] A. Bondorf. Towards a self-applicable partial evaluator for term rewriting systems. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 27–50. Amsterdam: North-Holland, 1988.

[Bon90] A. Bondorf. Automatic autoprojection of higher order recursive equations. In N.D. Jones, editor, *ESOP '90. 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990 (Lecture Notes in Computer Science, vol. 432)*, pages 70–87. Berlin: Springer-Verlag, May 1990. Revised version in [Bon91a].

[Bon91a] A. Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17:3–34, 1991.

[Bon91b] A. Bondorf. Similix manual, system version 4.0. Technical report, DIKU, University of Copenhagen, Denmark, 1991.

[Bul84] M.A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.

[CD91] C. Consel and O. Danvy. For a better support of static data flow. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pages 496–519. ACM, Berlin: Springer-Verlag, 1991.

[CD93] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Twentieth ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, January 1993*, pages 493–501. ACM, New York: ACM, 1993.

[CK92] C. Consel and S.C. Khoo. On-line & off-line partial evaluation: Semantics specifications and correctness proofs. Technical Report Research Repport 896, Yale University, New Haven, Connecticut, USA, 1992.

[Con88] C. Consel. New insights into partial evaluation: The Schism experiment. In H. Ganzinger, editor, *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 1988 (Lecture Notes in Computer Science, vol. 300)*, pages 236–246. Berlin: Springer-Verlag, 1988.

[Con90a] C. Consel. Binding time analysis for higher order untyped functional languages. In *1990 ACM Conference on Lisp and Functional Programming, Nice, France*, pages 264–272. New York: ACM, 1990.

[Con90b] Charles Consel. *The Schism Manual, Version 1.0.* Yale University, New Haven, Connecticut, December 1990.

[CR91] W. Clinger and J. Rees. Revised^4 report on the algorithmic language scheme. *LISP Pointers*, IV(3):1–55, 1991.

[Dan91] O. Danvy. Semantics-directed compilation of nonlinear patterns. *Information Processing Letters*, 37(6):315–322, March 1991.

[Fea86] M.S. Feather. A survey and classification of some program transformation techniques. In *Proc. TC2 IFIP Working Conference on Program Specification and Transformation, Bad-Tölz*, 1986.

[FN88] Y. Futamura and K. Nogi. Generalized partial computation. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 133–151. Amsterdam: North-Holland, 1988.

[FNT91] Y. Futamura, K. Nogi, and A. Takano. Essence of generalized partial computation. *Theoretical Computer Science*, 90(1):61–79, 1991. Also in D. Bjørner and V. Kotov: Images of Programming, North-Holland, 1991.

[Fut71] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[FW88] A. B. Ferguson and P. L. Wadler. When will deforestation stop? In *1988 Glasgow Workshop on Functionnal Programming*, pages 39–59, 1988.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GJ94a] R. Glück and J. Jørgensen. Generating optimizing specializers. In *IEEE Computer Society International Conference on Computer Languages, Toulouse, France, 1994*, pages 183–194. IEEE Computer Society Press, 1994.

[GJ94b] R. Glück and J. Jørgensen. Generating transformers for deforestation and supercompilation. In *International Static Analysis Symposium (SAS'94), LNCS 864*. Springer-Verlag, 1994.

[GK93] R. Glück and A.V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In P.Cousot, M.Falaschi, G.Filè, and A.Rauzy, editors, *3rd International Workshop on Static Analysis, Padova, Italy, September 1993. (Lecture Notes in Computer Science, vol. 724)*, pages 112–123. Berlin: Springer-Verlag, 1993.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1989.

[Gom90] C.K. Gomard. Partial type inference for untyped functional programs. In *1990 ACM Conference on Lisp and Functional Programming, Nice, France*, pages 282–287. New York: ACM, 1990.

[Har77] A. Haraldsson. *A Program Manipulation System Based on Partial Evaluation*. PhD thesis, Linköping University, Sweden, 1977. Linköping Studies in Science and Technology Dissertations 14.

[Hen91] F. Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pages 448–472. ACM, Berlin: Springer-Verlag, 1991.

[Hog81] C.J. Hogger. Derivation of logic programs. *Journal of the Association for Computing Machinery*, 2(28), apr 1981.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Englewood Cliffs, NJ: Prentice Hall, 1993.

[Joh85] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture, Nancy, France, September 1985 (Lecture Notes in Computer Science, vol. 201)*, pages 190–203. Berlin: Springer-Verlag, 1985.

[Jør91] J. Jørgensen. Generating a pattern matching compiler by partial evaluation. In S.L. Peyton Jones, G. Hutton, and C. Kehler Holst, editors, *Functional Programming, Glasgow 1990*, pages 177–195. Berlin: Springer-Verlag, 1991.

[Jør92] J. Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Nineteenth ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, January 1992*, pages 258–268. New York: ACM, 1992.

[JSS85] N.D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France. (Lecture Notes in Computer Science, vol. 202)*, pages 124–140. Berlin: Springer-Verlag, 1985.

[Kot80] L. Kott. *Des substitutions dans les systèmes d'équations algébriques sur le MAGMA. Application aux transformations de programmes et à leur correction.* PhD thesis, Université P. et M. Curie, Laboratoire informatique théorique et programmation, 1980.

[Laf90] Y. Lafont. Interaction nets. In *Proc. 17-th ACM Symp. on Principles of Programming Languages, San Francisco*, pages 95–108, January 1990.

[Laf91] Y. Lafont. The paradigm of interaction. Short version, 1991.

[Lau89] J. Launchbury. *Projection Factorisations in Partial Evaluation.* PhD thesis, Department of Computing, University of Glasgow, November 1989. Revised version in [Lau91a].

[Lau91a] J. Launchbury. *Projection Factorisations in Partial Evaluation.* Cambridge: Cambridge University Press, 1991.

[Lau91b] J. Launchbury. A strongly-typed self-applicable partial evaluator. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pages 145–164. ACM, Berlin: Springer-Verlag, 1991.

[Ler93] Xavier Leroy. *The Caml Light system, release 0.6. Documentation and user's manual.* INRIA, sep 1993. release 0.6.

[LR64] L.A. Lombardi and B. Raphael. Lisp as the language for an incremental computer. In E.C. Berkeley and D.G. Bobrow, editors, *The Programming Language Lisp: Its Operation and Applications*, pages 204–219. Cambridge, MA: MIT Press, 1964.

[MHD94] K. Malmkjær, N. Heintze, and O. Danvy. Ml partial evaluation using set-based analysis. In *1994 ACM SIGPLAN Workshop on ML and Its Applications, Orlando, Florida, June 1994 (Technical Report 2265, INRIA Rocquencourt, France)*, pages 112–119, 1994.

[Mog88] T. Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347. Amsterdam: North-Holland, 1988.

[Mog89] T. Mogensen. Binding time analysis for polymorphically typed higher order languages. In J. Diaz and F. Orejas, editors, *TAPSOFT '89. Proc. Int. Conf. Theory and Practice of Software Development, Barcelona, Spain, March 1989 (Lecture Notes in Computer Science, vol. 352)*, pages 298–312. Berlin: Springer-Verlag, 1989.

[Mog93] T. Mogensen. Constructor specialization. In *Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 22–32. New York: ACM, 1993.

[NN88] H.R. Nielson and F. Nielson. Automatic binding time analysis for a typed λ-calculus. *Science of Computer Programming*, 10:139–176, 1988.

[Plo75] G. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Ses88] P. Sestoft. Automatic call unfolding in a partial evaluator. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 485–506. Amsterdam: North-Holland, 1988.

[SGJ94] M.H. Sørensen, R. Glück, and N.D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In D. Sannella, editor, *Programming Languages and Systems — ESOP'94. 5th European Symposium on Programming, Edinburgh, U.K., April 1994 (Lecture Notes in Computer Science, vol. 788)*, pages 485–500. Berlin: Springer-Verlag, 1994.

[ST84] T. Sato and H. Tamaki. Unfold/fold transformation of logic programs. In *Proceedings of the Second International Logic Programming Conference, Uppsala*, pages 127–138. Sten-Åke Tärnlund, 1984.

[Tak91] A. Takano. Generalized partial computation for a lazy functional language. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 1–11. New York: ACM, 1991.

[Tur79] V.F. Turchin. A supercompiler system based on the language Refal. *SIGPLAN Notices*, 14(2):46–54, February 1979.

[Tur86] V.F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.

[Wad88] P. L. Wadler. Deforestation: Transforming programs to eliminate trees. In *European Symposium on Programming, LNCS 300*, pages 344–358. Springer-Verlag, 1988.

[WCRS91] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991 (Lecture Notes in Computer Science, vol. 523)*, pages 165–191. Berlin: Springer-Verlag, 1991.

[WL93] Pierre Weis and Xavier Leroy. *Le Langage CAML*. InterEditions, 1993.