No d'ordre

THESE de DOCTORAT de L'UNIVERSITE PARIS VI

spécialité

Systèmes Informatiques

présentée

par Monsieur *Vincent DORNIC*

pour obtenir le titre de **DOCTEUR DE L'UNIVERSITE PARIS VI**

Sujet de la thèse :

Analyse de Complexité des Programmes : Vérification et Inférence.

soutenue le *11 Juin 1992*

devant le jury composé de :

Monsieur B. Lorho	Président
Madame M. Soria	Rapporteur
Monsieur C. Consel	Rapporteur
Monsieur P. Feautrier	Examinateur
Monsieur D. K. Gifford	Examinateur
Monsieur C. Girault	Examinateur
Monsieur P. Jouvelot	Examinateur

ECOLE NATIONALE SUPERIEURE DES MINES DE PARIS
Rapport CRI/A/212

Hamburg
Bismarckstrasse 86
February 17, 1897

My dear Dr. Seidl,

[...]

The way in which I was inspired to do this is deeply significant and characteristic of the nature of artistic creation.

I had long contemplated bringing in the choir in the last movement, and only the fear that it would be taken as a formal imitation of Beethoven made me hesitate again and again. Then Bülow died, and I went to the memorial service. The mood in which I sat and pondered on the departed was utterly in the spirit of what I was working on the time. Then the choir, up in the organ-loft, intoned Klopstock's *Resurrection* chorale. It flashed on me like lightning, and everywhere became plain and clear in my mind! It was the flash that all creative artists wait for—conceiving by the holy Ghost!

What I then *experienced* had now to be expressed in sound. And yet—if I had not already born the work within me—how could I have had that experience? There were thousands of others sitting there in the church at the time! It is always the same with me: *Only when I experience something* do I compose, and only when composing do I experience! I know you will understand this without my enlarging on it. After all, a musician's nature can hardly be expressed in words. It would be easier to say what is *different* about him than about others. But what this difference is he himself would perhaps be least able to say. So it is the same too with his *aims*! He moves towards them like a sleepwalker—he does not know what road he is taking (perhaps past yawning abysses), but he heads towards the distant light, whether it be the ever-radian star or a seductive will-o'-the-wisp! ...

In sincere friendship,
Gustav Mahler.

Résumé

La programmation efficace des multi-processeurs suppose l'existence de méthodes de détection du parallélisme et surtout de mise en valeur du parallélisme utile. En effet, la gestion maximale du parallélisme peut s'avérer médiocre, par exemple si le ratio du temps d'exécution d'un processus distant sur le temps de communication et faible. Les méthodes d'équilibrage de charge utilisée jusqu'à présent sont empiriques, qu'elles soient utilisées à la compilation ou, comme c'est le plus fréquent, à l'exécution. Un besoin se fait donc sentir pour des méthode d'analyse de complexité des programmes.

L'analyse automatique de complexité est théoriquement limitée. En effet, l'arrêt d'un programme n'étant pas prédictible en général, le temps nécessaire au calcul l'est encore moins. Tout système automatique d'analyse est donc par définition incomplet. Ensuite, se posent les problèmes liés à l'expression des complexités. Quelle échelle de temps choisir? Sur quelle mesure des paramètres du programme doit on l'exprimer? Tous les systèmes précédents aspiraient à fournir des résultats précis en limitant le pouvoir d'expression du langage analysé. Même si l'intérêt théorique est indiscutable, ce type d'approche est peu approprié à la gestion du parallélisme, car inadapté aux langages réalistes.

Nous présentons des systèmes d'analyse automatique et statique basé sur le cadre formel du système d'effets. Introduit pour le langage *FX*, celui-ci permet d'associer aux expressions du langage des propriétés autres que les classiques types. Le langage *FX* étant conçu pour la parallélisation, les informations ajoutées concernent les régions mémoire dans lesquelles sont alloués les objets et les effets de bord éventuels lors de l'exécution des programmes. Pour notre étude, les informations concernent la complexité en temps des programmes. Il est utile d'ajouter que le langage *FX* est général car fusionne les paradigmes de la programmation applicative (fonctions d'ordre supérieur) et impérative.

Le parti pris pour la conception des systèmes présentés dans cette thèse est orthogonal aux précédents systèmes sur deux points : le langage analysé est un langage complet (applicatif et impératif) et les systèmes d'analyse sont prouvés cohérents avec le schéma d'évaluation du langage. Les informations de temps reconstruites sont simples et donc aisément manipulables par une module automatique d'équilibrage de charge. Enfin, preuve est faite de la flexibilité et de la puissance du système d'effet.

MOTS CLÉS : Analyse automatique de complexité, Induction de point fixe maximal, Langage applicatif et impératif, Polymorphisme, Système d'effet, Typage implicite et explicite.

Abstract

In parallel computing, automatic systems are required to identify useful parallelism, that is, the degree of task granularity that can be exploited efficiently in a given program.

Due to the halting problem, automatic complexity analysis is not decidable. Furthermore, in order to express time complexity, a time scale and a data measure are needed. All previous systems solve these problems by restricting the expressiveness of the language. As a result, those systems are not adapted to the automatic detection of useful parallelism.

We present three systems that use the notion of effect systems. Introduced for the FX language, this framework is general and flexible. It allows additional information to be associated with program expressions (e.g. side effects in FX and time in our thesis).

Our systems are conceptually new; the programming language is powerful and our systems are proved correct. The time information computed is adequate to support automatic detection of useful parallelism without programmer intervention.

KEYWORDS: Automatic complexity analysis, Maximal fixpoint induction, Functional and imperative languages, Polymorphism, Effect system, Implicit and explicit typing.

Remerciements

Une thèse n'est pas l'œuvre d'un seul homme. Il est naturel et juste de remercier chaleureusement toutes les personnes qui ont placé une pierre dans l'édifice. Dans le cas présent, les différentes pierres proviennent de :

Pierre Jouvelot *du CRI-ENSMP qui m'a encadré patiemment et qui m'a fait découvrir bien d'autres choses que l'informatique. Grand merci, car sans lui cette thèse ne serait pas.*

Prof. Paul Feautrier *du laboratoire MASI qui a été un directeur de thèse agréable et arrangeant.*

Prof. Michel Lenci, *directeur du CRI-ENSMP, qui m'a accepté dans son équipe avec une gentillesse inégalable et qui m'a fournit un financement pour ma dernière année.*

Gérard Memmi, *responsable de l'équipe PMacs de Bull-Les Clayes, qui m'a accueilli durant trois ans via un contrat CIFRE.*

Prof. Bernard Lorho, *qui à présidé mon jury.*

Prof. Michèle Soria *et* Charles Consel *qui ont accepté de rapporter avec célérité.*

Prof. David K. Gifford *qui fut un initiateur malgré lui de cette thèse et qui a accepté de venir du MIT pour ma soutenance.*

Prof. Claude Girault *qui a accepté de faire partie de mon jury.*

Yan Mei Tang *et* Jean-Pierre Talpin, *les deux membres de l'équipe BoP du CRI avec qui j'ai eu une riche et étroite collaboration.*

Dannie Durand, *qui m'a aidé pour la petite partie en langue Anglaise.*

Corinne Ancourt, François Irigoin, Rémi Triolet, Jacqueline Altimira, Steven Ericsson-Zenith *et tous les autres membres du CRI qui m'ont beaucoup aidé et soutenu.*

Babak Dehbonei, Nadia Tawbi *et* Fawzi Hassaine, *les membre de l'équipe PMacs de Bull, a qui je dois d'agréables heures passées en leur compagnie.*

Encore merci a tous.

Sommaire

Introduction

L'utilisation optimum des machines multi-processeurs suppose d'être à même de reconnaître le parallélisme potentiel des programmes (extraction du parallélisme) puis, parmi ce dernier d'isoler le sous-ensemble utile à une exécution rapide sur une architecture donnée (gestion du parallélisme utile). Effectuer ces deux étapes à la main n'est pas acceptable en général car requiert des programmeur une bonne connaissance de l'architecture cible et, de plus, produit des programmes peu portables d'une architecture vers une autre. La parallélisation automatique est un domaine à peu près maîtrisé mais réduit le plus souvent à la détection et à la gestion du parallélisme maximal. Pourtant l'exécution distante d'un processus très court apparaît plus pénalisante qu'efficace si les temps de communications sont longs. Les méthodes d'équilibrage de charge utilisées jusqu'à présent sont lw plus souvent empiriques. Il peut s'agir de *profiling* (On observe chaque exécution du programme pour obtenir un comportement moyen) ou d'une gestion *à la main* (le programmeur indique comment répartir les processus et les données sur les processeur et dans le temps). Elles sont aussi assez souvent mises en place au moment de l'exécution où un gestionnaire de tâches essaye de maintenir la machine dans un état proche d'un état idéal. Sur les machines à grain fin de parallélisme, vectoriser consiste à appliquer simultanément la même procédure de calcul à tous les éléments d'un vecteur. Un processeur est alloué par élément de vecteur. Cette procédure doit donc avoir un comportement temporel à peu près constant afin d'éviter que la majorité des processeurs n'attendent pour un petit nombre n'ayant pas terminé.

En fait, les informations nécessaires pour éliminer le parallélisme inutile sont principalement liées aux temps d'exécution des processus (complexité en temps) et aux tailles des données manipulées par ces processus (complexité en espace). Bien entendu, le calcul de ces informations doit être automatique, général et être, si possible, effectué à la compilation. Dans cette thèse, Nous présentons des méthodes d'analyse des programmes n'ayant trait qu'a la complexité en temps. Cependant quelques idées ou principes peuvent être adaptées à l'analyse de complexité en espace. Ces méthodes sont bien sûr totalement automatiques car destinées à être intégrées dans un paralléliseur, ou plus généralement dans une compilateur.

Découlant d'un infortuné résultat théorique [T36], l'analyse automatique de complexité est théoriquement limitée. En effet, l'arrêt d'un programme n'étant pas prédictible en général, le temps exact nécessaire au calcul l'est encore moins. Tout système automatique d'analyse de complexité est donc par définition incomplet; il ne répondra que partiellement ou n'analysera qu'une classe réduite de programmes. Les systèmes d'analyse automatique sont généralement constitués de deux phases. La phase *dynamique*[1] construit un jeu d'équations mutuellement récursives exprimant les dépendances entre les coûts des différentes procédures du programme. La phase *statique* tente de résoudre le jeu d'équations obtenu par diverses méthodes allant de la différentiation de fonctions génératrices à l'ap-

[1] Phase dynamique et phase statique constituent des terminologies standard en analyse automatique de complexité des programmes.

pariement de formes avec une base de motifs. Les coûts sont exprimés par rapport à une caractéristique significative des paramètres du programme, la *mesure*. Il peut s'agir de la taille (nombre d'éléments d'une liste) ou d'une valeur (variable borne de boucle). Une fois la mesure fixée, les coûts peuvent être fournis sous différentes formes (polynômes, fonctions, ...), avec une précision variable et correspondre à différents cas. Les coûts de pire (de meilleur) cas s'obtiennent aisément en maximisant (minimisant) le chemin parcouru par l'algorithme. Par contre, le coût moyen requiert une expression symbolique sur la distribution des objets paramètres. Même si cette distribution est connue au départ, l'exécution d'une procédure quelconque peut la modifier de façon non prévisible. L'analyse automatique de coût moyen est donc réservée à une classe de programmes sans effets de bords et avec peu de composition de fonctions.

Tous les systèmes proposés jusqu'a présent aspirent à fournir des résultats précis en limitant le pouvoir d'expression du langage analysé. Même si l'intérêt théorique est indiscutable, ce type d'approche est peu approprié à la gestion du parallélisme, car inadapté à l'analyse d'un langage réaliste (offrant la programmation par effets de bord). De plus, les résultats précis sont parfois difficiles à interpréter pour un module d'équilibrage de charge. Nos méthodes d'analyse ont été conçues en partant de ces constatations. Nous voulions que le langage support de l'analyse soit *réaliste*, c.à.d. qu'il soit assez puissant pour pouvoir être pris comme noyau de langage de programmation. Nous voulions aussi que les informations de complexité soient obtenues de façon totalement automatique et soient assez simple pour pouvoir être utilisées par un gestionnaire automatique de charge.

Nous utilisons la technique du typage non standard qui permet d'associer aux expressions du langage des propriétés autres que les classiques types. Plus précisément, nous reprenons le cadre formel du système d'effets conçu pour le langage *FX*. Ce dernier permet au programmeur d'exprimer un certain degré de parallélisme sans ce soucier de considérations pragmatiques comme l'architecture de la machine. Le système d'effet inclut, en plus des types, des informations concernant les régions mémoire dans lesquelles sont alloués les objets et décrivant les effets de bord éventuels lors de l'exécution des programmes. Le programmeur devra donc placer ses structures de données dans des régions mémoires abstraites et dire quel sont les effets des procédures sur ces régions. Le langage *FX* est explicitement typé et ne dispose pas de constructions parallèles explicites (du moins dans la version *FX87*). Les informations fournies par le programmeur lui permettent donc d'influencer indirectement le processus de parallélisation en interdisant l'exécution parallèle de procédures partageant des objets en écriture. Malgré son côté fortement typé, *FX* est un langage flexible car fusionnant les avantages des mondes applicatif et impératif. De plus, le polymorphisme générique présent permet aux programmes de s'abstraire sur les types, les régions, et les effets non significatifs des programmes.

Dans notre approche, les informations collectées par le système d'effet concernent bien sûr les temps d'exécution des procédures des programmes. A ce titre, il pourrait s'appeler système de temps mais les principes formels du système d'effets ne sont pas restreint à l'analyse des effets de bord. Nous présentons deux systèmes. Le premier reconstruit les informations de type et de temps sur les programmes. le langage est donc implicitement typé et libère le programmeur de la tâche de les fournir. Le second système est un vérificateur pour un langage explicite. Le langage est plus général à la fois pour le polymorphisme et pour la récursion. Nous présentons aussi une extension ou les types simples sont facultatifs. Il faut enfin noter que l'utilisation du cadre du système d'effet permet de prouver la consistance des temps avec un modèle d'évaluation et de prouver la correction des algorithmes.

Structure de la thèse

- Chapitre 1, nous présentons les concepts théoriques liés à l'analyse de complexité des algorithmes. Ensuite nous passons en revue les différents analyseurs automatiques existants.

- Le chapitre 2 est dédié au typage des langages applicatifs. Nous commençons par présenter les modèles théoriques de λ-calcul avec leurs propriétés de normalisation. Ensuite, nous présentons les techniques de preuve employées pour les système de typage avant de terminer sur la présentation des langages *FX* et Standard ML.

- Le chapitre 3 présente un système de reconstruction des types et des temps pour un langage implicitement typé. Le système est prouvé cohérent avec un schéma d'évaluation standard avec appel par valeur. L'algorithme le mettant en œuvre est prouvé correct par rapport à la spécification du système. En plus du type et du temps, l'algorithme fournit un jeu de contraintes constitué d'inégalités sur les temps. Un algorithme de résolution est proposé et sa terminaison est prouvée.

- Le chapitre 4 présente un système de vérification pour un langage explicitement typé. Le système est prouvé cohérent avec un schéma d'évaluation par appel par valeur. L'algorithme le mettant en œuvre est prouvé correct par rapport à la spécification du système. La puissance d'expression du langage est montrée à travers la présentation de l'opérateur de point fixe non codable avec le langage précédent. Enfin, nous présentons une extension où l'obligation d'écriture des types simple est relâchée.

- Enfin, nous concluons.

$$--\!-\!\diamondsuit\diamondsuit\!-\!--$$

Chapitre 1

Etudes de complexité

Bien qu'utile pour les systèmes de répartition de charge, l'analyse automatique et statique de complexité est un domaine de recherche peu développé. Ceci est dû aux limitations théoriques et à la diversité des méthodes combinatoires nécessaires. Quelques systèmes partiels ont néanmoins été proposés. Nous présentons quelques-uns de ceux-ci après avoir discuté des problèmes théoriques.

1 Terminaison et complexité

Cette section est dédiée à la présentation des problèmes et limitations théoriques de l'analyse automatique de complexité et à la définition de quelques concepts de base.

1.1 Incomplétude de l'analyse de complexité

Le concept de calcul est formalisé théoriquement par le modèle de la machine de Turing (TM). Le pouvoir d'expression de la TM est censé englober toutes les fonctions récursivement énumérables. C'est l'*hypothèse de Church* qui n'est pas prouvable formellement [HU79]. Elle ne pourrait être infirmée que si l'on trouvait un modèle théorique de calcul ayant un pouvoir d'expression supérieur à celui de la TM. Par contre, il est assez aisé de montrer (par simulations croisées) l'équivalence (en pouvoir d'expression) de la TM avec le λ-calcul et le modèle de la RAM (Random Acces Machine). Le λ-calcul est admis comme étant le modèle formel des langages applicatifs, tandis que la RAM l'est pour les ordinateurs séquentiels.

L'ensemble des algorithmes pouvant s'exécuter sur un ordinateur correspond donc aux fonctions récursives. Il inclut donc le problème connu sous le nom de *problème de l'arrêt de la machine de Turing* [Dew89,B80]. Celui-ci stipule qu'il ne peut exister de TM capable de déterminer si une TM quelconque s'arrêtera un jour après avoir imprimé un résultat ou bouclera infiniment. Ce théorème se démontre en construisant une TM inconsistante (qui s'arrête quand elle ne s'arrête pas et réciproquement). En d'autres termes, il est impossible de concevoir un algorithme décidant de l'arrêt de toutes les exécutions possibles de tous les algorithmes. Il est donc impossible de construire un système prenant en entrée un algorithme quelconque et calculant son temps d'exécution. L'analyse automatique de complexité d'algorithme est donc incomplète.

Il est à noter cependant que, s'il n'existe pas de méthode générale, il existe une méthode de calcul de complexité pour chaque algorithme. Les systèmes automatiques d'analyse de complexité ne pouvant utiliser une méthode générale, ceux-ci sont conçus en fusionnant des

méthodes diverses et variées afin de prendre en compte un ensemble le plus grand possible d'algorithmes.

1.2 Concepts

Lors de la conception d'un système automatique d'analyse, deux choix sont à effectuer. Ils concernent le choix du type de coût recherché et le moyen de l'exprimer.

Choix des coûts

Trois types de coût (en temps) des algorithmes sont utilisable. Le *coût de pire cas* correspond au maximum de temps nécessaire à l'algorithme pour s'exécuter. Le *coût de meilleur cas* correspond au temps minimum. Le *coût moyen*, correspondant à la moyenne des temps pris lors de toutes les exécutions possibles, est beaucoup plus difficile à obtenir. Pour les deux premiers, une analyse du programme maximisant (ou minimisant) le nombre d'étapes de calcul nécessaires à l'exécution suffit. L'analyse de coût en moyenne nécessite un comptage symbolique du nombre d'étapes lors de l'exécution de l'algorithme suivi d'une pondération liée à un critère de la donnée d'entrée. Pour cela, on calcule une expression symbolique de coût, fonction d'un paramètre caractéristique des valeurs des données d'entrée (nommé *mesure*). Ensuite, la moyenne pondérée est calculée à l'aide de la distribution des données sur cette *mesure*.

Expressions des résultats

La finesse avec laquelle sont exprimés les résultats est très variable. On peut se contenter d'une échelle proche de la notation de Landau ou désirer une expression de coût précise (avec les valeurs des constantes). La plupart du temps, même dans les systèmes à vocation *scientifique* (voir section 2.1), on se satisfait d'un taux de croissance asymptotique. Pour les systèmes d'équilibrage de charge, des informations plus grossières et très simples seraient déjà les bienvenues [G83].

2 Systèmes automatiques d'analyse de complexité

Nous présentons ici un aperçu des divers systèmes d'analyse automatique. Celui-ci ne se veut pas exhaustif. Notre but est surtout de mettre en lumière les différents points de vue pris par les chercheurs. Ceux-ci se caractérisent par le type de résultat recherché (coût moyen, de meilleur cas, de pire cas), la méthode utilisée pour l'obtenir et le type de langages pris en compte.

2.1 Préliminaires

Un analyseur automatique est le plus souvent composé de deux phases distinctes. La première est appelée *phase dynamique* car elle extrait du programme des informations concernant l'exécution. Elle construit un ensemble d'équations structurelles vérifiées par les coûts. Ces derniers sont exprimés en fonction des primitives et des procédures définies par l'utilisateur. Les procédures récursives mènent à des équations récursives. La deuxième phase est appelée *phase statique* car elle tente de résoudre les équations structurelles fournies par la première phase. Cette partie est la plus complexe. Elle fait appel à un ensemble de techniques mathématique diverses (liées généralement aux fonctions génératrices [Kn73]). Mais comme cette phase ne peut être complète, beaucoup ont adopté un module

d'appariement de formes. Cette méthode consiste à faire correspondre l'équation structurelle à résoudre avec un schéma contenu dans une base de connaissances et dont la fonction de coût est connue.

2.2 L'analyseur *Metric*

C'est le plus ancien des systèmes d'analyse automatique [W75]. Il effectue une micro-analyse de programmes *LISP* et fournit des informations complètes en temps et taille (meilleur cas, pire cas, coût moyen et variance). Il est constitué de trois phases : assignation des coûts, analyse de récursion et résolution des équations.

Assignation des coûts

METRIC sépare les fonctions en trois classes. Les primitives et les constantes ont des coûts fixes correspondant à la modélisation désirée du système. Viennent ensuite les *procédures fixes*; ce sont les fonctions ne contenant pas d'appel récursif et dont le coût est calculé par induction structurelle. Voici l'exemple de l'article [W75].

```
eq3(X,Y,Z) = if NOT(X=Y) then FALSE
             else if NOT(X=Z) then FALSE
                  else TRUE
```

On considère la fonction `eq3` comme ayant trois branches de probabilités respectives $(1-a)$, $(1-a)a$ et a^2 où a est la probabilité que `=` retourne `TRUE`. Le temps moyen d'exécution est $c_1 + a(c_2 - c_1)$. Les constantes c_1 et c_2 proviennent des temps des primitives :

$$\begin{aligned} c_1 &= 2.vref + eq + fncall + not + cref \\ c_2 &= 4.vref + 2.eq + 2.fncall + 2.not + cref \end{aligned}$$

où $vref$ est le temps d'accès à une variable, eq le temps de tester l'égalité, $fncall$ le temps d'appliquer une fonction, not le temps de la fonction de négation et $cref$ le temps d'accès à une constante.

Restent les fonctions récursives ou *procédures closes*. L'analyse d'une fonction récursive `f` est faite par induction structurelle sur la définition de `f` mais s'arrête sur un appel récursif à `f`. Prenons un nouvel exemple. Quelle est la taille du résultat de la fonction `append`?

```
append(X,Y) = if NULL(X) then Y
              else CONS(CAR(X),append(CDR(X),Y))
```

Construisons l'équation structurelle vérifiée par la taille du résultat de la fonction `append`. Pour cela, on utilise la fonction *length*. Il faut voir celle-ci comme marquant les appels à `append`.

```
length(append(X,Y)) = if NULL(X) then length(Y)
                      else 1+length(append(CDR(X),Y))
```

Analyse de récursion

Elle commence par une normalisation. Les équations structurelles précitées sont mises sous la forme d'une disjonction de cas (en remontant les tests). Une partie de ces derniers font une référence récursive. Ceux de l'autre partie n'en font pas. Ensuite, les équations structurelles sont transformées en une forme plus déclarative. Avec l'exemple de `append`, on obtient :

$$\begin{aligned} A(X,Y) &= length(\texttt{append}(X,Y)) \\ A((),Y) &= length(Y) \\ A((a.s_1),s_2) &= 1 + A(s_1,s_2) \end{aligned}$$

La liste vide est représentée par () et $(h.\ t)$ symbolise une liste dont h est le premier élément et dont t est le reste de la liste. Pour finir, les équations sont *projetées* sur le domaine des entiers de la façon suivante. Toute constante est substituée par sa valeur. Dans notre exemple, () est substitué par 0 tandis qu'un atome l'est par 1. Aux variables sont substituées des variables mathématiques formelles. Voici le résultat :

$$\begin{aligned} A(0,n_2) &= n_2 \\ A(1+n_1,n_2) &= 1 + A(n_1,n_2) \end{aligned}$$

Résolution

La résolution de systèmes comme le précédant ne présente pas de difficultés ($A(n_1,n_2) = n_1 + n_2$). Wegbreit nomme ces systèmes *Equations de différences non qualifiées*. Il existe un deuxième type de système : *Equations de différences qualifiées*. Prenons, par exemple, la fonction `member` suivante.

```
member(Z,L) = if NULL(L) then FALSE
              else if Z=CAR(L) then TRUE
                   else member(Z,CDR(L))
```

Le système d'équation associé, ci-dessous, s'interprète comme suit. Le coût de `member` sur une liste vide est constant. Le coût de `member` sur une liste dont le premier élément est celui recherché est constant. Le coût de `member` sur une liste dont le premier élément n'est pas celui recherché est l'addition d'une constante et du coût de la recherche dans la queue de la liste.

$$\begin{aligned} F(0) &= c_0 \\ F(n+1) &= c_1 \qquad \text{si } \texttt{X} = \texttt{CAR(Y)} \\ &= c_2 + F(n) \quad \text{si } \texttt{X} \neq \texttt{CAR(Y)} \end{aligned}$$

Les minima et maxima sont obtenus directement :

$$\begin{aligned} Min(F(n)) &= min(c_1, c_0 + n \times c_2) \\ Max(F(n)) &= c_2 \times (n-1) + max(c_1, c_0 + c_2) \end{aligned}$$

Par contre, l'obtention du coût moyen et de la variance est plus difficile. *METRIC* utilise, pour ce faire, la différentiation de fonctions génératrices [PB85] (méthode présentée en section 2.6) ou un module d'appariement de formes.

L'analyseur *METRIC* tente de fournir des résultats complets. Par contre, il ne traite pas les fonctions non mutuellement récursives ou contenant des effets de bord. La phase de résolution, qui ne peut être complète, est la plus complexe de l'analyseur. Wegbreit pensait qu'il pourrait, à long terme, y intégrer diverses techniques mathématiques, ceci au sein d'un système expert.

2.3 L'analyseur *ACE*

L'analyseur *ACE* [L88,L87], pour Automatic Complexity Evaluator, effectue une macro-analyse de programmes *FP* [B78]. Les deux phases de l'analyse, statique et dynamique, sont constituées respectivement d'un système de réécriture et d'un module d'appariement de formes. Le résultat de pire cas est exprimé dans le langage source.

Le langage *FP* est un langage applicatif. L'unique structure de données est la liste à laquelle on applique successivement des fonctions pures. Le langage *FP* est donc dépourvu d'effet de bord. On définit, par exemple, la fonction calculant la factorielle par la définition suivante (reprise de l'article [L88]) :

$$\texttt{fact} \;=\; \texttt{eq0} \rightarrow \texttt{1} \mid * \circ [\texttt{id}, \texttt{fact} \circ \texttt{sub1}]$$

Les fonctions `eq0`, `1`, `id` et `sub1` sont les fonctions unaires qui calculent respectivement le test d'égalité à `0`, la constance à `1`, l'identité et la décrémentation de `1`. L'expression $E_1 \rightarrow E_2 \mid E_3$ spécifie un test sur l'expression E_1 avec branchement sur E_2 si vrai ou sur E_3 si faux. L'expression $[E_1, ..., E_n]$ est le constructeur de listes. Enfin, les opérateurs $\circ$ et $*$ sont, respectivement, la composition des fonctions et la multiplication.

La phase dynamique est un système de transformation de programme dirigé par la syntaxe. Il fournit une équation structurelle vérifiée par la fonction Cf (fonction de coût de f). Suivent les règles nécessaires à la dérivation de $fact$ en $Cfact$. *prim* est une méta-variable symbolisant les fonction primitive de *FP* et f est la fonction en cours de dérivation. La réduction d'une liste $[E_1, ..., E_n]$ par une fonction g est notée $(/g)[E_1, ..., E_n]$.

$$\begin{aligned}
T(\texttt{E}_1 \rightarrow \texttt{E}_2 \mid \texttt{E}_3) &= \texttt{E}_1 \rightarrow \texttt{+} \circ [T(\texttt{E}_1), T(\texttt{E}_2)] \mid \texttt{+} \circ [T(\texttt{E}_1), T(\texttt{E}_3)] \\
T(\texttt{E}_1 \circ \texttt{E}_2) &= \texttt{+} \circ [T(\texttt{E}_1) \circ \texttt{E}_2, T(\texttt{E}_2)] \\
T([\texttt{E}_1, ..., \texttt{E}_n]) &= (/\texttt{+})[T(\texttt{E}_1), ..., T(\texttt{E}_n)] \\
T(prim) &= 0 \\
T(\texttt{f}) &= \texttt{plus1} \circ \texttt{Cf}
\end{aligned}$$

L'équation structurelle dérivée de la fonction $fact$ est la suivante :

$$\begin{aligned}
\texttt{Cfact} \;=\; \texttt{eq0} \;\rightarrow\; & \texttt{+} \circ [0,0] \mid \\
& \texttt{+} \circ [0, \texttt{+} \circ [0, \texttt{+} \circ [0, \texttt{+} \circ [\texttt{plus1} \circ \texttt{Cfact} \circ \texttt{sub1}, 0]]]]
\end{aligned}$$

La phase statique tente de résoudre l'équation obtenue. La suppression de la récursion est le problème principal. Le système commence par simplifier l'équation structurelle obtenue par une méthode analogue à celle de la phase dynamique mais avec un jeu de règles différent. On obtient, ainsi, une forme normale de l'équation.

$$Cfact \;=\; eq0 \rightarrow 0 \mid plus1 \circ Cfact \circ sub1$$

La dernière étape consiste à appliquer le principe d'induction récursive [M63]. Celui-ci stipule que deux fonctions vérifiant la même équation récursive sont équivalentes sur le domaine de la fonction définie par cette équation. Pour cette opération, on effectue un appariement de formes avec une base de connaissances contenant des équations standards dont le coût est connu. Dans notre exemple, notre équation s'apparie avec celle vérifiée par l'identité. On en déduit que la complexité de la fonction factorielle est linéaire par rapport à la valeur de son argument.

Le système *ACE* présente trois avantages majeurs.

- Sa base de connaissances est extensible et peut ainsi grossir de façon à prendre en compte de plus en plus de programmes.
- Toute l'analyse a lieu dans le cadre formel du langage *FP*. Ce dernier offre une syntaxe concise et une sémantique claire favorisant la validation des transformations.
- Enfin, il donne les résultats en fonction de la mesure relevant de l'argument.

Par contre, il souffre de plusieurs inconvénients.

- Les effets de bord et les structures de données complexes (graphe avec cycles, ...) ne sont pas pris en compte.
- Il n'assure pas la consistance des règles de la base.
- Il fournit un résultat parfois difficile à interpréter car exprimé en *FP*.

2.4 L'analyseur dynamique de Sands

Le dialecte du langage *FP*, analysé par le système *ACE*, n'est pas un langage applicatif d'ordre supérieur. En effet, il n'autorise pas de passer les fonctions en paramètre d'appel, de les rendre en résultat ni de les mémoriser dans les structures de données. Sands [S88] propose une extension de la phase dynamique pour prendre en compte de tels dialectes. L'intérêt de sa démarche est qu'il prouve la consistance des transformations présentées par rapport à une sémantique dynamique classique. Par contre, il n'offre pas de solution pour exploiter les équations structurelles et récursives que son système construit.

2.5 L'analyseur automatique de Rosendahl

Rosendahl, dans [R86,R89], propose un système d'analyse automatique de complexité pour les langages applicatifs sans fonction de première classe. Ce système est constitué d'une phase de réécriture de programme et d'une phase d'interprétation abstraite [CC77,AH87]. Le résultat est une fonction majorant le temps d'exécution du programme (TBF pour Time Bound Function). Une TBF exprime la complexité du programme original en codant un polynôme en la mesure (généralement, la taille) des données d'entrée.

La phase dynamique d'analyse, constituée d'un système de réécriture de programmes, construit une version du programme source décomptant les étapes de calcul (SCV pour Step Counting Version). Cette fonction (résultat intermédiaire) possède le même domaine argument que le programme original. Par contre, elle rend en résultat le nombre d'étapes de calcul du programme source. Les étapes de calcul considérées sont les applications de fonctions à leurs arguments, les tests et les accès aux variables. Il est à noter que si le programme original ne termine pas, la SVC réécrite ne termine pas non plus.

La phase statique d'analyse construit la TBF. Cette dernière doit respecter le critère conservatif suivant :

$$tb(n_1, ..., n_p) \geq Max\{t(x_1, ..., x_p) | l(x_i) = n_i\}$$

où tb est la TBF du programme, t est la fonction intermédiaire de décompte des étapes et l est une fonction associant une mesure aux structures de données. Une approximation est construite par interprétation abstraite sur l'ensemble des structures de données partiellement connues.

Pour une présentation plus détaillée, nous reprenons l'exemple présenté par Rosendahl dans son article. Il s'agit de la fonction calculant l'union de deux ensembles codés par des listes. Elle utilise, comme test de fin de récursion, la fonction auxiliaire définie par le programmeur, `member` (qui teste l'appartenance d'un atome à une liste). Pour cette raison, l'analyseur *Metric* n'était pas capable d'analyser ce type de programmes. En effet, la récursion de la fonction `union` n'est pas basée sur un aspect structurel (comme cela aurait été le cas par exemple avec les classiques fonctions de Lisp `map` et `reduce`).

```
union(x,y) = if null(x) then y
             else if member(car(x),y) then union(cdr(x),y)
                  else cons(car(x),union(cdr(x),y))

member(x,s) = if null(s) then false
              else if eq(x,car(s)) then true
                   else member(x,cdr(s))
```

Le système de réécriture, noté $\mathcal{T}$ est simple. L'accès aux variables et aux constantes prend un *top* d'horloge. Le temps pris par l'appel à une fonction est la somme des temps pris pour évaluer les arguments, du temps pris pour appliquer la fonction aux arguments et d'un *top*. Le temps total d'une expression de test est la somme du temps associé à la condition et du temps de la branche *vrai* si le test est vérifié ou du temps de la branche *fausse* sinon.

Après réécriture, on obtient les SCV `tunion`$=\mathcal{T}[\![$`union`$]\!]$ et `tmember`$=\mathcal{T}[\![$`member`$]\!]$. Il faut noter que la fonction `member` est encore utilisée par `union` (et doit donc être conservée) alors que la fonction `union` n'est plus nécessaire.

```
tunion(x,y) = if null(x) then 4
              else add(tmember(car(x),y),
                       if member(car(x),y)
                          then add(11,tunion(cdr(x),y))
                       else add(14,tunion(cdr(x),y)))

tmember(x,s) = if null(s) then 4
               else if eq(x,car(s)) then 9
                    else add(12,tmember(x,cdr(s)))
```

L'interprétation abstraite est construite avec la SCV du programme sur l'ensemble des structures partiellement connues. Le domaine standard, D, des structures de données est constitué des arbres à la Lisp. Pour obtenir le domaine des structures partielles, $\widetilde{\mathrm{D}}$, on ajoute l'atome spécial `all`. Celui-ci représente n'importe quel sous-arbre possible (l'ensemble D tout entier). On définit la paire de fonctions adjointes (abstraction et concrétisation) entre le domaine $\widetilde{\mathrm{D}}$ et la partition de l'ensemble des structures de données, $\mathcal{P}(\mathrm{D})$.

$$\gamma(x) = \begin{array}{ll} \emptyset & \text{si } x = \bot \\ \mathrm{D} & \text{si } x = \texttt{all} \\ \{x\} & \text{si } x \in \mathrm{Atom} \\ \{\langle a,b\rangle | a \in \gamma(y) \text{ et } b \in \gamma(z)\} & \\ & \text{si } x = \langle y,z\rangle \end{array}$$

$$\alpha(x) = \begin{array}{ll} \bot & \text{si } x = \emptyset \\ e & \text{si } x = \{e\} \\ \texttt{all} & \text{si } x \cap \mathrm{Atom} \neq \emptyset \\ \langle \alpha(\{a|\langle a,b\rangle \in x\}), \alpha(\{b|\langle a,b\rangle \in x\})\rangle & \end{array}$$

Les fonctions de base sur D sont étendues au domaine $\widetilde{\mathrm{D}}$ de façon naturelle. Par exemple, la fonction `car` devient :

```
cãr(x) = all  si  x = all
         a    si  x = ⟨a,b⟩
```

L'interprétation abstraite, de fonctionnelle $\tilde{\mathcal{U}}$, ressemble à l'interprétation standard. Les deux différences sensibles sont liées aux opérateurs de base et au test. Dans les cas des opérateurs, on utilise l'opérateur étendu associé (pour `car`, `cãr`). Pour une expression de test, une alternative supplémentaire est introduite pour le cas où le résultat du test est `all`. Dans cette éventualité, le maximum des deux branches est pris comme résultat. Grâce aux propriétés d'inclusion de l'interprétation abstraite, on peut montrer que :

$$\tilde{\mathcal{U}}[\![\mathcal{T}[\![P]\!]]\!]\langle\alpha(\{x_1|l(x_1)=n_1\}),...\rangle \geq Max\{t(x_1,...,x_p)|l(x_i)=n_i\}$$

Il ne reste plus qu'à exprimer, à travers la fonction l, la mesure que l'on désire prendre en compte dans l'analyse. Comme l'on cherche à connaître le temps d'exécution pour toutes les structures de taille n donnée, on a besoin de la fonction inverse l^{-1}. La fonction de mesure doit donc être inversible. En fait, l'abstraction est intégrée dans l'inverse et l^{-1} a pour domaine $I\!N \rightarrow \widetilde{\mathrm{D}}$.

Revenons à notre exemple. La fonction de mesure choisie est la fonction `length`. L'inverse sur $\widetilde{\mathrm{D}}$, appelée `length1`, est donc :

```
length1(n) = if eq(n,0) then nil
                 else cons(all,length1(sub(n,1)))
```

Après simplification, la TBF de la fonction `union` s'écrit :

```
tbunion(n,m) = add(4,add(mul(19,n),mul(12,mul(n,m))))
```

Les principaux avantages de ce système résident dans la qualité des résultats rendus (meilleurs que ceux de [W75] et équivalents à ceux de [L87,L88]) et dans le fait que le système de réécriture ainsi que l'évaluateur abstrait sont prouvés corrects. La qualité des résultats peut, de plus, être améliorée en choisissant un domaine abstrait plus riche. Par contre, il souffre de plusieurs défauts. Les résultats sont fournis sous formes de fonctions et, donc, sous une forme peu lisible. Ensuite, la TBF rendue peut très bien ne pas terminer. Enfin, le langage analysé est sans effet de bord et sans fonction d'ordre supérieur.

2.6 L'analyseur $\Lambda\Upsilon\Omega$

L'analyseur automatique $\Lambda\Upsilon^{\Omega}$ [FSZ88] effectue une analyse de coût moyen sur les algorithmes opérant sur des structures de données décomposables. La phase dynamique est assurée par un analyseur algébrique (ALAS) compilant les spécifications d'algorithme en fonctions génératrices. La phase statique, l'analyseur analytique (ANANAS), extrait des informations asymptotiques sur les coefficients des séries génératrices, donc sur les taux de croissance des temps.

Fonctions génératrices

Les fonctions génératrices sont utilisées en combinatoire pour décompter les objets en fonction d'un critère donné (ex : nombre d'arbres de taille n, nombre d'arbres ayant p feuilles, ...). Nous allons décrire la méthode sur un exemple tiré de [Kn81]. Il s'agit de calculer x^a où l'exposant a est codé par une chaîne de bits. Voici la spécification de l'algorithme naïf en ADL (Algorithm Description Language [Z89]).

```
type chain = bit × chain  |  null ;
     bit = zero | one ;
     zero,one = atom ;

procedure expmod(c :  chain) ;
   case c of
        () => nil ;
        (zero,c1) => begin expmod(c1); squaring; end ;
        (one,c1) => begin expmod(c1); squaring; multiply; end ;
   esac

measure multiply :  1 ;
        squaring :  1 ;
        atom :  1 ;
```

Le premier bloc spécifie le type des données prises en paramètre du programme. Le deuxième bloc est la spécification proprement dite de l'algorithme. Le troisième bloc définit les coûts associés aux primitives. Attention, il s'agit d'une spécification et non pas d'un programme. En particulier, la quantité symbolisée par x n'apparaît pas car elle n'influe pas sur la complexité de l'algorithme.

Avant d'analyser l'algorithme, il faut décompter les chaînes de bits en fonction de leurs tailles. Pour cela, on construit l'équation structurelle vérifiée par la fonction génératrice des chaînes ($chain_z$) :

$$chain_z = 1 + z \times chain_z + z \times chain_z$$

La variable symbolique z sert à *marquer* les objets que l'on désire compter. L'équation s'interprète comme suit :

- Il existe une seule chaîne de taille nulle ($1 = z^0$).
- Il existe une chaîne commençant par l'atome `one` ($z^1 \times chain_z$).
- Il existe une chaîne commençant par l'atome `zero` ($z^1 \times chain_z$).

Après simplification, l'équation se réécrit sous la forme :

$$chain_z = \frac{1}{1-2z}$$

Maintenant, il ne reste plus qu'à extraire les coefficients de la fonction génératrice $chain_z$. Pour cela, on peut faire une expansion de Taylor en zéro. Jusqu'au cinquième terme, cela donne :

$$chain_z = 1 + 2z + 4z^2 + 8z^3 + 16z^4 + 32z^5 + O(z^6)$$

Mais, on cherche généralement à obtenir une expression symbolique permettant la réutilisation du résultat dans des calculs ultérieurs. Grâce au théorème de Darboux [FV87], on sait que le n-ième coefficient, $[z^n]chain_z$, de $chain_z$ est asymptotiquement égal à :

$$[z^n]chain_z = \frac{1 + O(1/n)}{2^{-n}}$$

On peut maintenant analyser l'algorithme. La méthode est identique. Elle consiste à marquer d'un z les primitives que l'on doit compter. L'équation vérifiée par τ_z (coût de `expmod`) suit :

$$\tau_z = (z\tau_z + zchain_z) \ + \ (z\tau_z + 2zchain_z)$$

La première expression correspond à la branche `zero`. Il s'y trouve un appel récursif ($z\tau_z$) et un appel à `squaring` ($zchain_z$). La deuxième expression correspond à la branche `one`. Il s'y trouve les deux mêmes appels auxquels s'ajoute un appel à `multiply`. Par le même théorème (Darboux), on calcule la solution :

$$[z^n]\tau Expmod_z = 2^{n-1}3n\,(1 + O(1/n))$$

Le coût moyen asymptotique de l'algorithme sur une donnée de taille n est calculé en divisant le deuxième résultat par le premier : $\frac{3}{2}\,n$.

L'analyseur automatique

La phase dynamique se décompose en deux parties. La première extrait de la spécification *ADL* les équations vérifiées par les fonctions génératrices. La deuxième est la résolution de ce système d'équations pour obtenir les fonctions génératrices. Cette deuxième partie est faite sous le logiciel de calcul formel *MAPLE* [CGGW85]. La phase statique est plus complexe et fait intervenir diverses techniques de résolution mathématique autres que celles présentée. Cette phase est aussi effectuée sous *MAPLE*.

L'analyseur $\Lambda\Upsilon^{\Omega}$ est volontairement restreint. Les données admises doivent être structurellement décomposables et construites par certains types de constructeurs. Par exemple, les structures de graphe cyclique sont interdites. Les algorithmes acceptés sont ceux de la classe des calculs d'induction sur les structures déjà citées. Le problème majeur de cette méthode est qu'elle n'autorise pas la composition de fonctions. En effet, les distributions des valeurs des arguments sont supposées uniformes. Pour pouvoir appliquer une fonction aux résultats d'une précédente fonction, il faudrait que la distribution des résultats soit uniforme (ce qui est, en général, faux).

2.7 Le système formel de Ramshaw

Le système formel de Ramshaw [R79] n'est pas, à proprement parler, un outil d'analyse de complexité. Il calcule les distributions des valeurs des variables en tous les points du programme. Il est basé sur une analyse de flot d'exécution et exprime les distributions à l'aide d'une sémantique opérationnelle. Le langage analysé est un langage impératif simple contenant des affectations et lectures de variables, des tests et des boucles (boucles `for`).

Pour bien saisir la démarche, examinons les détails d'un exemple simple de petit programme impératif sans boucle.

```
if K = 0 then K := K + 2 else K := K - 1 fi ;
```

On considère que la variable `K` est affectée en début de programme à deux valeurs, `0` et `1`, avec la même probabilité. L'*état de fréquences* du programme au début de l'exécution s'exprime ainsi :

$$[Fr(\texttt{K} = 0) = \frac{1}{2}] \wedge [Fr(\texttt{K} = 1) = \frac{1}{2}]$$

Lorsque l'on raisonne en terme de probabilité, la somme doit être égale à un. Dans la branche `then`, la variable `K` a la valeur 0 avec une probabilité de 1. De même, dans la branche `else`, `K` a la valeur 1 avec une probabilité de 1. Comment fait on en sortie du test pour recombiner les états de fréquences? En effet, on a perdu l'information que `K` se répartit entre les deux valeurs 0 et 1 avec la même probabilité. Pour cela, Ramshaw introduit la notion de *fréquences* dont la somme est inférieure ou égale à un. En sortie des branches du test, les états de fréquences sont les suivants :

$$\begin{aligned} \texttt{then} \quad &: \quad [Fr(\texttt{K} = 2) = \frac{1}{2}] \wedge [Fr(\texttt{K} \neq 2) = 0] \\ \texttt{else} \quad &: \quad [Fr(\texttt{K} = 0) = \frac{1}{2}] \wedge [Fr(\texttt{K} \neq 0) = 0] \end{aligned}$$

On peut alors les recombiner facilement. Le résultat est clairement consistant.

$$[Fr(\texttt{K} = 2) = \frac{1}{2}] \wedge [Fr(\texttt{K} = 0) = \frac{1}{2}] \wedge [Fr([\texttt{K} \neq 2] \wedge [\texttt{K} \neq 0]) = 0]$$

Le langage pris en compte est très simple. Il n'offre aucune possibilité de manipuler des structures de données complexes ni d'utiliser des objets applicatifs.

Les extensions de Hickey et Cohen

En s'aidant de la sémantique des programmes statistiques due à Kozen [Ko81], [HC88] ont étendu l'approche de Ramshaw afin de prendre en compte les structures de données complexes. Ils proposent aussi une extension aux langages applicatifs purs tels que le langage *FP*.

2.8 La rapidité de Gray

Le langage applicatif *Multilisp* est conçu pour la programmation des machines parallèles. Le parallélisme est contrôlé grâce à l'insertion de *futurs*[1] dans les programmes. Un futur permet au programmeur de préciser, à l'interpréteur ou au compilateur, l'exécution en parallèle d'une expression du langage. L'ajout à la main des futurs devient vite difficile avec l'accroissement de la taille des programmes. Gray [G83] a proposé une méthode automatique utilisant un critère empirique de complexité en temps : la rapidité[2].

L'insertion de *Futur*s

Un futur est une directive d'évaluation indiquant à l'évaluateur d'évaluer l'expression concernée en parallèle. Par exemple, l'ordre d'évaluation du programme :

`(f` e_1 `(future` e_2`))`

[1] *futures* en Anglais.
[2] Gray la nomme *Quickness*.

est classique dans le fait qu'on évalue e_1, puis `(future` e_2`)` et qu'on appelle `f` avec les valeurs résultats. L'originalité réside dans l'expression `(future` e_2`)` qui n'attend pas que l'évaluation de e_2 soit terminée pour retourner immédiatement une valeur spéciale marquant l'expression e_2. L'évaluation de l'appel à `f` commence donc en parallèle avec celle de e_2. Lorsque la fonction `f` tentera d'accéder au résultat de e_2, elle se bloquera en attente de la terminaison de l'évaluation de e_2.

Comme le montre Gray, l'insertion automatique naïve des futures dégrade généralement les performances de l'évaluateur. Un critère de choix des points d'insertion, basé sur le temps d'évaluation, est donc nécessaire.

Le critère de *Rapidité*

L'idée principale est de considérer qu'une expression est rapide quand elle s'évalue en moins de temps qu'il en est nécessaire pour évaluer son futur. Le temps d'évaluer un futur, variant selon les implémentations, ne peut pas servir de référence absolue. Gray propose donc un critère de rapidité plus abstrait. Seront considérées comme rapides les expressions constituées d'évaluations de constantes (entiers, booléens), d'accès à des variables, de combinaisons de primitives lisp ou d'appels à des fonctions simples. Naturellement exclues de cette définition, les fonctions récursives sont considérées comme lentes.

Bien qu'incluant quelques particularités liées à l'insertion des futurs, le critère de rapidité proposé ressemble à la notion de temps que nous utilisons par la suite dans nos systèmes de typage. Malheureusement, il n'est conçu que pour un sous-ensemble du langage du premier ordre et dépourvu d'effets de bord.

3 Conclusion

Malgré l'utilité pratique évidente des méthodes d'analyse automatique de complexité dans les systèmes de répartition de charge, peu d'entre eux ont atteint le stade de l'intégration réelle. On leur préfère souvent des méthodes simples comme pour l'insertion automatique des *futures* [G83] où une notion de rapidité (et de lenteur) empirique est utilisée. Pourtant la demande existe pour les machines parallèles avec la répartition des tâches sur les différents processeurs, pour les machines vectorielles avec la génération de code vectoriel sur des fonctions *courtes* ou encore pour les systèmes de calcul formel avec le choix de la méthode de calcul à appliquer (temps de calcul contre taille mémoire nécessaire). Les systèmes utilisant l'appariement de forme ou d'autre méthodes de résolution simple sont rapides et donc de bon candidats pour l'intégration. D'un autre côté les systèmes fournissant des résultats précis resteront d'un intérêt plus théorique car ils nécessitent l'aide d'un intervenant humain et de longs temps de calculs.

– – –◇◇◇– ––

Chapitre 2

Systèmes de typage

La nécessité d'introduire la notion de typage dans les langages de programmation s'est fait sentir très tôt, dès la création des premiers compilateurs. Connaître le type d'une variable (e.g. entier ou flottant) est requis pour savoir quel emplacement réserver en mémoire ou quelle instruction d'addition générer dans le code objet. Après les premières approches très pragmatiques (les langages *Fortran* et *Cobol*), les systèmes de typage sont devenus plus complets et plus cohérents (e.g. *Pascal* et *Algol*). Le typage fort, en effet, incite à une programmation plus structurée, documente les programmes, durcit le code et permet d'effectuer diverses optimisations à la compilation.

Au plan scientifique, le domaine du typage se décompose en trois branches. La première consiste en l'étude de diverses propriétés théoriques comme la normalisation [FLD83,BH90] ou les classes de complexité des programmes [GSS90]. La deuxième concerne les modèles théoriques des systèmes de type. Le polymorphisme n'étant pas modélisable par des ensembles [R84], les modèles proposés font, par exemple, appel aux idéaux [MPS84] ou à la notion de *closure*[1] [M79]. La troisième branche cherche à enrichir les informations contenues dans les types. Le but est alors d'augmenter la sûreté et la rapidité du code généré [T88,LW90]. Des informations non standard ont même été étudiées pour la prise en compte des effets de bord [Lu87,LG88,TJ91b], de l'analyse de stricticité [KM89], de l'analyse de temps[2] en évaluation partielle [CJ91] ou de l'analyse de complexité [DJG91].

Le but de ce chapitre est de présenter les principales classes de systèmes de typages présents dans les langages applicatifs (section 1). Ensuite (section 2), nous présentons les techniques de spécification et de preuve utilisées pour valider la correction et la consistance des systèmes de typage présentés dans cette thèse. Pour terminer (section 3), nous donnons un aperçu des langages fortement typés que sont *FX'87* et *Standard ML*.

1 Hiérarchie de typage

Avant de parler de langages applicatifs typés et de preuves, il est bon d'avoir présent à l'esprit les différents modèles de λ-calcul , ainsi que leurs principales propriétés théoriques. Ces modèles sont exposés dans un ordre croissant de raffinement des types employés. Nous énumérons, dans l'ordre, le λ-calcul non typé, le λ-calcul typé du premier ordre, le λ-calcul typé du second ordre et le λ-calcul typé d'ordre ω. Tous sont accompagnés de leurs propriétés de terminaison.

[1] Il ne s'agit pas la de la fermeture à la LisP. Une closure correspond à une fonction f d'un domaine D à valeur dans D telle que $fof = f$ et $Id \sqsubseteq f$.

[2] BTA pour "Binding Time Analysis" en Anglais.

λ-calcul non typé (λNT) :

Les expressions du λ-calcul obéissent à la grammaire suivante :

Définition 2.1 (λNT)

$$e \ ::= \ c \mid v \mid \lambda v.e \mid (e_1 \ e_2)$$

Une expression est, respectivement, une constante, une variable, une abstraction sur une variable (v) ou l'application d'une expression e_1 à une expression e_2. On dit que la variable v est *liée* dans l'expression $\lambda\mathtt{v.e}$ et *libre* sinon. L'évaluation des expressions est exprimée à travers deux règles de réécriture :

$$\begin{aligned} \lambda\mathtt{v.e} &\longrightarrow_\alpha \lambda\mathtt{w.e}[v\backslash w] \\ (\lambda\mathtt{v.e}_1 \ \mathtt{e}_2) &\longrightarrow_\beta e_1[v\backslash e_2] \end{aligned}$$

La première, l'*α-conversion*, permet de renommer la variable muette d'une abstraction. Une expression $\lambda\mathtt{v.e}$ se réécrit $\lambda\mathtt{w.e}[v\backslash w]$ où $e[v\backslash w]$ représente l'expression e dans laquelle on a substitué la variable v par w. La deuxième, la *β-contraction*, modélise une étape de calcul. L'application $(\lambda\mathtt{v.e}_1 \ \mathtt{e}_2)$ se réécrit $e_1[v\backslash e_2]$. Une expression que l'on ne peut plus β-contracter est nommée *forme normale*. L'exécution d'un programme correspond à la fermeture transitive de la règle de β-contraction et est appelée *β-réduction*. La forme normale obtenue est le résultat du calcul. Pour terminer, nous ne considérons dans cette thèse que le mode d'évaluation avec appel par valeur. Celui-ci correspond à une β-réduction *rightmost-innermost* : L'expression la plus à droite et la plus intérieure est évaluée en premier.

Définition 2.2 *Une expression e est* normalisable *si elle peut être réduite en une forme normale. L'expression e est* fortement normalisable *si toute β-réduction de e termine sur une forme normale.*

L'expression $(\lambda\mathtt{v.c} \ (\lambda\mathtt{v.(v \ v)} \ \lambda\mathtt{v.(v \ v)}))$ est normalisable mais non fortement normalisable. En effet, la forme normale ($\mathtt{c}$) s'obtient par β-contraction de l'application la plus externe. Par contre, une réduction infinie peut être obtenue en ne contractant que l'application interne de $\lambda\mathtt{v.(v \ v)}$ à elle-même.

Définition 2.3 (Normalisation) *Un λ-calcul possède la propriété de* normalisation *(NP) ssi toute λ-expression est normalisable.*

Définition 2.4 (Normalisation Forte) *Un λ-calcul possède la propriété de* normalisation forte *(SNP) ssi toute λ-expression est fortement normalisable.*

Il faut noter que SNP implique NP. Nous arrivons au principal théorème concernant la terminaison de λNT.

Théorème 2.5 λNT *ne possède pas la propriété de normalisation (ni, par conséquent, celle de normalisation forte).*

Preuve 2.5 Il n'est pas difficile d'exhiber une λ-expression ne possédant pas de forme normale (ex : l'expression argument de l'exemple précédent). □

λ-calcul typé du premier ordre (λT1) :

Le λ-calcul typé du premier ordre est obtenu en imposant un type à chaque variable. Une application est correcte (et β-contractible) si le type de la variable de l'abstraction est identique à celui de l'expression argument.

Définition 2.6 (λT1)

$$\begin{array}{lcl} e & ::= & c \mid v \mid \lambda v{:}t.e \mid (e_1\ \ e_2) \\ t & ::= & s \mid t_1 \rightarrow t_2 \end{array}$$

La méta-variable de type, s, décrit un ensemble des constantes de type comme `int` ou `bool`. Le type $\mathtt{t}_1 \rightarrow \mathtt{t}_2$ est le type d'une abstraction de domaine t_1 et de codomaine t_2.

Théorème 2.7 *Le λ-calcul typé du premier ordre est fortement normalisable.*

Preuve 2.7 Prouvé pour la première fois par Turing (voir [FLD83] pour une preuve simple). Informellement, on définit aisément une mesure sur les expressions qui, grâce à la restriction imposée aux applications par le typage, décroît à chaque β-contraction. □

L'opérateur de point fixe n'est pas typable dans ce schéma. *Pascal* privé de la récursion correspond à λT1. Les limitations de *Pascal* , en ce qui concerne la définition de modules, sont bien connues. *Pascal* ne permet pas, par exemple, l'écriture d'un programme de tri abstrait sur le type des données à trier et sur le prédicat d'ordonnancement. Cet obstacle à la programmation modulaire disparaît avec les langages typés d'ordre supérieur.

λ-calcul typé du deuxième ordre[3] (λT2) :

On construit λT2, à partir de λT1 en y ajoutant un mécanisme permettant d'abstraire les expressions sur les types.

Définition 2.8 (λT2)

$$\begin{array}{lcl} e & ::= & c \mid v \mid \lambda v{:}t.e \mid (e_1\ \ e_2) \mid \Lambda w{::}k.e \\ t & ::= & s \mid w \mid t_1 \rightarrow \mathtt{T}_2 \mid \Delta w{::}k.t \\ k & ::= & \mathtt{type} \mid \mathtt{type} \rightarrow k \end{array}$$

Une expression est, respectivement, une constante, une variable, une abstraction, une application ou une abstraction sur un type. On appelle *polymorphisme* (*polymorphisme générique* ou encore *polymorphisme uniforme paramétrique*) cette capacité des expressions de s'abstraire sur un type. Un type est, respectivement, une constante de type, une variable de type, le type d'une abstraction ou le type d'une abstraction polymorphe (type polymorphe). Les types sont eux-mêmes typés par des sortes (k pour *kind* en anglais). Une *sorte*, ou *méta-type*, est, respectivement, une constante ou la sorte d'un type polymorphe. Par exemple, la fonction d'identité (polymorphe) s'écrit $\Lambda\mathtt{t}{::}\mathtt{type}.\lambda\mathtt{v}{:}\mathtt{t}.\mathtt{v}$ et a pour type $\Delta\mathtt{t}{::}\mathtt{type}.\mathtt{t} \rightarrow \mathtt{t}$. Plusieurs contraintes doivent être respectées lors de la formation des expressions. La principale est que l'on ne peut s'abstraire sur un type que s'il ne fait pas partie des variables de type libres des variables libres de l'expression. On interdit, de la sorte, les expressions incorrectes comme $\lambda\mathtt{x}{:}\mathtt{t}.\Lambda\mathtt{t}{::}\mathtt{type}.\mathtt{x}$.

[3] L'ordre de typage est la plus petite solution d'une équation définie sur les sortes (type de type, non terminal k). Dans le cas présent, les sortes sont définies par une équation de la forme $k = l \mid k_1 \rightarrow k_2$. L'équation à l'ordre s'écrit $Ordre(k) = max(Ordre(l), Ordre(k_1) + 1, Ordre(k_2))$ où $Ordre(type) = 1$.

Théorème 2.9 λT2 *est fortement normalisable.*

Preuve 2.9 La preuve est basée sur le même principe que pour λT1, mais avec une mesure plus fine. En effet, la précédente ne tient pas compte de certaines circularités apparues lors du passage au deuxième ordre. Voir [FLD83]. □

λ-calcul typé d'ordre ω (λTω) :

On construit λTω, à partir de λT2, en autorisant l'abstraction des expressions, non plus uniquement sur des types, mais aussi sur des types polymorphes. La syntaxe est donc pratiquement identique. La sorte *type* est remplacée par k dans le deuxième cas de la définition des sortes.

Définition 2.10 (λTω)

$$\begin{array}{rcl} e & ::= & c \mid v \mid \lambda v{:}t.e \mid (e_1\ e_2) \mid \Lambda w{::}k.e \\ t & ::= & s \mid w \mid t_1 \rightarrow t_2 \mid \Delta w{::}k.t \\ k & ::= & \texttt{type} \mid k \rightarrow k \end{array}$$

Les propriétés de terminaison son inchangées.

Théorème 2.11 *Le λ-calcul typé d'ordre ω est fortement normalisable.*

Preuve 2.11 Voir [G72], le problème étant toujours de définir une mesure décroissante malgré les circularités. □

Il est bon de remarquer que, dans les trois précédents λ-calculs typés, l'analyse de complexité est complète. Ces λ-calculs ayant la propriété de normalisation forte, tous les programmes terminent en un temps fini. L'analyse du temps de calcul des programmes est donc possible au pire en exécutant ceux-ci. L'introduction d'expressions récursives ou de types récursifs détruit cette propriété.

λ-calcul typé d'ordre ω + expressions récursives (λTω+Rec) :

Afin de pouvoir exprimer la récursion, on ajoute une construction à la syntaxe des expressions :

$$e ::= \ldots \mid (\texttt{rec}\ v\ e)$$

Dans une syntaxe étendue au test (l'introduction du test n'influe pas sur les propriétés dont nous discutons), la fonction factorielle s'écrit :

```
(rec f λv:nat.if (= v 0) then 1 else (× v (f (- v 1))))
```

Théorème 2.12 λTω+Rec *n'est pas normalisable.*

Preuve 2.12 L'application de la fonction définie par `(rec f λv:nat.f(v))` à une expression quelconque ne possède pas de forme normale. □

Ce λ-calcul ne possède pas de type récursif. Tout objet possède une taille bornée statiquement. Cette propriété disparaît avec le suivant λ-calcul typé.

λ-calcul typé d'ordre ω + types récursifs (λTω+TRec) :

On ajoute aux descriptions de type le constructeur de types récursifs suivant.

$$t ::= \ldots \mid (\texttt{trec}\, w\, t)$$

Théorème 2.13 *En termes de pouvoir d'expression, le* λTω+TRec *contient* λTω+Rec*.*

Preuve 2.13 L'opérateur de récursion `rec` est exprimable. Par exemple, sur les entiers, `(rec f` e`)` est égal à `(Y` λ`f:nat` $\rightarrow$ `nat.`e`)` où `Y` est un opérateur de point fixe. Ce dernier nécessite un type récursif et n'est donc pas exprimable dans les λ-calcul s typés précédents. □

Théorème 2.14 λTω+TRec *n'est pas normalisable.*

Preuve 2.14 Il contient λTω+Rec qui ne l'est pas. □

L'existence de temps de calcul bornés par une constante calculable statiquement (sans exécution du programme) est liée à la propriété de normalisation forte (SNP). Ce problème devient indécidable dans les langages de programmation ayant pour modèle λTω+TRec (ou à défaut λTω+Rec). Nous intéressant aux temps de calcul dans les langages de type λTω+TRec, nous serons très préoccupés par les constructeurs `trec` et `rec`, ainsi que par la possibilité d'exprimer l'opérateur `Y`.

2 Systèmes de typage et preuves

Dans cette thèse, les systèmes de typage sont tous présentés de manière identique. Chaque langage est d'abord spécifié à l'aide d'une grammaire BNF. Ensuite le modèle d'évaluation et le système de typage sont spécifiés à l'aide de jeux de règles sémantiques. Une preuve de consistance entre le modèle d'évaluation et le système de typage est alors proposée. Pour finir, un algorithme de vérification (ou d'inférence) de type est fournit accompagné de sa preuve de correction. Nous présentons, dans cette section, le formalisme servant à spécifier les sémantiques (dynamique et statique) et les techniques de preuves utilisées.

2.1 Spécifications sémantiques

Une sémantique (statique ou dynamique) est spécifiée à l'aide d'un ensemble de règles de déduction. Une règle est un doublet constitué par un ensemble de jugements (encore appelés séquents), les prémisses de la règle, et d'un jugement, la conclusion de la règle. Une règle est graphiquement représentée comme suit :

$$\frac{\begin{array}{c} Env^1 \vdash\ e^1\ r_1\ a_1^1 \ldots r_n\ a_n^1 \\ \vdots \\ Env^p \vdash\ e^p\ r_1\ a_1^p \ldots r_n\ a_n^p \end{array}}{Env \vdash\ e\ r_1\ a_1 \ldots r_n\ a_n}\ [\text{Nom}]$$

L'ensemble des prémisses est placé au dessus de la barre horizontale. La conclusion l'est en dessous. Les e^1, ..., e^p et e sont des expressions du langage. Les r_1, ..., r_n sont

des relations binaires liant, respectivement, l'expression aux valeurs a_1, ..., a_n. Il n'y a généralement qu'une ou deux relations. Enfin, les Env^1, ..., Env^p et Env sont des environnements, non obligatoirement présents, associant des identificateurs à des valeurs. Un jugement, $Env \vdash e\ r\ a$, est valide si, dans l'environnement Env, l'expression e est liée à la valeur a par le relation r. Par exemple, quel que soit l'environnement Env, le jugement $Env \vdash$ `True`:`bool` où la relation ":" signifie *est de type* est vrai. L'environnement vide est noté []. L'environnement $Env[i\ r\ a]$ représente un environnement dans lequel l'identificateur i est lié à la valeur a par la relation r. Cela peut être dû à une extension (i n'est pas lié dans Env) ou à une modification (i est lié dans Env, mais cette liaison est ignorée). Pour terminer, la conclusion d'une règle est prouvable si toutes les prémisses le sont. A noter qu'une prémisse peut aussi être une relation mathématique simple comme dans la règle [Var] ci-dessous.

Voila un exemple simple : la spécification du système de typage du λ-calcul simplement typé. Les trois règles spécifient respectivement le typage d'une variable, le typage de l'application d'une expression (e_0) à une autre (e_1) et le typage de l'abstraction d'une expression sur une variable (x).

$$\frac{[x : t] \subseteq T}{T \vdash x\ :\ t}\ [\text{Var}] \qquad \frac{\begin{array}{l} T \vdash e_0\ :\ t_1 \rightarrow t_0 \\ T \vdash e_1\ :\ t_1 \end{array}}{T \vdash (e_0\ e_1)\ :\ t_0}\ [\text{Apply}]$$

$$\frac{T[x : t_1] \vdash e\ :\ t_0}{T \vdash \lambda \mathtt{x}{:}\mathtt{t}_1.\mathtt{e}\ :\ t_1 \rightarrow t_0}\ [\text{Lambda}]$$

La relation notée ":" signifie *est de type* et celle notée $\subseteq$ est l'inclusion étendue aux fonctions. La conclusion de la règle [Var] signifie "Dans l'environnement de typage T, la variable x est de type t". La prémisse de cette règle, comme c'est le cas occasionnellement, n'est pas un jugement, mais une assertion. Dans le cas présent, elle signifie que la relation associant la variable x au type t appartient à l'environnement T.

Généralement, les systèmes de règles sont conçus par induction sur la structure des expressions. Si à chaque type d'expression est associée une règle, le système spécifié est déterministe. Il est parfois nécessaire d'ajouter au système une règle explicitant une propriété devant être vérifiée par toutes les expressions ou d'avoir deux règles pour un même type d'expression. Le système est alors non déterministe et peut, par exemple, être mis en œuvre par un algorithme utilisant la technique du retour arrière.

2.2 Technique de preuve

Une fois le langage spécifié (syntaxe, sémantiques et algorithmes), quelques preuves sont nécessaires afin de valider la consistance des sémantiques et la correction des algorithmes. Le principe de preuve obéit au schéma suivant :

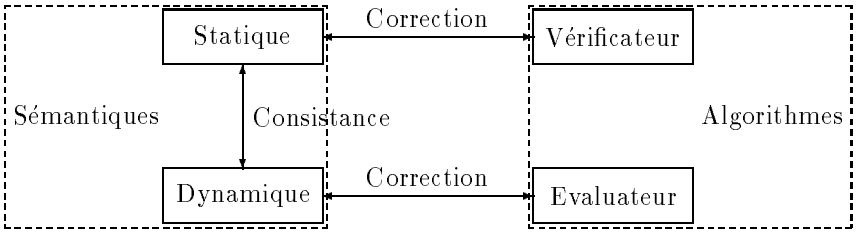


Schéma de validation.

A gauche figurent les spécifications des sémantiques (statique et dynamique). A droite figurent les algorithmes correspondants. Les flèches symbolisent les preuves nécessaires à la validation du système. La preuve de consistance entre la sémantique statique et dynamique (symbolisée par la flèche verticale) stipule que les valeurs calculées lors de l'exécution correspondent bien à celles décrites par le typage. La preuve de correction entre une spécification sémantique et son algorithme est généralement découplée en deux. La preuve d'*exactitude* vérifie que les résultats calculés par l'algorithme sont bien ceux spécifiés par la sémantique. La preuve de *complétude* assure que toutes les valeurs spécifiées par le système sont calculées.

Ne nous intéressant qu'aux aspects liés au typage, nous ne présentons pas d'algorithme d'évaluation, ni par conséquent, de preuve de correction avec la sémantique dynamique. L'étude des principes d'évaluation des programmes applicatifs constitue un domaine bien connu et maîtrisé [AS85,SJ87,F88]. Mosses, Lee et Deutsch [M76,LP87,Deu89] ont même proposé des systèmes automatiques de génération d'évaluateurs ou de compilateurs .

2.3 Induction de point fixe maximal

Notre langage n'est pas purement applicatif et, par conséquent, interdit l'utilisation de l'induction classique pour prouver la consistance des sémantiques statique et dynamique. Comme l'a proposé Tofte dans sa thèse [T88], nous utilisons l'induction de point fixe maximal qui fonctionne par élimination des cas incorrects plutôt que par construction des cas valides.

La preuve de consistance entre la sémantique dynamique et la sémantique statique se fait par induction sur la longueur de dérivation des expressions. On utilise, pour cela, une relation de consistance entre la valeur (résultat d'exécution du programme) et le type du résultat du programme. Cette relation de consistance valeur/type est aisée à définir pour les langages purement applicatifs. L'introduction d'effets de bords fait que les valeurs, résultats d'exécutions des programmes, peuvent dépendre de structures de données conservées en mémoire. La relation de consistance valeur/type doit donc être définie modulo l'état mémoire. Le second problème introduit est lié à d'éventuels cycles dans les structures de données mémoire. Prenons l'exemple de deux listes mutuellement imbriquées. La consistance de la première liste avec son type n'est vraie que modulo la consistance de la deuxième avec le sien (le problème se posant de manière identique pour la deuxième). Il n'est donc plus possible de valider la relation de consistance mais seulement de vérifier sa cohérence. L'induction standard (induction de point fixe minimum) ne peut plus être utilisée car ne permet que la validation d'une relation. Elle est basée sur la validité de cas de base et vérifie la validité des cas construits sur les cas précédents. Elle est donc inapte à vérifier la consistance de l'exemple de nos deux listes. L'induction de point fixe maximal, par contre, fonctionne par élimination des cas réfutables (i.e. que l'on peut prouver faux). Tous les cas consistants mais non validables, comme celui de nos deux listes, sont donc admis.

Nous présentons maintenant de façon plus formelle le principe d'induction de point fixe maximal. Soient S, V et T respectivement les ensembles des mémoires, des valeurs et des types. Nous appelons Q tout sous-graphe de l'ensemble produit $S \times V \times T$. Enfin, soit $\mathcal{U}$

l'ensemble des graphes, l'ensemble des parties de $S \times V \times T$. La relation de consistance entre les valeurs et les types est représentée par la fonction notée $\mathcal{F}$:

$$\begin{array}{rcccc} \mathcal{F} & : & \mathcal{P}(\mathcal{U}) & \rightarrow & \mathcal{P}(\mathcal{U}) \\ & & \mathcal{Q} & \mapsto & \mathcal{F}(\mathcal{Q}) \end{array}$$

Elle prend un graphe Q et rend un graphe Q' dont les triplets $\langle s, v, t \rangle$, réfutables en une étape depuis ceux contenus dans Q, sont éliminés. Par exemple, si l'on sait que la valeur v est associée au type t, l'application de $\mathcal{F}$ éliminera tous les couples valeur/type où les pointeurs sur v ne sont pas associés au type $t\ ref$ (type des références aux valeurs de type t).

La fonction $\mathcal{F}$ doit être construite monotone :

$$\mathcal{Q} \subseteq \mathcal{Q}' \Rightarrow \mathcal{F}(\mathcal{Q}) \subseteq \mathcal{F}(\mathcal{Q}')$$

Elle possède ainsi un point fixe minimal, $\mathcal{F}\uparrow$, et un point fixe maximal, $\mathcal{F}\downarrow$, sur $\mathcal{U}$. L'ensemble point fixe minimal de $\mathcal{F}$ est construit sur un ensemble vide d'hypothèses. A la première application de $\mathcal{F}$, l'ensemble Q contient tous les triplets validables sans prérequis (i.e. les cas de base). Chaque nouvelle application de $\mathcal{F}$ pourra augmenter l'ensemble Q avec de nouveaux triplets uniquement validables avec les précédents. Après un certain nombre d'étapes (peut être transfini), on atteint le plus petit point fixe de $\mathcal{F}$ contenant tous les triplets dont la validité (au sens de la relation codée par $\mathcal{F}$) est vérifiable.

Par opposition, l'ensemble, point fixe maximal de $\mathcal{F}$, contient tous les triplets non réfutables, car construit sur le graphe complet. Chaque application de $\mathcal{F}$ élimine les cas apparaissant comme réfutables dans l'ensemble courant. L'ensemble $\mathcal{F}\downarrow$ contient donc tous les triplets ne pouvant être réfutés, donc cohérents.

En fait, on n'utilise pas directement l'ensemble point fixe maximal de $\mathcal{F}$, mais la propriété suivante :

$$Q \subseteq \mathcal{F}(Q) \implies Q \subseteq \mathcal{F}\downarrow$$

qui signifie que tout ensemble vérifiant $Q \subseteq \mathcal{F}(Q)$ est sous-ensemble du point fixe maximal de $\mathcal{F}$. Ces sous-ensembles contiennent donc un jeu de triplets cohérent en regard de la relation de consistance représentée par $\mathcal{F}$. Le principe de démonstration consiste à construire un ensemble de triplets correspondant à la propriété à démontrer et à vérifier que cet ensemble est contenu dans son image par $\mathcal{F}$.

3 Langages fortement typés

Deux tendances, ayant chacune leurs avantages, s'opposent sur le style de typage : implicite ou explicite. Le typage implicite libère le programmeur de la contrainte de l'écriture des types. Les types ainsi reconstruits sont les plus généraux et favorisent la réutilisation du code. Cependant, le programmeur ne peut pas exprimer directement le polymorphisme des fonctions et ne peut le faire qu'à travers des constructeurs spéciaux (ex : le `let` de *SML*). De plus, le reconstructeur de type, utilisant un algorithme d'unification, interdit généralement l'auto-application. La récursion est donc introduite dans le langage de façon ad hoc (ex : le `rec` de *SML*) et nécessite un traitement particulier dans le reconstructeur de type ainsi que dans le modèle théorique. Malgré un alourdissement du code, le typage explicite permet au programmeur de gérer lui-même le polymorphisme. Il est alors possible, à tout moment, d'abstraire une expression sur un type ou de projeter une expression polymorphe sur un type. L'algorithme de vérification n'utilisant pas de fonction d'unification autorise

l'écriture d'auto-application et réduit au λ-calcul le modèle théorique nécessaire. Bien sûr, le système idéal est un reconstructeur partiel de type qui obligerait le programmeur à ne donner uniquement que ce qui n'est pas recalculable [OG89].

Le calcul du temps d'exécution est directement lié à l'aspect auto-application. Un programme implicitement typé n'utilisant pas l'opérateur de récursion sur les expressions est fortement normalisable. Le temps d'exécution est alors aisément calculable statiquement. Le typage implicite engendrera donc des systèmes de calcul de temps uniquement axés sur l'analyse des constructions récursives (`rec`). Ces propriétés ne sont plus vérifiées dans le cadre du typage explicite où `Y` est définissable.

Nous présentons deux langages à titre d'exemple. Le premier, *FX'87*, conçu au MIT comme langage pour le parallélisme, est explicitement typé. Le deuxième, conçu comme méta-langage du démonstrateur de théorèmes nommé LCF, l'est implicitement. Tout deux offrent les fonctions d'ordre supérieur et la programmation par effets de bord.

3.1 Le langage *FX*

Le langage *FX*[4] est un langage applicatif (il autorise la manipulation des fonctions comme objet de premier ordre) et impératif (il possède les primitives `new`, `set` et `get`). *FX* est fortement typé et utilise la discipline de typage avec *sorte*[5] de MacCracken [M79]. Les types des expressions étant fournis par le programmeur, le vérificateur de type les contrôle avant de les utiliser. Pour cela, il utilise des *sortes* (types des types). La version *FX87* [GJLS87,JD89] utilise un système de typage totalement explicite. Le pouvoir d'expression du noyau purement applicatif correspond à celui du λ-calcul λTω+TRec. Par commodité, un langage de programmation fortement normalisable a été ajouté aux types (essentiellement constitué d'un mécanisme d'abstraction et d'application). Le polymorphisme est explicite et placé sous la responsabilité du programmeur. Quelques extensions récentes de *FX* dégagent le programmeur de l'écriture de quelques informations de types. Le langages devient en partie typé implicitement avec, principalement, la reconstruction des types des abstractions [OG89,JG91,TJ91a].

Au départ, le langage *FX* a été conçu pour la programmation des machines multiprocesseurs. Dans ce but, le système de typage intègre des informations sur les régions de la mémoire dans lesquelles sont allouées les structures de données et sur les effets de bord possibles lors de l'exécution des programmes. Lors de la rédaction d'un programme, il est de la responsabilité du programmeur de spécifier dans quelle région (abstraite) de la mémoire résident les objets manipulés. Il devra ensuite dire au vérificateur de type quels sont les effets (de bord) des fonctions constituant le programme. Cette fonctionnalité permet au programmeur d'exprimer implicitement, de contrôler indirectement, le parallélisme des expressions des programmes. Deux fonctions manipulant des objets résidant dans une unique région devront être exécutées en séquence. Si, par contre, le programmeur place les objets dans deux régions distinctes, les fonctions pourront être exécutées en parallèle. Les informations d'effet permettent aussi d'autres extensions du langage comme l'ajout de la fonction `call/cc` [JG89] ou l'introduction de modules de première classe [SG89]. Elles permettent aussi aux compilateurs pour machines séquentielles d'effectuer des optimisations jusqu'alors impossibles comme la *mémoisation*[6] des fonctions pures, l'élimination des sous-expressions communes ou l'allocation sur la pile des objets locaux résidant dans des régions locales à la fonction.

La démarche prise dans la conception des systèmes de typage présentés dans les chapitres

[4] Prononcer *éfexe*, car correspond à side-`effect`. L'homophonie disparaît avec la traduction française.
[5] L'Anglais utilise *kind*, difficile à traduire en Français.
[6] Technique qui consiste à conserver des résultats afin d'éviter un recalcul ultérieur

suivants est très proche de celle du langage *FX* . Nous donnons donc dans cette section quelques éléments permettant d'acquérir une compréhension intuitive de ce langage. Toute expression est typée. Ce type est fourni par le programmeur et doit donc être vérifié avant usage. Pour ce faire, les types sont eux-mêmes *typés* avec des *sortes* (types de types). Par exemple, la sorte de `bool` est `type`. Les régions et effets se définissent comme suit.

Région : Une région abstrait un ensemble arbitraire de locations mémoires. Le programmeur dispose en fait d'un ensemble inépuisable d'identificateurs de régions, identificateurs commençant par le symbole "`@`" (e.g., `@paris`, `@sherwood` ...). Toute région est de sorte `region`. Le domaine des régions est muni d'un opérateur d'union, noté `runion`, correspondant à l'union ensembliste sur les régions vues comme des ensembles de locations. L'opérateur `runion` est un constructeur binaire de région : `(dfunc (region region) region)`. Il est de la responsabilité du programmeur de gérer le placement des structures de données dans les régions. Un doublet d'entiers résidant dans la région `@rouge` a pour type `(pairof int int @rouge)`. Ce type est différent de `(pairof int int @orange)` car la région mémoire dans laquelle se trouvent les objets décrits n'est pas la même. Les objets jamais modifiés par le programme sont alloués dans la région spéciale `@=`. L'algèbre induite par `runion` sur le domaine des régions est une algèbre ACI (associative, commutative et idempotente).

Effet : L'effet d'une expression n'accédant pas à la mémoire, comme `1` ou `(lambda ((x int)) x)`, est `pure`. Autrement, l'effet est construit avec les trois opérateurs, `alloc`, `read` et `write`, sur une région. Tout effet est de sorte `effect`. La construction d'un doublet d'entiers dans la région `@laprochaine` a pour effet `(alloc @laprochaine)`, l'accès à l'élément gauche `(read @laprochaine)` et la modification de l'élément droit `(write @laprochaine)`. Les fonctions sont dotées d'un effet latent, c.à.d. un effet éventuellement observable en cas d'application. La fonction d'identité sur les entiers a pour type `(subr pure (int) int)`. `subr` signifie qu'il s'agit du type d'une fonction, `pure` que son effet en cas d'application est inexistant, `(int)` qu'elle prend en paramètre un entier et `int` qu'elle rend en résultat un entier. On appelle *effet latent*, l'effet conservé dans le type des fonctions. Par contre, la fonction `cons` travaillant sur les entiers et la région `@lentour` a pour type `(subr (alloc @lentour) (int int) (pairof int int @lentour))`. Les effets peuvent être additionnés grâce à l'opérateur `maxeff`. Ce dernier induit sur le domaine des effets une algèbre ACUI (U signifie qu'il existe un élément unitaire : `pure`).

Polymorphisme : Le polymorphisme permet au programmeur d'abstraire une expression sur les types, les régions et les effets. Par exemple, la fonction d'identité doit pouvoir être appliquée à n'importe quel type d'objets. Elle s'écrit et a pour type :

```
Id = (plambda ((t type))
              (lambda ((x t)) x))
   : (poly ((t type)) (subr pure (t) t))
```

La forme spéciale `plambda` permet d'abstraire une expression polymorphiquement. Ce polymorphisme se retrouve exprimé dans les types grâce au constructeur `poly`. De même, la primitive `cons` doit être polymorphe sur la région dans laquelle est créé le doublet et sur les deux types des objets contenus. Dans le cas contraire, *FX* devrait fournir une opération `cons` par région et par type. Par convention, les abstractions polymorphes sont toujours curryfiées sur les régions.

```
cons : (poly ((r region))
             (poly ((t1 type)(t2 type))
                   (subr (alloc r) (t1 t2) (pairof t1 t2 r))))
```

Les fonctions polymorphes doivent être *projetées* sur une description de bonne sorte avant application. Si l'on veut créer une paire d'entiers dans la région `@lyon`, on écrit le programme suivant. Le point d'exclamation signifie : *a pour effet*. L'effet `(alloc @lyon)` découle de l'effet latent de la fonction `cons`, soit `(alloc r)`, où la variable de region `r` est `@lyon`.

```
FX> ((proj (proj cons @lyon) int int) 2 3)

(2 . 3) : (pairof int int @lyon) ! (alloc @lyon)
```

Récursion Le langage *FX* est très puissant et permet de définir les opérateurs de point fixe. *FX* doit cette puissance à la possibilité de définir des types récursifs à l'aide du constructeur de type `drec`. Dans l'opérateur de point fixe donné ci-dessous, la variable `x` est destinée à être appliquée à elle-même. Son type est donc celui d'une fonction dont le paramètre est de type identique à celui de `x`. Le type de `x` est donc, par définition, récursif.

```
Y = (plambda ((t type))
       (lambda ((f (subr pure
                         ((subr pure () (subr pure (t) t)))
                         (subr pure (t) t))))
          ((lambda ((x (drec t' (subr pure
                                      (t')
                                      (subr pure (t) t)))))
              (f (lambda () (x x))))
           (lambda ((x (drec t' (subr pure
                                      (t')
                                      (subr pure (t) t)))))
              (f (lambda () (x x)))))))
  : (poly ((t type))
       (subr pure
             ((subr pure
                    ((subr pure () (subr pure (t) t)))
                    (subr pure (t) t)))
             (subr pure (t) t)))
```

Ne pas s'affoler à la lecture de ce programme! En effet, l'opérateur de point fixe n'est pas, par nature, une fonction simple à saisir et cela à cause des auto-applications. Une section entière lui est consacrée au chapitre 4. Quand aux déclarations de types, elles peuvent laisser croire, sur cet exemple, à un alourdissement du code. Cela est peut fréquent en pratique où celles-ci aident le programmeur à spécifier les objets qu'il manipule.

Cette propriété de définir les opérateurs de point fixe influence directement les décisions prises pour notre langage explicitement typé (chapitre 4). En effet, la récursion et donc les temps d'exécution difficiles à calculer peuvent apparaître dans toute fonction définie par l'utilisateur et non pas uniquement dans celles utilisant explicitement la forme spéciale de récursion.

3.2 Le langage *Standard ML*

Le langage *Standard ML* permet la manipulation des fonctions d'ordre supérieur et des primitives d'effet de bord. Il est typé implicitement, c.à.d. que le programmeur ne fournit pas les informations de type qui sont reconstruites dans la phase statique précédant l'évaluation. Le pouvoir d'expression du noyau purement applicatif est équivalent à λT+Rec (donc inférieur à celui de *FX*). Les arguments formels possèdent par défaut un type monomorphe, la quantification universelle n'étant obtenue qu'a travers la définition de variables locales (construction `let`) :

```
- fn x => x ;
val it = fn : 'a -> 'a

- (fn x => x) 3 ;
val it = 3 : int

- (fn f => (f(3),f("4"))) fn x => x ;
Error: operator and operand don't agree (tycon mismatch)
  operator domain: int
  operand:         string
  in expression:   f "4"

- let val id = (fn x => x) in (id(3),id("4")) end ;
val it = (3,"4") : int * string
```

Le symbole `-` est le prompt de l'interpréteur. Le mot clé `val` signifie que l'on définit une nouvelle valeur qui sera associée à l'identificateur suivant `val`. Les symboles `fn` et `=>` correspondent aux λ et "." du λ-calcul . L'application d'une fonction à une valeur s'écrit comme en mathématique (i.e. `f(x)`). Les parenthèses autour du paramètre peuvent être omises dans les cas simples. La ligne, suivant celle commençant par `-`, constitue la réponse de l'interpréteur après reconstruction du type et évaluation de l'expression. Les variables locales sont définies avec la construction `let` <*définition*> `in` <*expression*> `end ;`.

Dans le premier cas la fonction d'identité se voit attribuer un type monomorphe (`'a -> 'a`). On peut donc l'appliquer à un paramètre entier (`3`) mais pas aux deux éléments d'un couple (la deuxième application, `f "4"`, est refusée). Par contre, une fois la fonction `id` définie localement, l'application aux deux éléments est autorisée, le type de l'identité ayant été quantifié sur les variables de type libres. La reconstruction impose d'autre limitations. Prenons l'exemple de la fonction `in?`, définie comme suit en *FX* :

```
in? ≡ (plambda ((setof (dfunc (type) type)))
         (lambda ((add-one (poly ((t0 type))
                                 (subr pure (t0 (setof t0)) (setof t0))))
                  (included? (poly ((t0 type))
                                   (subr pure ((setof t0) (setof t0)) bool))))
            (plambda ((t type))
               (lambda ((x t) (s (setof t)))
                  ((proj included? t) ((proj add-one t) x s) s)))))
```

testant l'appartenance d'un élément à un ensemble. Celle-ci est abstraite sur le constructeur d'ensembles `setof` et peut ainsi manipuler n'importe quelle implémentation des ensembles. L'équivalent *SML* sans déclaration explicite de type est :

```
- fn (add_one,included) => fn (x,s) => included(add_one(x,s),s) ;
val it = fn : ('a * 'b -> 'c) * ('c * 'b -> 'd) -> 'a * 'b -> 'd
```

où la notion d'abstraction sur le constructeur `setof` ne peut apparaître. *SML* offre, grâce à l'utilisation de modules, la possibilité de définir une fonction `in?` proche de celle définie en *FX*.

Le reconstructeur de type utilisant un algorithme d'unification, l'auto-application n'est pas autorisée sans aide explicite du programmeur. Prenons l'exemple de la fonction non typée λ`x.(x x)` :

```
- val Y = fn x => x(x) ;
Error: operator is not a function
  operator: 'S
  in expression: x x
```

Le reconstructeur de type la refuse, n'étant pas capable d'unifier `'a -> 'b` avec `'a`. Par contre, après définition d'un type récursif, l'auto-application devient possible à condition de manager explicitement les déroulages du type récursif.

```
- datatype 'a trec = func of 'a trec -> 'a ;
datatype 'a trec
con func : ('a trec -> 'a) -> 'a trec
```

La directive `datatype` permet de définir le type `'a trec` comme construit à partir du type `'a trec -> 'a` et grâce au constructeur de type `func`. Le constructeur `func` permet d'*enrouler* d'un cran un type récursif. L'interpréteur répond que le type `'a trec` est maintenant pris en compte avec son constructeur. Il est possible d'écrire la fonction `Y` précédente. L'expression `func(x)` représente la variable `x` coercée au type `'a trec`. La variable `x` doit donc être de type `'a trec -> 'a`.

```
- val Y = fn func(x) => x(func(x)) ;
val Y = fn : 'a trec -> 'a
```

Dans le chapitre suivant, nous présentons un système de reconstruction des types et des temps pour un langage purement implicitement typé. Les temps d'exécution non statiquement bornés découlant uniquement de la récursion, tout notre système est basé sur l'analyse des expressions construites avec l'opérateur `rec`.

4 Conclusion

Deux principaux points sont à retenir pour la lecture des chapitres suivants :

- Le principe du système d'effet (typage non standard) introduit par Gifford et Lucassen [GL86,Lu87] pour le langage *FX*. Les informations de type ne sont plus les seules à décrire les expressions du langage. Toutes autres sortes d'informations, comme les régions mémoire, les effets de bord et les temps d'exécution, peuvent être vérifiées (reconstruites) statiquement et utilisées, à la compilation dans un but d'optimisation. Détail technique sur le système d'effet, les types des fonctions sont dotés d'un effet latent, calculé à la définition de la fonction, et observable au moment de l'application.
- La différence typage explicite/implicite. Celle-ci influence directement l'analyse statique des temps à travers les problèmes liés à l'auto-application.

– – –◇◇◇– – –

Chapitre 3

Reconstruction des temps

1 Introduction

Bien qu'indécidable, l'analyse automatique de complexité des algorithmes est envisageable à condition de choisir une description de temps assez pauvre. Pour faire un parallèle avec le principe d'incertitude en physique des particules, c'est en fait le temps exact de l'algorithme qui n'est pas prévisible en général. Par contre, il n'existe pas d'obstacle théorique à la conception d'un système général et automatique calculant un résultat approché ou partiel. Nous avons pris cette optique pour concevoir le système présenté dans ce chapitre. Nous n'avons pas voulu restreindre le pouvoir d'expression du langage de programmation analysé. Ce dernier contient donc les principaux paradigmes de la programmation applicative (fonctions d'ordre supérieur, ...) et impérative (création, accès et modification d'objets en mémoire). L'échelle des temps choisie permet de faire la différence entre un temps borné par une constante (i.e. le temps d'exécution des primitives et combinaisons de primitives) et un temps non borné, qualifié de `long` (i.e. temps des fonctions récursives).

Les temps appartiennent à un treillis formé de l'ensemble des entiers positifs non nuls auxquels on a ajouté l'élément `long` représentant une quantité de temps non statiquement bornée. L'analyseur de complexité est conçu comme une extension à la phase de reconstruction des types. Comme cette dernière ne constitue pas notre préoccupation principale mais juste un moyen d'obtenir les temps, les types sont réduits au minimum nécessaire pour la reconstruction des temps. Tous les types de base (entier, booléen, ...) sont fusionnés dans l'unique type de base `base`. Ne restent que les types des fonctions, opérateurs ternaires sur le type du domaine, le type du codomaine et le temps latent. Le *temps latent* est calculé à la définition de la fonction et représente le temps maximum qu'elle peut utiliser pour s'exécuter. Ses types peuvent apparaître comme trop simples et donc inutiles. Il est à noter que notre système est conçu pour être fusionné avec un vrai système d'inférence de type et d'effet (comme [T88,TJ91a]). Cette fusion est simple et directe, les flots de contrôle des algorithmes étant identiques.

Dans ce chapitre, nous présentons le langage implicitement typé utilisé avec sa sémantique dynamique (section 2). Le programmeur ne fournit aucune information et le système d'inférence (section 3) reconstruit les types et les temps estimés des expressions. La consistance du système par rapport à un modèle classique d'évaluation est prouvée (section 4). Nous donnons l'algorithme de reconstruction (section 5) et prouvons sa correction (section 6). Ensuite, nous présentons le principe de résolution des contraintes générées par le précédent algorithme et montrons sa validité et sa terminaison (section 7). En appendice, après avoir conclu, nous donnons quelques exemples.

2 Définition du langage

La syntaxe abstraite du langage ITL[1] est définie à travers le domaine des expressions (Expr). ITL permet de définir des fonctions (`lambda`), des fonctions récursives (`rec`), d'appliquer une fonction à une valeur, de créer une nouvelle location mémoire (`new`), d'accéder à une valeur conservée en mémoire (`get`), et enfin, de modifier la valeur associée à une location en mémoire (`set`). Le domaine Id est le domaine des identificateurs de variables.

$e \in$ Expr		
$e ::= i$		
	(`lambda` (i) e)	Abstraction
	(`rec` (f i) e)	Récursion
	(e e)	Application
	(`new` e)	Création de location
	(`get` e)	Accès à une location
	(`set` e e)	Modification
$i \in$ Id		Identificateurs

Afin de compléter la définition d'ITL, on donne sa sémantique dynamique. Celle-ci est similaire à une sémantique standard de langage applicatif et impératif. L'unique différence est le maintien d'une horloge décomptant le temps écoulé depuis le début de l'exécution. Toutes les opérations élémentaires ont été considérées comme ayant un temps d'exécution de `1` *top* d'horloge. Bien sûr, tout autre jeu de constantes peut être pris pour modéliser une architecture précise ou un évaluateur particulier, sans changer le principe de notre méthode. La sémantique dynamique standard est obtenue en effaçant les informations de comptage du temps.

Nous commençons par définir les domaines sémantiques utiles à l'expression des règles. Deux d'entre eux méritent attention :

- Le domaine AbStore qui n'est en fait utilisé que dans la preuve de consistance entre la sémantique dynamique et la sémantique statique. Il correspond à un typage de la mémoire.
- Le domaine Clos des fermetures qui inclut, non pas un, mais deux environnements comme dans [HMT89]. Le second environnement est, soit vide si la fonction codée n'est pas récursive, soit constitué d'une unique liaison de variable à valeur si la fonction l'est (cf. règle D.Apply).

$l \in$	Loc	Référence
$b \in$	BVal = Bool + Num + {`unit`}	Valeurs de base
$v \in$	Val = Loc + BVal + Clos	Valeurs
$\langle i, e, E, E \rangle \in$	Clos = Id $\times$ Expr $\times$ Env $\times$ Env	Fermetures
$E \in$	Env = Id $\rightarrow$ Val	Environnements
$st \in$	Store = Loc $\rightarrow$ Val	Mémoires
$ST \in$	AbStore = Loc $\rightarrow$ Type	Mémoires abstraites

Reprise de [TJ91a], la fonction Rec suivante est utilisée dans la règle D.Apply afin de dérouler d'un cran la récursion. Elle s'applique alors au second environnement des fermetures.

[1] pour Implicitly Timed Language

$$\begin{array}{rcl} \mathrm{Rec} & : & \mathrm{Env} \to \mathrm{Env} \\ \mathrm{Rec}([\,]) & = & [\,] \\ \mathrm{Rec}([f{\leftarrow}\langle i,e,E',[\,]\rangle]) & = & [f{\leftarrow}\langle i,e,E',[f{\leftarrow}\langle i,e,E',[\,]\rangle]\rangle] \end{array}$$

Les règles sémantique sont constituées de jugements de type :

$$st, E \vdash e \to v, n, st' \;\subseteq\; \mathrm{Store} \times \mathrm{Env} \times \mathrm{Expr} \times \mathrm{Val} \times \mathrm{Num} \times \mathrm{Store}$$

où *st*,$E\vdash e \to v$,*n*,*st'* signifie *Avec la mémoire st et dans l'environnement E l'expression e s'évalue en la valeur v et modifie la mémoire en st' en n tops d'horloge.*

$$\frac{[i{\leftarrow}v] \;\subseteq\; E}{st, E \vdash i \to v, \mathtt{1}, st}\;[\text{D.Env}]$$

$$\frac{}{st, E \vdash \texttt{(lambda (}i\texttt{) }e\texttt{)} \to \langle i, e, E, [\,]\rangle, \mathtt{1}, st}\;[\text{D.Lambda}]$$

$$\frac{}{st, E \vdash \texttt{(rec (}f\ i\texttt{) }e\texttt{)} \to \langle i, e, E, [f{\leftarrow}\langle i, e, E, [\,]\rangle]\rangle, \mathtt{1}, st}\;[\text{D.Rec}]$$

$$\frac{\begin{array}{c} st, E \vdash e_0 \to \langle i, e, E', E''\rangle, n_0, st_0 \\ st_0, E \vdash e_1 \to v_1, n_1, st_1 \\ st_1, E' :: \mathrm{Rec}(E'')[i{\leftarrow}v_1] \vdash e \to v, n, st' \end{array}}{st, E \vdash \texttt{(}e_0\ \ e_1\texttt{)} \to v, 1 + n + n_0 + n_1, st'}\;[\text{D.Apply}]$$

$$\frac{\begin{array}{c} st, E \vdash e \to v, n, st_0 \\ [l{\leftarrow}v'] \;\not\subseteq\; st_0 \end{array}}{st, E \vdash \texttt{(new }e\texttt{)} \to l, n+1, St_0[l{\leftarrow}v]}\;[\text{D.New}]$$

$$\frac{\begin{array}{c} st, E \vdash e \to l, n, st_0 \\ [l{\leftarrow}v] \;\subseteq\; st_0 \end{array}}{st, E \vdash \texttt{(get }e\texttt{)} \to v, n+1, st_0}\;[\text{D.Get}]$$

$$\frac{\begin{array}{c} st, E \vdash e_0 \to l, n_0, st_0 \\ st_0, E \vdash e_1 \to v, n_1, st_1 \end{array}}{st, E \vdash \texttt{(set }e_0\ e_1\texttt{)} \to \texttt{unit}, n_0 + n_1 + 1, St_1[l{\leftarrow}v]}\;[\text{D.Set}]$$

Il n'existe qu'une règle d'évaluation par type d'expression. L'algorithme d'évaluation est donc déterministe et dirigé par la structure syntaxique des expressions. Décrivons ce schéma d'évaluation. La règle D.Env indique qu'une variable s'évalue en la valeur qui lui est associée dans l'environnement E sans modifier la mémoire *st*, et ce, en un top d'horloge. La relation $\subseteq$ dénote l'inclusion d'environnements ou de mémoires (au sens de l'inclusion des fonctions partielles). L'opérateur :: est la concaténation d'environnements. La règle D.Lambda indique qu'une lambda-expression s'évalue en une fermeture non récursive (le second environnement de la fermeture est vide). Une définition récursive, règle D.Rec, s'évalue en une fermeture récursive. Le second environnement contient alors une unique

association de la variable f à la fonction définie de manière non récursive. Lors de chaque application, règle D.Apply, cet environnement supplémentaire est "déroulé" d'un cran et permet ainsi la récursion. L'expression `new`, règle D.New, ajoute à la mémoire une nouvelle association location à valeur. L'expression `get`, règle D.Get, permet d'accéder à la valeur conservée en mémoire à une location donnée. Enfin, la règle D.Set spécifie la modification d'une valeur conservée en mémoire.

Signalons qu'il n'y a pas de règle d'évaluation pour les tests. Le noyau applicatif de notre langage est assez puissant pour permettre de les exprimer par une imbrication de fermeture.

3 Système d'inférence des types et des temps

Le système d'inférence spécifie la façon d'associer le type et le temps à chaque expression d'un programme. Il reprend les principes du système d'effets de Lucassen [Lu87], mais est plus proche de la reconstruction des régions et des effets telle que l'ont proposée Talpin et Jouvelot [TJ91a][2]. Le principe est d'inclure dans le type des fonctions un temps latent. Le *temps latent* modélise le temps maximum nécessaire aux fonctions décrites par ce type pour s'exécuter.

Le système est constitué d'un jeu de règles utilisant de nouveaux domaines propres à l'expression des types et des temps. Un temps (m) peut être un entier strictement positif, la valeur spéciale `long`, la somme de deux temps ou le produit d'un temps par un entier. La variable μ représente les variables d'unification de temps utilisées dans l'algorithme de reconstruction. Un type (t) peut être l'unique type de base `base` (décrivant les booléens, les entiers, ...), le type d'une fonction ou le type d'une référence en mémoire. Une fonction de type $t_1 \xrightarrow{m} t_0$ est de domaine t_1, de codomaine t_0 et de temps latent m. Les variables d'unification de type sont représentées par τ. L' environnement de type T lie des identificateurs de variable à des types.

$$
\begin{array}{rcl}
\delta & \in & \text{Descr} \\
\delta & ::= & t \mid m \\
 & & \\
m & \in & \text{Time} \\
m & ::= & \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \ldots \mid \texttt{long} \\
 & & m \oplus m \mid n.m \mid \mu \\
 & & \\
t & \in & \text{Type} \\
t & ::= & \texttt{base} \\
 & & t \xrightarrow{m} t \mid \texttt{ref}\ t \mid \tau \\
 & & \\
n & \in & \text{Num} \\
 & & \\
\text{T} & \in & \text{TEnv} = \text{Id} \rightarrow \text{Type}
\end{array}
$$

On peut voir l'ensemble des temps muni de l'opérateur de sommation ,$\oplus$, comme une algèbre ACZ, c.à.d. une algèbre où la loi est associative, commutative et possède un élément

[2]Le système de Lucassen est conçu, au départ, pour un typage totalement explicite. Le principe de reconstruction de Talpin et Jouvelot est totalement implicite. En fait, ces deux systèmes ont en commun la notion de région, d'effet et d'effet latent.

absorbant. Le *zéro* est la valeur `long`. Les deux dernières règles spécifient des simplifications utilisées pour l'expression des temps et n'ajoutent pas de propriété à l'algèbre.

$$\begin{array}{rl}
m_1 \oplus m_2 \sim m_2 \oplus m_1 & \text{Commutativité} \\
m_1 \oplus (m_2 \oplus m_3) \sim (m_1 \oplus m_2) \oplus m_3 & \text{Associativité} \\
m \oplus \texttt{long} \sim \texttt{long} & \text{Elément absorbant} \\
 & \\
m_1 \oplus m_2 \sim m_1 + m_2 \ \text{ ssi } \ m_i \neq \texttt{long} & \text{Additivité} \\
m \oplus n.m \sim (n+1).m & \text{Factorisabilité 1} \\
1.m \sim m & \text{Factorisabilité 2}
\end{array}$$

La relation d'inclusion sur les temps est directe. Elle est constituée de la relation d'ordre sur les entiers étendue à la valeur `long` comme supérieure à toutes les autres.

$$m \sqsubseteq m' \Longleftrightarrow \begin{cases} \mathit{vrai} & \mathit{si}\ m' \sim \texttt{long} \\ m \leq m' & \mathit{sinon} \end{cases}$$

L'intérêt d'une telle algèbre est mis en lumière par l'équation $m \sim m \oplus m_0$. De fait, quelle que soit la valeur de m_0, la seule valeur possible pour m est `long` (car `0` n'est pas un temps). C'est cette propriété qui permet d'associer un temps `long` aux fonctions récursives car produisant toujours des équations de cette forme (cf. règle S.Rec).

La relation d'équivalence sur les types est simple, modulo celle existant sur les temps. Les types des fonctions sont équivalents, règle E.Subr, si les types du domaine et du codomaine le sont et si les deux temps sont équivalents modulo les axiomes précédents. Les types des références sont équivalents, règle E.Ref, si les types des objets référencés le sont.

$$\frac{\begin{array}{c} t_0 \sim t_0{}' \\ t_1 \sim t_1{}' \\ m \sim m' \end{array}}{t_1 \xrightarrow{m} t_0 \sim t_1{}' \xrightarrow{m'} t_0{}'} \text{[E.Subr]}$$

$$\frac{t \sim t'}{\texttt{ref}\ t \sim \texttt{ref}\ t'} \text{[E.Ref]}$$

Les règles du système de typage utilisent un jugement de forme $\mathrm{T} \vdash e : t \$ m$ signifiant : *dans l'environnement de typage* T, *l'expression* e *est de type* t *et s'exécutera en un temps inférieur à* m.

$$\mathrm{T} \vdash e : t \$ m \ \subseteq \ \mathrm{TEnv} \times \mathrm{Expr} \times \mathrm{Type} \times \mathrm{Time}$$

Suivent les règles de typage des expressions. Une variable i, règle S.Env, liée au type t dans l'environnement T, est de type t et a pour temps `1`. On considère que le temps d'accès à une valeur à travers l'environnement est constant. Cela n'est évidemment vrai que dans les implémentations efficaces (tables de hachage). Noter, dans la règle S.Lambda, que le temps de l'expression e devient le temps latent de la fonction. Le même principe est utilisé pour les définitions récursives, règle S.Rec. Mais, m_0 peut ici être défini en fonction de lui-même (dans le cas d'une fonction effectivement récursive) et mener à la solution `long`. Lors de l'application, règle S.Apply, on ajoute aux temps des expressions e_0 et e_1 le temps latent de la fonction résultat de l'évaluation de e_0. Les règles de typage des primitives d'effet de bord sont directes. Enfin, la règle S.Takes est une règle d'inclusion de temps. Elle affirme

que si le temps d'exécution de e est majoré par m, alors il est majoré par tout temps m' plus grand. Cette règle clé de ce système est acceptable car conservative. Elle permet, dans la règle d'application S.Apply (où l'égalité des types de l'argument et du paramètre formel est requise), d'utiliser en argument une fermeture de temps latent supérieur afin de rendre légale et donc typable l'application.

$$\frac{[i:t] \subseteq \mathrm{T}}{\mathrm{T} \vdash i : t \,\$\, \mathtt{1}} \text{[S.Env]}$$

$$\frac{\mathrm{T}[i : t_1] \vdash e : t_0 \,\$\, m}{\mathrm{T} \vdash \mathtt{(lambda\ (}i\mathtt{)}\ e\mathtt{)} : t_1 \xrightarrow{m} t_0 \ \$\ \mathtt{1}} \text{[S.Lambda]}$$

$$\frac{\mathrm{T}[f : t_1 \xrightarrow{m_0} t_0\,, i : t_1] \vdash e : t_0 \,\$\, m_0}{\mathrm{T} \vdash \mathtt{(rec\ (}f\ i\mathtt{)}\ e\mathtt{)} : t_1 \xrightarrow{m_0} t_0 \ \$\ \mathtt{1}} \text{[S.Rec]}$$

$$\frac{\begin{array}{l}\mathrm{T} \vdash e_0 : t_1 \xrightarrow{m} t_0 \ \$\ m_0 \\ \mathrm{T} \vdash e_1 : t_1 \ \$\ m_1\end{array}}{\mathrm{T} \vdash \mathtt{(}e_0\ \ e_1\mathtt{)} : t_0 \ \$\ m_0 \oplus m_1 \oplus m \oplus \mathtt{1}} \text{[S.Apply]}$$

$$\frac{\mathrm{T} \vdash e : t \,\$\, m}{\mathrm{T} \vdash \mathtt{(new}\ e\mathtt{)} : \mathtt{ref}\ t \ \$\ m \oplus \mathtt{1}} \text{[S.New]}$$

$$\frac{\mathrm{T} \vdash e : \mathtt{ref}\ t \,\$\, m}{\mathrm{T} \vdash \mathtt{(get}\ e\mathtt{)} : t \ \$\ m \oplus \mathtt{1}} \text{[S.Get]}$$

$$\frac{\begin{array}{l}\mathrm{T} \vdash e_0 : t_0 \,\$\, m_0 \\ \mathrm{T} \vdash e_1 : \mathtt{ref}\ t_0 \,\$\, m_1\end{array}}{\mathrm{T} \vdash \mathtt{(set}\ e_1\ e_0\mathtt{)} : \mathtt{base} \ \$\ m_1 \oplus m_0 \oplus \mathtt{1}} \text{[S.Set]}$$

$$\frac{\begin{array}{c}\mathrm{T} \vdash e : t \ \$\ m \\ m \sqsubseteq m'\end{array}}{\mathrm{T} \vdash e : t \ \$\ m'} \text{[S.Takes]}$$

Les deux jeux de règles sémantiques (dynamique et statique), apparaissent comme naturellement consistants. Cela reste évidemment à prouver.

4 Preuve de consistance

Dans cette section nous présentons la preuve de consistance entre la sémantique dynamique et la sémantique statique. Pour cela, quelques définitions et une propriété de consistance entre les valeurs résultats d'évaluation et les types sont nécessaires. Notre langage disposant de primitives d'effets de bord, nous utilisons l'induction de point fixe maximal pour prouver quelques lemmes.

4.1 Consistance des types et des valeurs

Le principe de preuve de consistance entres les sémantiques dynamique et statique est de montrer que le type d'une expression quelconque est consistant avec la valeur en laquelle l'expression s'évalue. Par exemple, le programme `(+ 2 3)` a pour type `int` et s'évalue en la valeur *5*. On accepte sans inquiétude que la valeur *5* ait le type `int`. Cela n'est malheureusement pas aussi simple pour toutes les valeurs. Définir une relation de consistance entre les valeurs et les types revient en fait à typer les valeurs. Avant de donner cette relation, quelques définitions sont nécessaires.

Définition 3.15 (Mémoire typée) *Une mémoire typée est un couple composé d'une mémoire st et d'une mémoire abstraite ST telles que* $\mathrm{dom}(st) = \mathrm{dom}(ST)$. *On la note st* $: ST$.

Il peut paraître surprenant que la consistance des valeurs conservées dans *st* avec les types contenus dans ST ne soit pas imposée dans cette définition. Cela n'est pas utile car cette propriété est imposée plus loin dans la relation de consistance des valeurs et des types (3.17).

Définition 3.16 (Inclusion) *On dit qu'une mémoire typée,* $st : ST$, *est incluse dans une autre,* $st' : ST'$, *ce que l'on note* $st : ST \subseteq st' : ST'$, *ssi* $st \subseteq st'$ *et* $ST \subseteq ST'$.

Il s'agit juste de l'extension de la relation d'inclusion de fonctions aux paires de fonctions. Nous pouvons maintenant définir les relations de consistance. La première exprime la consistance entre les valeurs et les types. Elle est structurelle sur les valeurs et est basée sur les règles du système de typage.

Propriété 3.17 ($\models$:)

$$
\begin{aligned}
st : ST \models v : t \iff\ & \text{si } v = b \text{ alors } t \sim \texttt{base} \\
& \text{si } v = \langle i, e, E, [\,] \rangle \\
& \quad \text{alors } \exists t_1, t_0, m, \mathrm{T} \text{ t.q. } t \sim t_1 \xrightarrow{m} t_0 \wedge \\
& \qquad st : ST \models E : \mathrm{T} \wedge \mathrm{T}[i : t_1] \vdash e : t_0 \,\$\, m \\
& \text{si } v = \langle i, e, E, [f \leftarrow \langle i, e, E, [\,] \rangle] \rangle \\
& \quad \text{alors } \exists t_1, t_0, m, \mathrm{T} \text{ t.q. } t \sim t_1 \xrightarrow{m} t_0 \wedge \\
& \qquad st : ST \models E : \mathrm{T} \wedge \mathrm{T}[f : t_1 \xrightarrow{m} t_0, i : t_1] \vdash e : t_0 \,\$\, m \\
& \text{si } v = l \\
& \quad \text{alors } \exists v', t' \text{ t.q. } t \sim \texttt{ref}\ t' \wedge \\
& \qquad [l \leftarrow v'] : [l : t'] \subseteq st : ST \wedge st : ST \models v' : t'
\end{aligned}
$$

$$
st : ST \models E : \mathrm{T} \iff \mathrm{dom}(E) = \mathrm{dom}(\mathrm{T}) \wedge \forall i \in \mathrm{dom}(E), st : ST \models E(i) : \mathrm{T}(i)
$$

La deuxième relation exprime la consistance entre les temps escomptés et les temps d'exécution réels. C'est une extension de la relation $\leq$ au temps `long`.

Propriété 3.18 ($\models \leq$)

$$
\begin{aligned}
\models n \leq m \iff\ & \text{si } m \sim \texttt{long} \text{ alors } \textit{vrai} \\
& \text{si } m \not\sim \texttt{long} \text{ alors } n \leq m
\end{aligned}
$$

Reste une définition, la succession. C'est une inclusion de mémoires typées qui conserve les propriétés de consistance entre les valeurs et les types.

Définition 3.19 (Succession) *Une mémoire typée, $st':ST'$, succède à une précédente, $st:ST$, ce que l'on note $st:ST \sqsubseteq st':ST'$, ssi*

$$\left\{ \begin{array}{rcl} ST & \subseteq & ST' \\ \forall v,t,st:ST \models v:t & \Rightarrow & st':ST' \models v:t \end{array} \right.$$

4.2 Induction de point fixe maximal

Les primitives d'effets de bord permettent de créer des cycles dans les structures de données contenues en mémoire. Cet entrelacement des valeurs oblige, dans toutes les propriétés de consistance avec les types, à utiliser le principe d'induction de point fixe maximal (Voir chapitre 2, section 2.3). Cela implique la définition d'une fonction monotone représentant la relation de consistance entre les valeurs et les types.

On définit l'ensemble $\mathcal{U}$ comme l'univers des quadruplets (st:ST,v,t) et (st:ST,E,T) possibles :

$$\begin{array}{rcl} \mathcal{U} & = & \mathcal{U}_1 \uplus \mathcal{U}_2 \\ \mathcal{U}_1 & \subseteq & \text{Store} \times \text{AbStore} \times \text{Val} \times \text{Type} \\ \mathcal{U}_2 & \subseteq & \text{Store} \times \text{AbStore} \times \text{Env} \times \text{TEnv} \end{array}$$

La fonction $\mathcal{F}$, représentant la relation de consistance, est définie sur des ensembles $\mathcal{Q}$, parties de $\mathcal{U}$:

$$\begin{array}{rcccl} \mathcal{F}: & \mathcal{P}(\mathcal{U}) & \longrightarrow & \mathcal{P}(\mathcal{U}) \\ & \mathcal{Q}_1 \uplus \mathcal{Q}_2 & \longmapsto & \mathcal{F}_1(\mathcal{Q}_1 \uplus \mathcal{Q}_2) \uplus \mathcal{F}_2(\mathcal{Q}_1 \uplus \mathcal{Q}_2) \end{array}$$

Les composantes de $\mathcal{F}$, $\mathcal{F}_1$ et $\mathcal{F}_2$, sont construites à partir des deux composantes de la relation de consistance entre les valeurs et les types. La relation $st:ST \models v:t$ est remplacée par l'appartenance du quadruplet correspondant, (st:ST,v,t), à l'ensemble $\mathcal{Q}_1$ (même principe pour $st:ST \models E:\mathrm{T}$).

$$\begin{array}{rcl} \mathcal{F}_1(\mathcal{Q}_1 \uplus \mathcal{Q}_2) & = & \{(st{:}ST, v, t) \mid \\ & & \quad \text{si } v = b \text{ alors } t \sim \texttt{base} \\ & & \quad \text{si } v = \langle i, e, E, [\,] \rangle \\ & & \qquad \text{alors } \exists t_1, t_0, m, \mathrm{T} \text{ t.q. } t \sim t_1 \xrightarrow{m} t_0 \wedge \\ & & \qquad\quad (st{:}ST, E, \mathrm{T}) \in \mathcal{Q}_2 \wedge \mathrm{T}[i : t_1] \vdash e : t_0 \,\$\, m \\ & & \quad \text{si } v = \langle i, e, E, [f \leftarrow \langle i, e, E, [\,] \rangle] \rangle \\ & & \qquad \text{alors } \exists t_1, t_0, m, \mathrm{T} \text{ t.q. } t \sim t_1 \xrightarrow{m} t_0 \wedge \\ & & \qquad\quad (st{:}ST, E, \mathrm{T}) \in \mathcal{Q}_2 \wedge \mathrm{T}[f : t_1 \xrightarrow{m} t_0, i : t_1] \vdash e : t_0 \,\$\, m \\ & & \quad \text{si } v = l \\ & & \qquad \text{alors } \exists v', t' \text{ t.q. } t \sim \texttt{ref}\ t' \wedge \\ & & \qquad\quad [l \leftarrow v'] : [l : t'] \subseteq st : ST \wedge (st{:}ST, v', t') \in \mathcal{Q}_1 \} \end{array}$$

$$\begin{aligned}\mathcal{F}_2(\mathcal{Q}_1 \uplus \mathcal{Q}_2) \quad = \quad & \{(st{:}ST, E, \mathrm{T}) | \\ & \mathrm{dom}(E) = \mathrm{dom}(\mathrm{T}) \wedge \forall i \in \mathrm{dom}(E), (st{:}ST, E(i), \mathrm{T}(i)) \in \mathcal{Q}_1\}\end{aligned}$$

Noter que l'on peut définir $\mathcal{F}$ en utilisant les règles d'évaluation plutôt que les règles de typage. Cela dit, l'opérateur $\mathcal{F}$ tel qu'il est défini est monotone sur le treillis formé des parties de $\mathcal{U}$et ordonné par la relation d'inclusion. Il n'est donc pas abusif de parler de plus petit ou plus grand point fixe de $\mathcal{F}$.

Propriété 3.20 (Monotonicité) *L'opérateur $\mathcal{F}$ est monotone sur le treillis formé de l'ensemble $\mathcal{P}(\mathcal{U})$ et ordonné par l'inclusion $\subseteq$.*

Preuve Soient deux ensembles quelconques de triplets $\mathcal{Q}$ et $\mathcal{Q}'$tels que $\mathcal{Q} \subseteq \mathcal{Q}'$. Montrer que $\mathcal{F}$ est monotone consiste à montrer que la relation d'inclusion est conservée par $\mathcal{F}$(c.a.d. $\mathcal{F}(\mathcal{Q}) \subseteq \mathcal{F}(\mathcal{Q}')$). Il y a deux cas.

Cas $\boxed{q_1 = (st{:}ST, v, t)}$
Supposons que $q_1 \in \mathcal{F}_1(\mathcal{Q})$, on en déduit que

- si $v = b$ alors $t \sim$ `base` et donc $q_1 \in \mathcal{F}_1(\mathcal{Q}')$.
- si $v = \langle i, e, E, [] \rangle$ alors $t \sim t_1 \xrightarrow{m} t_0$ et il existe T t.q. $(st{:}ST, E, \mathrm{T}) \in \mathcal{Q}_2$ et $\mathrm{T}[i{:}t_1] \vdash e : t_0 \$ m$. Donc, par inclusion, $(st{:}ST, E, \mathrm{T}) \in \mathcal{Q}_2{}'$ et $q_1 \in \mathcal{F}_1(\mathcal{Q}')$.
- si $v = \langle i, e, E, [f \leftarrow \langle i, e, E, [] \rangle] \rangle$ alors $t \sim t_1 \xrightarrow{m} t_0$ et il existe T t.q. $(st{:}ST, E, \mathrm{T}) \in \mathcal{Q}_2$ et $\mathrm{T}[f{:}t_1 \xrightarrow{m} t_0, i{:}t_1] \vdash e : t_0 \$ m$. Donc, par inclusion, on a $(st{:}ST, E, \mathrm{T}) \in \mathcal{Q}_2{}'$ et $q_1 \in \mathcal{F}_1(\mathcal{Q}')$.
- si $v = l$ alors $t \sim$ `ref` t' et $[l \leftarrow v'] : [l : t'] \subseteq st : ST$ et $(st{:}ST, v', t') \in \mathcal{Q}_1$. Par inclusion, on a $(st{:}ST, v', t') \in \mathcal{Q}_1{}'$ et $q_1 \in \mathcal{F}_1(\mathcal{Q}')$.

Cas $\boxed{q = (st{:}ST, E, \mathrm{T})}$
Supposons que $q_2 \in \mathcal{F}_2(\mathcal{Q})$, on en déduit que $\mathrm{dom}(E) = \mathrm{dom}(\mathrm{T})$ et $\forall i \in \mathrm{dom}(E)$ nous avons $(st{:}ST, E(i), \mathrm{T}(i)) \in \mathcal{Q}_1$. Par inclusion, on sait que $(st{:}ST, E(i), \mathrm{T}(i)) \in \mathcal{Q}_1{}'$ et donc que $q_2 \in \mathcal{F}_2(\mathcal{Q}')$. □

Prouver la $\mathcal{F}$-consistance (consistance en regard de la relation représentée par la fonction $\mathcal{F}$) d'un ensemble de quadruplets se résume à montrer qu'il est inclus dans son image par $\mathcal{F}$. Tout ensemble de quadruplets ayant cette propriété est un sous-ensemble du point fixe maximal de $\mathcal{F}$. Pour les propriétés nécessitant ce type de preuve, on construit l'ensemble de quadruplet correspondant et l'on démontre sa qualité de point fixe maximal.

4.3 Lemmes préliminaires

Quelques lemmes nous sont utiles pour la démonstration de consistance. Le premier n'est que la réécriture de la définition 3.19.

Lemme 3.21

$$\left.\begin{array}{l} st : ST \sqsubseteq st' : ST' \\ st : ST \models v : t \end{array}\right\} \Rightarrow st' : ST' \models v : t$$

Preuve Vérifié par définition. □

Le second est l'extension du premier aux environnements d'évaluation et aux environnements de typage.

Lemme 3.22

$$\left.\begin{array}{l} st:ST \sqsubseteq st':ST' \\ st:ST \models E:\mathrm{T} \end{array}\right\} \Rightarrow st':ST' \models E:\mathrm{T}$$

Preuve Depuis $st:ST \models E:\mathrm{T}$ et par la relation 3.17 nous savons que $\mathrm{dom}(E) = \mathrm{dom}(\mathrm{T})$ et $\forall i \in \mathrm{dom}(E)$, $st:ST \models E(i):\mathrm{T}(i)$. Par le lemme 3.21, on déduit que $st':ST' \models E(i):\mathrm{T}(i)$ et donc, à nouveau par la relation 3.17, que $st':ST' \models E:\mathrm{T}$. □

Le lemme 3.23 atteste de la transitivité de la relation de succession dont jouissent les mémoires typées.

Lemme 3.23 (Transitivité)

$$\left.\begin{array}{l} st:ST \sqsubseteq st':ST' \\ st':ST' \sqsubseteq st'':ST'' \end{array}\right\} \Rightarrow st:ST \sqsubseteq st'':ST''$$

Preuve Des deux hypothèses, on déduit que $st \subseteq st'$, $st' \subseteq st''$ et que pour toute valeur v, pour tout type t, les deux implications suivantes sont vérifiées.

$$\begin{array}{rcl} st:ST \models v:t & \Rightarrow & st':ST' \models v:t \\ st':ST' \models v:t & \Rightarrow & st'':ST'' \models v:t \end{array}$$

On en déduit directement que $st \subseteq st''$ et $\forall v,t,\ st:ST \models v:t \Rightarrow st'':ST'' \models v:t$. □

Le lemme 3.24 affirme qu'en fait l'inclusion de mémoires typées est suffisante car conserve les propriétés de consistance.

Lemme 3.24 (Expansion)

$$st:ST \subseteq st':ST' \Rightarrow st:ST \sqsubseteq st':ST'$$

Preuve La mémoire st' est constituée de la mémoire st étendue avec de nouvelles associations référence à valeur. Celles-ci pouvant introduire des cycles, l'utilisation du principe d'induction de point fixe maximal s'avère nécessaire. En fait, il suffit de prouver que :

$$st:ST \subseteq st':ST' \implies (\forall v,t,\ st:ST \models v:t \Rightarrow st':ST' \models v:t)$$

Pour cela, on définit $\mathcal{Q} = \mathcal{Q}_1 \uplus \mathcal{Q}_2$ avec

$$\begin{array}{rcl} \mathcal{Q}_1 & = & \{(st':ST', v, t) | st:ST \models v:t\} \\ \mathcal{Q}_2 & = & \{(st':ST', E, \mathrm{T}) | st:ST \models E:\mathrm{T}\} \end{array}$$

Il ne reste plus qu'à prouver l'inclusion $\mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q})$. Soit $q \in \mathcal{Q}$.

Cas $\boxed{q \in \mathcal{Q}_1}$
q est de la forme (st' :ST',v,t) et tel que $st:ST \models v:t$.

- si $v = b$ alors $t \sim \texttt{base}$ donc $q \in \mathcal{F}_1(\mathcal{Q})$
- si $v = \langle i, e, E, [] \rangle$ alors $t \sim t_1 \xrightarrow{m} t_0$ et $\exists$T t.q. $st:ST \models E:\mathrm{T}$ et $\mathrm{T}[i{:}t_1] \vdash e : t_0 \,\$\, m$. Cela nous suffit pour dire que $(st'{:}ST', E, \mathrm{T}) \in \mathcal{Q}_2$ et donc que $q \in \mathcal{F}_1(\mathcal{Q})$.
- si $v = \langle i, e, E, [f \leftarrow \langle i, e, E, [] \rangle] \rangle$ alors $t \sim t_1 \xrightarrow{m} t_0$ et $\exists$T t.q. $st:ST \models E:\mathrm{T}$ et $\mathrm{T}[f{:}t_1 \xrightarrow{m} t_0, i{:}t_1] \vdash e : t_0 \,\$\, m$. De même que précédemment, $(st'{:}ST', E, \mathrm{T}) \in \mathcal{Q}_2$ et $q \in \mathcal{F}_1(\mathcal{Q})$.
- si $v = l$ alors $t \sim \texttt{ref}\, t'$ et $[l \leftarrow v'] : [l : t'] \subseteq st:ST$ et $st:ST \models v' : t'$. Par la définition de $\mathcal{Q}_1$, on a $(st'{:}ST', v', t') \in \mathcal{Q}_1$ et par inclusion $[l \leftarrow v'] : [l : t'] \subseteq st' : ST'$. On en déduit que $q \in \mathcal{F}_1(\mathcal{Q})$.

Cas $\boxed{q \in \mathcal{Q}_2}$
q est de la forme $(st'{:}ST', E, \mathrm{T})$ et tel que $st:ST \models E:\mathrm{T}$. De la définition 3.17, on sait que $\mathrm{dom}(E) = \mathrm{dom}(\mathrm{T})$ et $\forall i \in \mathrm{dom}(E), st:ST \models E(i) : \mathrm{T}(i)$. On en déduit que pour tout i, $(st{:}ST, E(i), \mathrm{T}(i)) \in \mathcal{Q}_1$ et donc que $q \in \mathcal{F}_2(\mathcal{Q})$. □

Le dernier lemme concerne la modification d'une référence en mémoire. Une affectation valide (d'un point de vue typage) ne détruit pas les propriétés de consistance.

Lemme 3.25 (Modification)

$$st:ST \models v_0 : ST(l_0) \Rightarrow st:ST \sqsubseteq St[l_0 \leftarrow v_0] : ST$$

Preuve Pour la même raison que dans la preuve précédente, on utilise le principe d'induction de point fixe maximal. On doit démontrer que $\mathcal{Q} \subseteq \mathcal{F}(\mathcal{Q})$ avec les définitions suivantes :

$$\begin{aligned}
\mathcal{Q} &= \mathcal{Q}_1 \uplus \mathcal{Q}_2 \\
\mathcal{Q}_1 &= \{(st'{:}ST, v, t) | st:ST \models v:t\} \\
\mathcal{Q}_2 &= \{(st'{:}ST, E, \mathrm{T}) | st:ST \models E:\mathrm{T}\} \\
&\quad avec\ st:ST \models v_0 : ST(l_0) \\
&\quad\quad et\ St' = St[l_0 \leftarrow v_0]
\end{aligned}$$

Cas $\boxed{q \in \mathcal{Q}_1}$
q est de la forme $(st'{:}ST, v, t)$.

- si $v = b$ alors $t \sim \texttt{base}$ donc $q \in \mathcal{F}_1(\mathcal{Q})$
- si $v = \langle i, e, E, [] \rangle$ alors $t \sim t_1 \xrightarrow{m} t_0$ et $\exists$T t.q. $st:ST \models E:\mathrm{T}$ et $\mathrm{T}[i{:}t_1] \vdash e : t_0 \,\$\, m$. On en déduit que $(st'{:}ST, E, \mathrm{T}) \in \mathcal{Q}_2$ et que $q_1 \in \mathcal{F}_1(\mathcal{Q})$.
- si $v = \langle i, e, E, [f \leftarrow \langle i, e, E, [] \rangle] \rangle$ alors $t \sim t_1 \xrightarrow{m} t_0$ et $\exists$T t.q. $st:ST \models E:\mathrm{T}$ et $\mathrm{T}[f{:}t_1 \xrightarrow{m} t_0, i{:}t_1] \vdash e : t_0 \,\$\, m$. De même, on en déduit que $(st'{:}ST, E, \mathrm{T}) \in \mathcal{Q}_2$ et que $q_1 \in \mathcal{F}_1(\mathcal{Q})$.
- si $v = l$ alors $t \sim \texttt{ref}\, t'$ et $[l \leftarrow v'] : [l : t'] \subseteq st:ST$ et $st:ST \models v' : t'$. De la proposition $st:ST \models v_0 : ST(l_0)$, on déduit que $st:ST \models st'(l) : t'$ puis que $(st'{:}ST, st'(l), t') \in \mathcal{Q}_1$ et enfin que $(st'{:}ST, l, \texttt{ref}\, t') \in \mathcal{F}_1(\mathcal{Q})$. Si $l = l_0$ la démonstration est identique et s'appuie sur $st:ST \models v_0 : ST(l_0)$ et $St' = St[l_0 \leftarrow v_0]$.

Cas $\boxed{q \in \mathcal{Q}_2}$
q est de la forme $(st'{:}ST', E, \mathrm{T})$. De la définition 3.17, on sait que $\mathrm{dom}(E) = \mathrm{dom}(\mathrm{T})$ et $\forall i \in \mathrm{dom}(E), st:ST \models E(i) : \mathrm{T}(i)$. Donc, pour tout $i \in \mathrm{dom}(E)$ on a $(st'{:}ST', E(i), \mathrm{T}(i)) \in \mathcal{Q}_1$ donc $q \in \mathcal{F}_2(\mathcal{Q})$. □

4.4 Consistance

Le théorème peut être interprété comme suit : Si les deux environnements (évaluation et typage) sont consistants, si l'expression e a pour type t et pour temps m et si l'expression e s'évalue en la valeur v, la mémoire st' et un temps d'exécution de n alors il existe une mémoire typée succédant à celle du départ comprenant st', la valeur v est consistante avec le type t et le temps d'exécution n est consistant avec le temps m. Il est à noter que le théorème ne s'applique que si l'évaluation de l'expression termine.

Théorème 3.26 (SDC) *Soient st et st'des mémoires, ST une mémoire abstraite, e une expression,* T *un environnement de type, t un type, m un temps, E un environnement, v une valeur et n un entier, alors l'implication suivante est vérifiée :*

$$\left.\begin{array}{l} st : ST \models E : \mathrm{T} \\ \mathrm{T} \vdash e : t \$\, m \\ st, E \vdash e \rightarrow v, n, st' \end{array}\right\} \Longrightarrow \exists ST' \text{ t.q. } \left\{\begin{array}{l} st : ST \sqsubseteq st' : ST' \\ st' : ST' \models v : t \\ \models n \leq m \end{array}\right.$$

La preuve ne présente pas de difficulté majeure. Le lecteur pressé peut se contenter des cas `rec`, `set` et de l'application.

Preuve Par analyse des cas et induction sur le temps d'évaluation des expressions.

Cas $\boxed{e = i}$
De la règle S.Env et $\mathrm{T} \vdash i : t \$\, m$, on sait que $m \sim 1$ et que $[i : t] \subseteq \mathrm{T}$. De la règle D.Env et $st,E \vdash i \rightarrow v,n,st'$, on sait que $n = 1$, $st = st'$ et $[i \leftarrow v] \subseteq E$. En posant $ST' = ST$, il découle directement que $\models n \leq m$ et que $st : ST \models [i \leftarrow v] : [i : t]$ donc que $st : ST \models v : t$.

Cas $\boxed{e = (\texttt{lambda}\ (i)\ e)}$
De S.Lambda et $\mathrm{T} \vdash (\texttt{lambda}\ (i)\ e) : t \$\, m$, on sait que $m \sim 1$ et $t \sim t_1 \xrightarrow{m_l} t_0$ et $\mathrm{T}[i{:}t_1] \vdash e : t_0 \$\, m_l$. De D.Lambda et $st,E \vdash (\texttt{lambda}\ (i)\ e) \rightarrow v,n,st'$, on sait que $n = 1$, $v = \langle i, e, E, [] \rangle$ et $st = st'$. En posant $ST' = ST$, il découle directement que $\models n \leq m$. De l'hypothèse $st : ST \models E : \mathrm{T}$ et de la relation 3.17, on déduit que $st : ST \models v : t$.

Cas $\boxed{e = (\texttt{rec}\ (f\ i)\ e)}$
De S.Rec et $\mathrm{T} \vdash (\texttt{rec}\ (f\ i)\ e) : t \$\, m$, on sait que $m \sim 1$ et $t \sim t_1 \xrightarrow{m_l} t_0$ et $\mathrm{T}[f{:}t_1 \xrightarrow{m_l} t_0, i : t_1] \vdash e : t_0 \$\, m_l$. De D.Rec et $st,E \vdash (\texttt{rec}\ (f\ i)\ e) \rightarrow v,n,st'$, on sait que $n = 1$, $st = st'$ et $v = \langle i, e, E, [f \leftarrow \langle i, e, E, [] \rangle] \rangle$. On a directement $\models n \leq m$. On pose $ST' = ST$. De l'hypothèse $st : ST \models E : \mathrm{T}$ et de la relation 3.17, on déduit que $st : ST \models v : t$.

Cas $\boxed{e = (e_0\ \ e_1)}$
De S.Lambda et $\mathrm{T} \vdash (e_0\ \ e_1) : t \$\, m$, on sait que $t \sim t_0$, $m \sim m_0 \oplus m_1 \oplus m_l \oplus 1$, $\mathrm{T} \vdash e_1 : t_1 \$\, m_1$ et $\mathrm{T} \vdash e_0 : t_1 \xrightarrow{m_l} t_0 \$\, m_0$. De D.lambda et $st,E \vdash (e_0\ \ e_1) \rightarrow v,n,st'$, on sait que

$$\left\{\begin{array}{ll} n = n_0 + n_1 + n' + 1 & \\ st_1, E' :: \mathrm{Rec}(E'')[i \leftarrow v_1] \vdash e \rightarrow v, n', st' & (a) \\ st_0, E \vdash e_1 \rightarrow v_1, n_1, st_1 & \\ st, E \vdash e_0 \rightarrow \langle i, e, E', E'' \rangle, n_0, st_0 & \end{array}\right.$$

Par induction sur n_0, on sait qu'il existe ST_0 t.q.

$$\left\{\begin{array}{ll} st : ST \sqsubseteq st_0 : ST_0 & (d) \\ st_0 : ST_0 \models \langle i, e, E', E'' \rangle : t_1 \xrightarrow{m_l} t_0 & (e) \\ \models n_0 \leq m_0 & \end{array}\right.$$

Grâce au lemme 3.22, la proposition (d) implique $st_0 : ST_0 \models E : \mathrm{T}$. Par induction sur n_1, on sait qu'il existe ST_1 t.q.

$$\begin{cases} st_0 : ST_0 \sqsubseteq st_1 : ST_1 & (g) \\ st_1 : ST_1 \models v_1 : t_1 \\ \models n_1 \leq m_1 \end{cases}$$

Ici, se présentent deux cas : $E'' = [\,]$ ou $E'' = [f \leftarrow \langle i, e, E', [\,]\rangle]$.

- $E'' = [\,]$ De la proposition (e) et la relation 3.17, on sait qu'il existe T′ t.q. $st_0 : ST_0 \models E' : \mathrm{T}'$ et $\mathrm{T}'[i : t_1] \vdash e : t_0 \,\$\, m_l$. Comme $\mathrm{Rec}(E'') = \mathrm{Rec}([\,]) = [\,]$, la proposition (a) se réécrit $st_1, E'[i \leftarrow v_1] \vdash e \rightarrow v, n', st'$. Le lemme 3.22 et la proposition (g) impliquent $st_1 : ST_1 \models E' : \mathrm{T}'$ et $st_1 : ST_1 \models v_1 : t_1$ implique $st_1 : ST_1 \models E'[i \leftarrow v_1] : \mathrm{T}'[i : t_1]$. Maintenant, par induction sur n', nous savons qu'il existe ST' t.q.

  $$\begin{cases} st_1 : ST_1 \sqsubseteq st' : ST' \\ st' : ST' \models v : t_0 \\ \models n' \leq m_l \qquad\qquad \Rightarrow \models n \leq m \end{cases}$$

  On constate que l'on a bien $st : ST \sqsubseteq st' : ST'$ par transitivité (lemme 3.23) et aussi $st' : ST' \models v : t$.

- $E'' = [f \leftarrow \langle i, e, E', [\,]\rangle]$ De la proposition (e) et de la relation 3.17, on sait qu'il existe T′ t.q. $st_0 : ST_0 \models E' : \mathrm{T}'$ et $\mathrm{T}'[f : m_l \xrightarrow{t_1} t_0; i : t_1] \vdash e : t_0 \,\$\, m_l$. Comme $\mathrm{Rec}(E'') = [f \leftarrow \langle i, e, E', E''\rangle]$, la proposition (a) se réécrit alors $st_1, E'[f \leftarrow \langle i, e, E', E''\rangle; i \leftarrow v_1] \vdash e \rightarrow v, n', st'$. Les proposition (e), (g) et le lemme 3.21 impliquent que $st_1 : ST_1 \models \langle i, e, E', E''\rangle : m_l \xrightarrow{t_1} t_0$. Du lemme 3.22 et la proposition (g), on sait que $st_1 : ST_1 \models E' : \mathrm{T}'$ et $st_1 : ST_1 \models v_1 : t_1$ implique $st_1 : ST_1 \models E'[f \leftarrow \langle i, e, E', E''\rangle\, i \leftarrow v_1] : \mathrm{T}'[f : m_l \xrightarrow{t_1} t_0; i : t_1]$. Maintenant, par induction sur m', nous savons qu'il existe ST' t.q.

  $$\begin{cases} st_1 : ST_1 \sqsubseteq st' : ST' \\ st' : ST' \models v : t_0 \\ \models n' \leq m_l \qquad\qquad \Rightarrow \models n \leq m \end{cases}$$

  De même, on constate que l'on a bien $st : ST \sqsubseteq st' : ST'$ par transitivité (lemme 3.23) et aussi $st' : ST' \models v : t$.

Cas $\boxed{e = (\texttt{new}\ e)}$
De S.New et $\mathrm{T} \vdash (\texttt{new}\ e) : t \,\$\, m$, on sait que $m \sim m_0 \oplus 1$, $t \sim \texttt{ref}\ t_0$ et $\mathrm{T} \vdash e_0 : t_0 \,\$\, m_0$. De D.New et $st, E \vdash (\texttt{new}\ e) \rightarrow v, n, st'$, on sait que $v = l$, $n = n_0 + 1$, $st' = st_0[l \leftarrow v]$ et $st, E \vdash e \rightarrow v_0, n_0, st_0$. Par induction sur n_0, on déduit qu'il existe ST_0 t.q.

$$\begin{cases} st : ST \sqsubseteq st_0 : ST_0 \\ st_0 : ST_0 \models v_0 : t_0 \\ \models n_0 \leq m_0 \qquad\qquad \Rightarrow \models n \leq m \end{cases}$$

Soit ST'la mémoire typée $ST_0[l : t_0]$. La proposition $st_0 : ST_0 \sqsubseteq st' : ST'$ est alors vérifiée. De la relation 3.17 et $st' : ST' \models l_0 : \texttt{ref}\ t_0$ on sait que $st' : ST' \models v : t$.

Cas $\boxed{e = (\texttt{get}\ e)}$
De S.Get et $\mathrm{T} \vdash (\texttt{get}\ e) : t \,\$\, m$, on sait que $m \sim m_0 \oplus 1$ et $\mathrm{T} \vdash e_0 : \texttt{ref}\ t \,\$\, m_0$. De D.Get et

$st,E\vdash$ (get e) $\rightarrow v,n,st'$, on sait que $n = n_0 + 1$, $[l\leftarrow v] \subseteq st'$ et $st,E\vdash e_0 \rightarrow l,n_0,st'$. Par induction sur n_0, on en déduit qu'il existe st' t.q.

$$\begin{cases} st : ST \sqsubseteq st' : ST' & \\ st' : ST' \models l : \texttt{ref}\ t & (a) \\ \models n_0 \leq m_0 & \Rightarrow \models n \leq m \end{cases}$$

La proposition (a) et la relation 3.17 impliquent que $[l\leftarrow v] : [l : t] \subseteq st' : ST'$ et $st' : ST' \models v : t$.

Cas $\boxed{e = (\texttt{set}\ e_0\ e_1)}$
De S.Set et T$\vdash$ (set e_0 e_1) $: t \$ m$, on sait que $t \sim$ base, $m \sim m_1 \oplus m_0 \oplus 1$, T$\vdash e_0 : t_0$ $\$\, m_0$ et T$\vdash e_1 :$ref $t_0 \$\, m_1$. De D.Set et $st,E\vdash$ (set e_0 e_1) $\rightarrow v,n,st'$, on sait que $v =$ unit, $n = n_1 + n_0 + 1$, $st' = st_0[l\leftarrow v]$, $st_1,E\vdash e_0 \rightarrow v,n_0,st_0$ et $st,E\vdash e_1 \rightarrow l,n_1,st_1$. Par induction sur n_1, on déduit qu'il existe ST_1 t.q.

$$\begin{cases} st : ST \sqsubseteq st_1 : ST_1 & (a) \\ st_1 : ST_1 \models l : \texttt{ref}\ t_0 & (b) \\ \models n_1 \leq m_1 & (c) \end{cases}$$

Grâce au lemme 3.22, la proposition (a) implique $st_1 : ST_1 \models E :$ T. On peut maintenant faire une induction sur n_0, ce qui nous donne qu'il existe $ST' = ST_0$ t.q.

$$\begin{cases} st_1 : ST_1 \sqsubseteq st_0 : ST_0 & (d) \\ st_0 : ST_0 \models v : t_0 & \\ \models n_0 \leq m_0 & \text{et } (c) \Rightarrow \models n \leq m \end{cases}$$

les propositions (b) et (d) impliquent $st_0 : ST_0 \models l :$ ref t_0. Comme $l \in \mathrm{dom}(st_0)$, on utilise le lemme 3.24 pour déduire que $st_0 : ST_0 \sqsubseteq st_0[l\leftarrow v] : ST_0$. De plus, $st' : ST' \models$ unit : base est vérifiée. □

5 Algorithmes

Nous présentons, dans cette section, le principe de reconstruction des types et des temps. Nous présentons d'abord l'algorithme d'unification utilisé puis continuons par l'algorithme de reconstruction.

5.1 Algorithme d'unification

Il s'agit d'un unificateur syntaxique à la Robinson [R65]. Il rend en résultat la substitution la plus générale (i.e. celle qui appliquée à l'une des deux descriptions, fournit l'unificateur le plus général).

U $\in$ (Descr)^2 $\longrightarrow$ Subst

```
U(δ,δ') = case (δ,δ') in
(base,base)              => []
(τ,t) or (t,τ)           => if τ ∈ FV(t) then FAIL else [τ\t]
(t1 -m-> t0, t1' -m'-> t0')  => let s1 = U(t1,t1')
                                let s0 = U(s1 t0, s1 t0')
                                let s= U(s0(s1 m), s0(s1 m'))
```

```
                                ss0s1
(ref t,ref t')                  => U(t,t')
(μ,m) or (m,μ)                  => if μ ∈ FV(m) then FAIL else [μ\m]
else                            => FAIL
```

Il est à noter que cet algorithme est conçu pour unifier des types. S'il accepte aussi des temps, c'est uniquement pour unifier des temps latents. Ces derniers sont toujours représentés par des variables d'unification dans l'algorithme de reconstruction. Nous n'avons donc pas à unifier des sommes de temps ($\oplus$) en respectant le caractère ACZ de leur algèbre.

5.2 Algorithme de reconstruction

L'algorithme de reconstruction des types et des temps utilise l'algorithme d'unification précédent pour calculer un quadruplet formé d'un type, d'un temps, d'une substitution et d'un ensemble de contraintes. L'ensemble de contraintes est constitué d'inéquations, notées c, de la forme $\mu \geq \bigoplus_i m_i$.

$$\mathtt{c} \in \text{Const} = \text{Time} \times \text{Time} \qquad \text{Contraintes}$$

Cet ensemble de contraintes admet toujours plusieurs solutions. Celles-ci s'ordonnent suivant une extension simple de l'ordre sur les temps. La plus petite parmi l'ensemble des solutions correspond à la solution recherchée. En effet, il s'agit de la plus générale au sens de la relation d'inclusion des temps induite par la règle S.Takes.

L'algorithme fonctionne par induction sur la structure des expressions. On suppose que les expressions sont α-renommées afin d'éviter les conflits de noms des identificateurs. On considère les substitutions comme capables de gérer divers types de données, à savoir les types, les temps, les environnements ou les ensembles de contraintes. Enfin, l'algorithme fournit des quadruplets dans lesquels la substitution a déjà été appliquée aux autres membres.

$\mathtt{R} \in \text{TEnv} \times \text{Expr} \longrightarrow \text{Type} \times \text{Time} \times \text{Subst} \times \text{Const}$

```
R(T,e) = case e in
i       => if [i:t] ⊆ T then (t,1,[],∅) else FAIL
(lambda (i) e)
        => let (t,m,s,c) = R(T[i:τ],e) where τ is FRESH
           (sτ --μ--> t,1,s,c∪{μ ≥ m}) where μ is FRESH
(rec (f i) e)
        => let (t,m,s,c) = R(T[f:τ1 --μ--> τ0,i:τ1],e) where τ1, τ0 and μ are FRESH
           let s'= U(t,sτ0)
           (s'(sτ1 --sμ--> t),1,s's,s'(c∪{sμ ≥ m}))
(e0 e1) => let (t0,m0,s0,c0) = R(T,e0)
           let (t1,m1,s1,c1) = R(s0T,e1)
           let s2 = U(s1t0,t1 --μ--> τ) where μ and τ are FRESH
           (s2τ,s2(1⊕μ⊕m1⊕s1m0),s2s1s0,s2(c1∪s1c0))
(new e) => let (t,m,s,c) = R(T,e)
           (ref t,m⊕1,s,c)
(get e) => let (t,m,s,c) = R(T,e)
           let s'= U(t,ref τ) where τ is FRESH
           (s'τ,s'm⊕1,s's,s'c)
```

```
(set e0 e1)
        => let (t0,m0,s0,c0) = R(T,e0)
           let (t1,m1,s1,c1) = R(s0T,e1)
           let s = U(s1t0,ref t1)
           (base,s(s1 m0 ⊕ m1 ⊕ 1),ss1s0,s(s1c0 ∪ c1))
else    => FAIL
```

Dans le cas de l'application, l'algorithme reconstruit le type et le temps de l'expression en place fonctionnelle. Ensuite, il reconstruit ceux de l'argument avec un environnement où les substitutions de variables sont propagées. Une unification est nécessaire pour assurer l'accord du type de l'argument avec celui du domaine de la fonction.

En s'abstrayant de la complexité de l'algorithme d'unification, on peut considérer que l'algorithme de reconstruction est de complexité linéaire en la taille du programme. Il y a une étape d'induction par expression. Le nombre de contraintes contenues dans l'ensemble résultat est aussi linéaire en la taille du programme. En effet, les contraintes ne sont introduites qu'une à une et uniquement par les expressions `lambda` et `rec`. Comme les variables d'unification sont introduites de façon analogue et que l'algorithme d'unification n'en ajoute pas, les substitutions sont aussi de tailles linéaires en la taille du programme.

Comme nous l'avons déjà précisé, notre algorithme de reconstruction n'est conçu que comme une extension à un reconstructeur classique de type. Il faut noter cependant qu'il ne limite pas le pouvoir d'expression du langage. Tout programme typable est correct du point de vue reconstruction des temps. L'argument informel en faveur de cette propriété est qu'un programme dont le type est correct peut être exécuté. Son temps d'exécution est équivalent à celui que nous fournit la sémantique dynamique avec comptage. Comme celle-ci est prouvée cohérente avec le système statique, le temps reconstruit majore bien le temps réel d'exécution. Il ne nous reste qu'à prouver que l'algorithme proposé est bien l'implémentation du système statique présenté précédemment.

6 Correction

Cette section est consacrée à la preuve de correction de l'algorithme de reconstruction par rapport à la sémantique statique. Pour cela, on doit établir que les résultats de l'algorithme sont bien spécifiés par le système de typage (Cohérence) et que toutes les solutions spécifiées par le système de typage sont effectivement calculées par l'algorithme (Complétude).

Il est à noter, tout d'abord, que la sémantique statique ne spécifie que des *types* et des *temps fondamentaux*, i.e. sans variable d'unification. Cette propriété n'est évidemment pas vérifiée pour les types et les temps fournis par l'algorithme. Ces derniers, contenant des variables d'unifications définissent donc implicitement des familles de types et de temps fondamentaux. Nous devons donc établir une correspondance entre les types et temps spécifiés par le système et les familles de types et de temps exprimées par les résultats de l'algorithme. Pour cela, nous définissons deux substitutions particulières :

Définition 3.27 (Modèle) *Un modèle, noté $\mathcal{M}$, est une substitution associant des variables d'unification de temps avec des temps fondamentaux.*

Définition 3.28 (Réalisation) *Une réalisation, notée $\mathcal{S}$, est une substitution associant des variables d'unification de type avec des types fondamentaux.*

6.1 Correction de l'unification

L'algorithme d'unification suit les principes d'unification de Robinson [R65] et jouit donc de bonnes propriétés. Nous ne les énonçons que brièvement.

Théorème 3.29 (Terminaison) *L'algorithme d'unification termine.*

Preuve L'algorithme fonctionne par induction sur la structure des types. Ceux-ci ne possèdent qu'un nombre d'imbrications fini. □

Théorème 3.30 (Cohérence) *Si* $\mathtt{s} = \mathtt{U}(t_1, t_2)$ *alors les types* t_1 *et* t_2 *sont unifiables et* $\mathtt{s}t_1 \sim \mathtt{s}t_2$.

Théorème 3.31 (Complétude) *Si* $\mathtt{s} = \mathtt{U}(t_1, t_2)$ *alors pour toute substitution* $\mathtt{s}'$ *telle que* $\mathtt{s}'t_1 \sim \mathtt{s}'t_2$ *il existe une substitution* $\mathtt{s}''$*telle que* $\mathtt{s}' = \mathtt{s}''\mathtt{s}$.

Preuve (Cohérence et Complétude)
L'algorithme d'unification suit les principes de l'unification à la Robinson et jouit donc des mêmes propriétés. □

6.2 Correction de la reconstruction

Les théorèmes suivant expriment la terminaison, la cohérence et la complétude de l'algorithme de reconstruction.

Théorème 3.32 (Terminaison) *L'algorithme de reconstruction termine.*

Preuve L'algorithme parcourt structurellement les expressions, ce qui correspond à un parcours d'arbre de hauteur finie. □

Avant d'énoncer les théorèmes de cohérence et de complétude, nous avons besoin d'une définition et d'un lemme supplémentaires.

Définition 3.33 (Satisfiabilité) *Le modèle* $\mathcal{M}$ *satisfait l'ensemble de contraintes* $\mathtt{c}$*, noté* $\mathcal{M} \models \mathtt{c}$*, si l'ensemble des inéquations, obtenu en appliquant le modèle* $\mathcal{M}$ *à chaque terme des inéquations de* $\mathtt{c}$*, est valide. Un ensemble de contraintes est valide si toutes les inéquations, exprimées uniquement avec des temps fondamentaux, sont vérifiées.*

Lemme 3.34 *Si* $\forall \mathcal{M}, \mathcal{S},\ \mathcal{M} \models \mathtt{c} \ \Rightarrow\ \mathcal{MS}\mathrm{T} \vdash e : \mathcal{MS}t\,\$\,\mathcal{M}m$ *et* $\mathcal{M}' \models \mathtt{s}'\mathtt{c}$ *alors on a* $\mathcal{M}'\mathcal{S}'\mathtt{s}'\mathrm{T} \vdash e : \mathcal{M}'\mathcal{S}'\mathtt{s}'t\,\$\,\mathcal{M}'\mathtt{s}'m$.

Preuve Tout d'abord, rappelons que les substitutions sont associatives (e.g. $\mathcal{M}(\mathcal{S}\mathrm{T})$ = $\mathcal{MS}\mathrm{T}$). De $\mathcal{M}' \models \mathtt{s}'\mathtt{c}$, on déduit que $\mathcal{M}'\mathtt{s}' \models \mathtt{c}$. En utilisant l'implication, on obtient $\forall \mathcal{S}',\ \mathcal{M}'\mathtt{s}'\mathcal{S}'\mathrm{T} \vdash e : \mathcal{M}'\mathtt{s}'\mathcal{S}'t\,\$\,\mathcal{M}'\mathtt{s}'m$. La réalisation $\mathcal{S}'$ ne retourne pas de variables d'unification. Son codomaine ne peut donc pas interférer avec le domaine de la substitution $\mathtt{s}'$. Par contre, rien n'assure que les variables d'unification introduites par $\mathtt{s}'$ appartiennent au domaine de la réalisation $\mathcal{S}'$. En fait, on considère que $\mathcal{M}'$ et $\mathcal{S}'$ sont définies par défaut sur l'ensemble des variables libres de leurs arguments. Cette approche se justifie par le fait que les expressions comme $\mathcal{MS}\mathrm{T}$, $\mathcal{MS}t$ et $\mathcal{M}m$, étant utilisées dans les séquents d'assignation de type et temps, doivent être libres de toute variable d'unification. □

Théorème 3.35 (Cohérence) *Soient* $(t, m, \mathtt{s}, \mathtt{c}) = \mathtt{R}(\mathrm{T}, e)$ *et* $\mathcal{M}$ *un modèle. Pour toute réalisation* $\mathcal{S}$*, l'implication suivante est vérifiée.*

$$\mathcal{M} \models \mathtt{c} \ \Rightarrow\ \mathcal{MS}\mathtt{s}\mathrm{T} \vdash e : \mathcal{MS}t\,\$\,\mathcal{M}m$$

Preuve Par analyse de cas et induction sur la structure des expressions.

Cas $\boxed{e = i}$
Soit $(t, m, \mathtt{s}, \mathtt{c}) = \mathtt{R}(\mathrm{T}, i)$. De l'algorithme, on déduit que $[i : t] \subseteq \mathrm{T}$, $m \sim 1$, $\mathtt{s} = []$ et $\mathtt{c} = \emptyset$. Quelque soit le modèle $\mathcal{M}$, on a bien $\mathcal{M} \models \mathtt{c}$. De S.env et $[i : t] \subseteq \mathrm{T}$, on déduit que $\mathrm{T} \vdash i : t \,\$\, 1$, ce qui implique que $\mathcal{S}\mathrm{T} \vdash i : \mathcal{S}t \$ 1$ et enfin $\mathcal{M}\mathcal{S}\mathrm{T} \vdash i : \mathcal{M}\mathcal{S}t \$ \mathcal{M}1$.

Cas $\boxed{e = (\mathtt{lambda}\ (i)\ e)}$

Soit $(t, m, \mathtt{s}, \mathtt{c}) = \mathtt{R}(\mathrm{T}, (\mathtt{lambda}\ (i)\ e))$. De l'algorithme, on déduit que $t \sim \mathtt{s}\tau \xrightarrow{\mu} t_0$, $m \sim 1$, $\mathtt{c} = \mathtt{c}_0 \cup \{\mu \geq m_0\}$, $\mathtt{s} = \mathtt{s}_0$ et $(t_0, m_0, \mathtt{s}_0, \mathtt{c}_0) = \mathtt{R}(\mathrm{T}[i{:}\tau], e)$. Par induction sur e l'implication suivante est vérifiée pour tout modèle $\mathcal{M}_0$ et toute réalisation $\mathcal{S}_0$.

$$\mathcal{M}_0 \models \mathtt{c}_0 \;\Rightarrow\; \mathcal{M}_0\mathcal{S}_0\mathtt{s}_0(\mathrm{T}[i : \tau]) \vdash e : \mathcal{M}_0\mathcal{S}_0 t_0 \,\$\, \mathcal{M}_0 m_0$$

Soit un modèle $\mathcal{M}$ tel que $\mathcal{M} \models \mathtt{c}$, c.à.d $\mathcal{M} \models \mathtt{c}_0 \cup \{\mu \geq m_0\}$. On a, pour toute réalisation $\mathcal{S}$, $\mathcal{M}\mathcal{S}\mathtt{s}_0(\mathrm{T}[i{:}\tau]) \vdash e : \mathcal{M}\mathcal{S}t_0 \,\$\, \mathcal{M}m_0$ qui se réécrit comme suit $(\mathcal{M}\mathcal{S}\mathtt{s}_0\mathrm{T})[i : \mathcal{M}\mathcal{S}\mathtt{s}_0\tau] \vdash e : \mathcal{M}\mathcal{S}t_0 \,\$\, \mathcal{M}m_0$. Comme $\mathcal{M} \models \{\mu \geq m_0\}$, l'inclusion des temps $\mathcal{M}\mu \sqsupseteq \mathcal{M}m_0$ est valide. On applique alors la règle S.Takes pour obtenir $(\mathcal{M}\mathcal{S}\mathtt{s}_0\mathrm{T})[i : \mathcal{M}\mathcal{S}\mathtt{s}_0\tau] \vdash e : \mathcal{M}\mathcal{S}t_0 \,\$\, \mathcal{M}\mu$. De la règle S.Lambda, on déduit $\mathcal{M}\mathcal{S}\mathtt{s}_0\mathrm{T} \vdash (\mathtt{lambda}\ (i)\ e) : \mathcal{M}\mathcal{S}\mathtt{s}_0\tau \xrightarrow{\mathcal{M}\mu} \mathcal{M}\mathcal{S}t_0 \,\$\, 1$ ce qui implique $\mathcal{M}\mathcal{S}\mathtt{s}_0\mathrm{T} \vdash (\mathtt{lambda}\ (i)\ e) : \mathcal{M}\mathcal{S}(\mathtt{S}_0\tau \xrightarrow{\mu} t_0) \,\$\, \mathcal{M}1$.

Cas $\boxed{e = (\mathtt{rec}\ (f\ i)\ e)}$

Soit $(t, m, \mathtt{s}, \mathtt{c}) = \mathtt{R}(\mathrm{T}, (\mathtt{rec}\ (f\ i)\ e))$. De l'algorithme, on déduit que $t \sim \mathtt{s}'(\mathtt{s}_0\tau_1 \xrightarrow{\mathtt{s}_0\mu} t_0)$, $m \sim 1$, $\mathtt{s} = \mathtt{s}'\mathtt{s}_0$, $\mathtt{c} = \mathtt{s}'(\mathtt{c}_0 \cup \{\mathtt{s}_0\mu \geq m_0\})$, $\mathtt{s}' = \mathtt{U}(t_0, \mathtt{s}_0\tau_0)$ et $(t_0, m_0, \mathtt{s}_0, \mathtt{c}_0) = \mathtt{R}(\mathrm{T}[f{:}\tau_1 \xrightarrow{\mu} \tau_0, i{:}\tau_1], e)$. Par induction sur e on a pour tout modèle $\mathcal{M}_0$ et toute réalisation $\mathcal{S}_0$

$$\mathcal{M}_0 \models \mathtt{c}_0 \;\Rightarrow\; \mathcal{M}_0\mathcal{S}_0\mathtt{s}_0(\mathrm{T}[f : \tau_1 \xrightarrow{\mu} \tau_0,\, i : \tau_1]) \vdash e : \mathcal{M}_0\mathcal{S}_0 t_0 \,\$\, \mathcal{M}_0 m_0$$

Soit $\mathcal{M}$ un modèle tel que $\mathcal{M} \models \mathtt{c}$, c.à.d. $\mathcal{M} \models \mathtt{s}'(\mathtt{c}_0 \cup \{\mathtt{s}_0\mu \geq m_0\})$. On a, pour toute réalisation $\mathcal{S}$, $(\mathcal{M}\mathcal{S}\mathtt{s}'\mathtt{s}_0\mathrm{T})[f : \mathcal{M}\mathcal{S}\mathtt{s}'\mathtt{s}_0(\tau_1 \xrightarrow{\mu} \tau_0), i : \mathcal{M}\mathcal{S}\mathtt{s}'\mathtt{s}_0\tau_1] \vdash e : \mathcal{M}\mathcal{S}\mathtt{s}'t_0 \,\$\, \mathcal{M}\mathtt{s}'m_0$. Comme $\mathcal{M} \models \mathtt{s}'\{\mathtt{s}_0\mu \geq m_0\}$, l'inclusion des temps $\mathcal{M}\mathtt{s}'\mathtt{s}_0\mu \sqsupseteq \mathcal{M}\mathtt{s}'m_0$ est valide. De la règle S.Takes, on déduit alors

$$(\mathcal{M}\mathcal{S}\mathtt{s}'\mathtt{s}_0\mathrm{T})[f : \mathcal{M}\mathcal{S}\mathtt{s}'\mathtt{s}_0(\tau_1 \xrightarrow{\mu} \tau_0),\, i : \mathcal{M}\mathcal{S}\mathtt{s}'\mathtt{s}_0\tau_1] \vdash e : \mathcal{M}\mathcal{S}\mathtt{s}'t_0 \,\$\, \mathcal{M}\mathtt{s}'\mathtt{s}_0\mu$$

L'algorithme d'unification est cohérent donc $\mathtt{s}' = \mathtt{U}(t_0, \mathtt{s}_0\tau_0)$ implique que $\mathtt{s}'t_0 \sim \mathtt{s}'\mathtt{s}_0\tau_0$. On obtient la proposition $(\mathcal{M}\mathcal{S}\mathtt{s}'\mathtt{s}_0\mathrm{T})[f : \mathcal{M}\mathcal{S}\mathtt{s}'\mathtt{s}_0(\tau_1 \xrightarrow{\mu} \tau_0), i : \mathcal{M}\mathcal{S}\mathtt{s}'\mathtt{s}_0\tau_1] \vdash e : \mathcal{M}\mathcal{S}\mathtt{s}'\mathtt{s}_0\tau_0 \,\$\, \mathcal{M}\mathtt{s}'\mathtt{s}_0\mu$ nécessaire pour appliquer la règle S.Rec.

Cas $\boxed{e = (e_0\ \ e_1)}$
Soit $(t, m, \mathtt{s}, \mathtt{c}) = \mathtt{R}(\mathrm{T}, (e_0\ \ e_1))$. De l'algorithme, on déduit que

$$\begin{cases} t \sim \mathtt{s}_2\tau \\ m \sim \mathtt{s}_2(\mu \oplus m_1 \oplus \mathtt{s}_1 m_0) \oplus 1 \\ \mathtt{s} = \mathtt{s}_2\mathtt{s}_1\mathtt{s}_0 \\ \mathtt{c} = \mathtt{s}_2(\mathtt{c}_1 \cup \mathtt{s}_1\mathtt{c}_0) \\ \mathtt{s}_2 = \mathtt{U}(\mathtt{s}_1 t_0, t_1 \xrightarrow{\mu} \tau) \\ (t_1, m_1, \mathtt{s}_1, \mathtt{c}_1) = \mathtt{R}(\mathtt{s}_0\mathrm{T}, e_1) \\ (t_0, m_0, \mathtt{s}_0, \mathtt{c}_0) = \mathtt{R}(\mathrm{T}, e_0) \end{cases}$$

Par induction sur e_1 et e_0, on a

$$\begin{cases} \mathcal{M}_0 \models \mathtt{c}_0 & \Rightarrow \quad \mathcal{M}_0\mathcal{S}_0\mathtt{s}_0\mathrm{T} \vdash e_0 : \mathcal{M}_0\mathcal{S}_0 t_0 \,\$\, \mathcal{M}_0 m_0 \\ \mathcal{M}_1 \models \mathtt{c}_1 & \Rightarrow \quad \mathcal{M}_1\mathcal{S}_1\mathtt{s}_1\mathtt{s}_0\mathrm{T} \vdash e_1 : \mathcal{M}_1\mathcal{S}_1 t_1 \,\$\, \mathcal{M}_1 m_1 \end{cases}$$

Soit $\mathcal{M}$ un modèle tel que $\mathcal{M} \models \mathtt{c}$, c.à.d. $\mathcal{M} \models \mathtt{s}_2(\mathtt{c}_1 \cup \mathtt{s}_1\mathtt{c}_0)$. Par induction et en utilisant le lemme 3.34, on obtient les propositions suivantes

$$\begin{cases} \mathcal{M}\mathcal{S}\mathtt{s}_2\mathtt{s}_1\mathtt{s}_0\mathrm{T} \vdash e_0 : \mathcal{M}\mathcal{S}\mathtt{s}_2\mathtt{s}_1 t_0 \,\$\, \mathcal{M}\mathtt{s}_2\mathtt{s}_1 m_0 \\ \mathcal{M}\mathcal{S}\mathtt{s}_2\mathtt{s}_1\mathtt{s}_0\mathrm{T} \vdash e_1 : \mathcal{M}\mathcal{S}\mathtt{s}_2 t_1 \,\$\, \mathcal{M}\mathtt{s}_2 m_1 \end{cases}$$

L'unification étant cohérente, $\mathtt{s}_2\mathtt{s}_1 t_0 \sim \mathtt{s}_2(t_1 \xrightarrow{\mu} \tau)$, on peut appliquer le règle S.Apply pour obtenir $\mathcal{M}\mathcal{S}\mathtt{s}_2\mathtt{s}_1\mathtt{s}_0\mathrm{T} \vdash (e_0\ \ e_1) : \mathcal{M}\mathcal{S}\mathtt{s}_2\tau \,\$\, \mathcal{M}\mathtt{s}_2(\mu \oplus m_1 \oplus \mathtt{s}_1 m_0) \oplus 1$

Cas $\boxed{e = (\mathtt{new}\ e)}$
Soit $(t,m,\mathtt{s},\mathtt{c}) = \mathtt{R}(\mathrm{T},(\mathtt{new}\ e))$. De l'algorithme, on déduit que $t = \mathtt{ref}\, t_0$, $m = m_0 \oplus 1$ et $(t_0, m_0, \mathtt{s}, \mathtt{c}) = \mathtt{R}(\mathrm{T}, e)$. Par induction sur e on a, pour tout modèle $\mathcal{M}_0$ et pour toute réalisation $\mathcal{S}_0$,

$$\mathcal{M}_0 \models \mathtt{c} \ \Rightarrow\ \mathcal{M}_0\mathcal{S}_0\mathtt{s}\mathrm{T} \vdash e : \mathcal{M}_0\mathcal{S}_0 t_0 \,\$\, \mathcal{M}_0 m_0$$

Soit un modèle $\mathcal{M}$ tel que $\mathcal{M} \models \mathtt{c}$. On a, pour toute réalisation $\mathcal{S}$, $\mathcal{M}\mathcal{S}\mathtt{s}\mathrm{T} \vdash e : \mathcal{M}\mathcal{S}t_0 \,\$\, \mathcal{M}m_0$. Par la règle S.New, nous savons alors que $\mathcal{M}\mathcal{S}\mathtt{s}\mathrm{T} \vdash (\mathtt{new}\ e) : \mathtt{ref}\, \mathcal{M}\mathcal{S}t_0 \,\$\, \mathcal{M}m_0 \oplus 1$, ce dont on déduit $\mathcal{M}\mathcal{S}\mathtt{s}\mathrm{T} \vdash (\mathtt{new}\ e) : \mathcal{M}\mathcal{S}\mathtt{ref}\, t_0 \,\$\, \mathcal{M}(m_0 \oplus 1)$.

Cas $\boxed{e = (\mathtt{get}\ e)}$
Soit $(t,m,\mathtt{s},\mathtt{c}) = \mathtt{R}(\mathrm{T},(\mathtt{get}\ e))$. De l'algorithme, on déduit que $t \sim \mathtt{s}'\tau$, $m \sim \mathtt{s}' m_0 \oplus 1$, $\mathtt{s} = \mathtt{s}'\mathtt{s}_0$, $\mathtt{c} = \mathtt{s}'\mathtt{c}_0$, $\mathtt{s}' = \mathtt{U}(t_0, \mathtt{ref}\,\tau)$ et $(t_0, m_0, \mathtt{s}_0, \mathtt{c}_0) = \mathtt{R}(\mathrm{T}, e)$. Par induction sur e on sait que pour tout modèle $\mathcal{M}_0$ et pour toute réalisation $\mathcal{S}_0$, l'implication suivante est vérifiée.

$$\mathcal{M}_0 \models \mathtt{c}_0 \ \Rightarrow\ \mathcal{M}_0\mathcal{S}_0\mathtt{s}_0\mathrm{T} \vdash e : \mathcal{M}_0\mathcal{S}_0 t_0 \,\$\, \mathcal{M}_0 m_0$$

Soit $\mathcal{M}$ un modèle tel que $\mathcal{M} \models \mathtt{c}_0$, c.à.d. $\mathcal{M} \models \mathtt{s}'\mathtt{c}$. On a donc par induction et par le lemme 3.34, $\mathcal{M}\mathcal{S}\mathtt{s}'\mathtt{s}_0\mathrm{T} \vdash e : \mathcal{M}\mathcal{S}\mathtt{s}' t_0 \,\$\, \mathcal{M}\mathtt{s}' m_0$. L'algorithme d'unification étant cohérent, $\mathtt{s}' t_0 \sim \mathtt{s}'\mathtt{ref}\,\tau$. On peut alors appliquer la règle S.Get pour trouver $\mathcal{M}\mathcal{S}\mathtt{s}\mathrm{T} \vdash (\mathtt{get}\ e) : \mathcal{M}\mathcal{S}t \,\$\, \mathcal{M}m$.

Cas $\boxed{e = (\mathtt{set}\ e_0\ e_1)}$
Soit $(t,m,\mathtt{s},\mathtt{c}) = \mathtt{R}(\mathrm{T},(\mathtt{set}\ e_0\ e_1))$. De l'algorithme, on déduit que

$$\begin{cases} t \sim \mathtt{base} \\ m \sim \mathtt{s}' m_0 \oplus \mathtt{s}'\mathtt{s}_0 m_1 \oplus 1 \\ \mathtt{s} = \mathtt{s}'\mathtt{s}_0\mathtt{s}_1 \\ \mathtt{c} = \mathtt{s}'(\mathtt{c}_0 \cup \mathtt{s}_0\mathtt{c}_1) \\ (t_0, m_0, \mathtt{s}_0, \mathtt{c}_0) = \mathtt{R}(\mathtt{s}_1\mathrm{T}, e_0) \\ (t_1, m_1, \mathtt{s}_1, \mathtt{c}_1) = \mathtt{R}(\mathrm{T}, e_1) \end{cases}$$

Par induction sur e_1 et e_0, on obtient

$$\begin{cases} \mathcal{M}_1 \models \mathtt{c}_1 & \Rightarrow \quad \mathcal{M}_1\mathcal{S}_1\mathtt{s}_1\mathrm{T} \vdash e_1 : \mathcal{M}_1\mathcal{S}_1 t_1 \,\$\, \mathcal{M}_1 m_1 \\ \mathcal{M}_0 \models \mathtt{c}_0 & \Rightarrow \quad \mathcal{M}_0\mathcal{S}_0\mathtt{s}_0\mathtt{s}_1\mathrm{T} \vdash e_0 : \mathcal{M}_0\mathcal{S}_0 t_0 \,\$\, \mathcal{M}_0 m_0 \end{cases}$$

Soit un modèle $\mathcal{M}$ tel que $\mathcal{M} \models \mathtt{c}$, c.a.d. $\mathcal{M} \models \mathtt{s}'(\mathtt{c}_0 \cup \mathtt{s}_0\,\mathtt{c}_1)$. Par inductions on en déduit les propositions suivantes :

$$\begin{cases} \mathcal{MS}\mathtt{s}'\mathtt{s}_0\mathtt{s}_1\mathrm{T} \vdash e_0 : \mathcal{MS}\mathtt{s}'t_0 \,\$\, \mathcal{M}\mathtt{s}'m_0 \\ \mathcal{MS}\mathtt{s}'\mathtt{s}_0\mathtt{s}_1\mathrm{T} \vdash e_1 : \mathcal{MS}\mathtt{s}'\mathtt{s}_0 t_1 \,\$\, \mathcal{M}\mathtt{s}'\mathtt{s}_0 m_1 \end{cases}$$

Comme l'unification est cohérente, $\mathtt{s}'\mathtt{s}_0 t_1 \sim \mathtt{s}'\mathtt{ref}\, t_0$, on peut appliquer la règle S.Set et obtenir $\mathcal{MS}\mathtt{s}'\mathtt{s}_0\mathtt{s}_1\mathrm{T} \vdash (\mathtt{set}\ e_1\ e_0) : \mathtt{base} \,\$\, \mathcal{M}\mathtt{s}'(m_0 \oplus \mathtt{s}_0\, m_1 \oplus 1)$ □

Théorème 3.36 (Complétude) *Si $\mathcal{MS}\mathrm{T} \vdash e : t \,\$\, m$ alors il existe un n-uplet $(t', m', \mathtt{s}', \mathtt{c}')$, un modèle $\mathcal{M}'$ et une substitution $\mathcal{S}'$ tels que :*

$$\begin{cases} (t', m', \mathtt{s}', \mathtt{c}') = \mathtt{R}(\mathrm{T}, e) \\ \mathcal{M}'\mathcal{S}'\mathtt{s}'\mathrm{T} = \mathcal{MS}\mathrm{T} \quad \textit{sur} \quad \mathrm{FV}(e) \\ \mathcal{M}' \models \mathtt{c}' \\ t \sim \mathcal{M}'\mathcal{S}'t' \\ m \sim \mathcal{M}'m' \end{cases}$$

Preuve Par analyse de cas et induction sur la structure des expressions.

Cas $\boxed{e = i}$
Supposons que $\mathcal{MS}\mathrm{T} \vdash i : t \,\$\, m$. De la règle S.Env, on déduit que $m \sim 1$ et $[i : t] \subseteq \mathcal{MS}\mathrm{T}$. Il existe donc un type t' tel que $\mathcal{MS}t' \sim t$ et $[i : t'] \subseteq \mathrm{T}$. L'algorithme retourne en résultat $\mathtt{R}(\mathrm{T}, i) = (t', \mathtt{1}, [], \emptyset)$. Il suffit de prendre $\mathcal{M}'$ et $\mathcal{S}'$ respectivement égaux à $\mathcal{M}$ et $\mathcal{S}$ pour vérifier le théorème.

Cas $\boxed{e = (\mathtt{lambda}\ (i)\ e)}$
Supposons que $\mathcal{MS}\mathrm{T} \vdash (\mathtt{lambda}\ (i)\ e) : t \,\$\, m$. De la règle S.Lambda, on déduit que $m \sim 1$, $t \sim t_1 \xrightarrow{m_l} t_0$ et $(\mathcal{MS}\mathrm{T})[i : t_1] \vdash e : t_0 \,\$\, m_l$. Soient une variable d'unification τ_1, un modèle $\mathcal{M}_1$ et une réalisation $\mathcal{S}_1$ tels que $\tau_1 \notin \mathrm{FV}(\mathrm{T})$, $\mathcal{M}_1\mathcal{S}_1\tau_1 \sim t_1$ et $\forall \tau \neq \tau_1, \mathcal{M}_1\mathcal{S}_1\tau \sim \mathcal{MS}\tau$. On a alors $\mathcal{M}_1\mathcal{S}_1(\mathrm{T}[i{:}\tau_1]) \vdash e : t_0 \,\$\, m_l$. Par induction, il existe $(t_0', m_l', \mathtt{s}_0', \mathtt{c}_0')$, $\mathcal{M}_0'$ et $\mathcal{S}_0'$ tels que

$$\begin{cases} (t_0', m_l', \mathtt{s}_0', \mathtt{c}_0') = \mathtt{R}(\mathrm{T}[i : \tau_1], e) \\ \mathcal{M}_0'\mathcal{S}_0'\mathtt{s}_0'(\mathrm{T}[i : \tau_1]) = \mathcal{M}_1\mathcal{S}_1(\mathrm{T}[i : \tau_1]) \quad \textit{sur} \quad \mathrm{FV}(e) \\ \mathcal{M}_0' \models \mathtt{c}_0' \\ t_0 \sim \mathcal{M}_0'\mathcal{S}_0't_0' \\ m_l \sim \mathcal{M}_0'm_l' \end{cases}$$

L'algorithme retourne en résultat $(\mathtt{s}_0'\tau_1 \xrightarrow{\mu} t_0', \mathtt{1}, \mathtt{s}_0', \mathtt{c}_0' \cup \{\mu \geq m_l'\})$. Il suffit de prendre $\mathcal{M}' = \mathcal{M}_0'[\mu \backslash m_l']$ et $\mathcal{S}' = \mathcal{S}_0'$ pour vérifier le théorème.

Cas $\boxed{e = (\mathtt{rec}\ (f\ i)\ e)}$
Supposons que $\mathcal{MS}\mathrm{T} \vdash (\mathtt{rec}\ (f\ i)\ e) : t \,\$\, m$. De la règle S.Rec, on déduit que $m \sim 1$, $t \sim t_1 \xrightarrow{m_0} t_0$ et $(\mathcal{MS}\mathrm{T})[f : t_1 \xrightarrow{m_0} t_0, i : t_1] \vdash e : t_0 \,\$\, m_0$. Soient les variables d'unification τ_1, τ_0, μ, le modèle $\mathcal{M}_1$ et la réalisation $\mathcal{S}_1$ tels que τ_1, τ_0 et μ n'appartiennent pas à $\mathrm{FV}(\mathrm{T})$, $\mathcal{M}_1\mathcal{S}_1(\tau_1 \xrightarrow{\mu} \tau_0) \sim t_1 \xrightarrow{m_0} t_0$ et $\forall \delta \notin \{\tau_1, \tau_0, \mu\}, \mathcal{M}_1\mathcal{S}_1\delta \sim \mathcal{MS}\delta$. On a alors $\mathcal{M}_1\mathcal{S}_1(\mathrm{T}[f : \tau_1 \xrightarrow{\mu_0} \tau_0, i : \tau_1]) \vdash e : t_0 \,\$\, m_0$. Par induction, il existe $(t_0', m_0', \mathtt{s}_0', \mathtt{c}_0')$, $\mathcal{M}_0'$ et $\mathcal{S}_0'$ tels que :

$$\begin{cases} (t_0', m_0', \mathtt{s}_0', \mathtt{c}_0') = \mathtt{R}(\mathrm{T}[f : \tau_1 \xrightarrow{\mu_0} \tau_0, i : \tau_1], e) \\ \mathcal{M}_0'\mathcal{S}_0'\mathtt{s}_0'(\mathrm{T}[f : \tau_1 \xrightarrow{\mu_0} \tau_0, i : \tau_1]) = \mathcal{M}_1\mathcal{S}_1(\mathrm{T}[f : \tau_1 \xrightarrow{\mu_0} \tau_0, i : \tau_1]) \quad \textit{sur} \quad \mathrm{FV}(e) \\ \mathcal{M}_0' \models \mathtt{c}_0' \\ t_0 \sim \mathcal{M}_0'\mathcal{S}_0't_0' \\ m_l \sim \mathcal{M}_0'm_0' \end{cases}$$

De $t_0 \sim \mathcal{M}_0\,'\mathcal{S}_0\,'t_0\,'$ et $\mathcal{M}_1\mathcal{S}_1\tau_0 \sim t_0$, on sait que $\mathcal{M}_1\mathcal{S}_1\tau_0 \sim \mathcal{M}_0\,'\mathcal{S}_0\,'t_0\,'$. Comme f appartient aux variables libres de e, $\mathcal{M}_0\,'\mathcal{S}_0\,'\mathtt{s}_0\,'(\tau_1 \xrightarrow{\mu_0} \tau_0) \sim \mathcal{M}_1\mathcal{S}_1(\tau_1 \xrightarrow{\mu_0} \tau_0)$ et donc $\mathcal{M}_0\,'\mathcal{S}_0\,'\mathtt{s}_0\,'\tau_0 \sim \mathcal{M}_1\mathcal{S}_1\tau_0$. On obtient alors $\mathcal{M}_0\,'\mathcal{S}_0\,'(\mathtt{s}_0\,'\tau_0) \sim \mathcal{M}_0\,'\mathcal{S}_0\,'t_0\,'$ et l'on sait que le couple $(\mathtt{s}_0\,'\tau_0, t_0\,')$ est unifiable. Soit $\mathtt{s}_0\,''$ le résultat de l'unification. L'algorithme fournit en résultat $(\mathtt{s}\,''(\mathtt{s}_0\,'\tau_1 \xrightarrow{\mathtt{s}_0\,'\mu_0} t_0\,'), 1, \mathtt{s}_0\,'', \mathtt{s}_0\,''(\mathtt{c}_0\,' \cup \{\mathtt{s}_0\,'\mu \geq m_0\,'\}))$. La substitution $\mathcal{M}_0\,'\mathcal{S}_0\,'$ est un unificateur des types $t_0\,'$ et $\mathtt{s}_0\,'\tau_0$, donc, d'après le théorème 3.31, il existe un modèle $\mathcal{M}\,'$ et une réalisation $\mathcal{S}\,'$ tels que $\mathcal{M}\,'\mathcal{S}\,'\mathtt{s}_0\,'' = \mathcal{M}_0\,'\mathcal{S}_0\,'$. Il est aisé de montrer que $\mathcal{M}\,'$ et $\mathcal{S}\,'$ ont les propriétés requises par le théorème.

Cas $\boxed{e = (e_0\ \ e_1)}$
Supposons que $\mathcal{MS}\mathrm{T} \vdash (e_0\ \ e_1) : t\,\$\,m$. De la règle S.Apply, on déduit que $m \sim m_l \oplus m_0 \oplus m_1 \oplus 1$, $t \sim t_0$, $\mathcal{MS}\mathrm{T} \vdash e_1 : t_1\,\$\,m_1$ et $\mathcal{MS}\mathrm{T} \vdash e_0 : t_1 \xrightarrow{m_l} t_0\,\$\,m_0$. Par induction sur e_0, on sait qu'il existe $(t_0\,', m_0\,', \mathtt{s}_0\,', \mathtt{c}_0\,')$, un modèle $\mathcal{M}_0\,'$ et une réalisation $\mathcal{S}_0\,'$ tels que :

$$\begin{cases} (t_0\,', m_0\,', \mathtt{s}_0\,', \mathtt{c}_0\,') = \mathtt{R}(\mathrm{T}, e_0) \\ \mathcal{M}_0\,'\mathcal{S}_0\,'\mathtt{s}_0\,'\mathrm{T} = \mathcal{MS}\mathrm{T} \ \ \textit{sur}\ \ \mathrm{FV}(e_0) \\ \mathcal{M}_0\,' \models \mathtt{c}_0\,' \\ t_1 \xrightarrow{m_l} t_0 \sim \mathcal{M}_0\,'\mathcal{S}_0\,'t_0\,' \\ m_0 \sim \mathcal{M}_0\,'m_0\,' \end{cases}$$

Soient $\mathcal{M}_0\,''$ et $\mathcal{S}_0\,''$ les extensions de $\mathcal{M}_0\,'$ et $\mathcal{S}_0\,'$ telles que $\mathcal{M}_0\,''\mathcal{S}_0\,''\mathrm{T} = \mathcal{MS}\mathrm{T}$ sur l'ensemble $\mathrm{FV}(e_0) \cup \mathrm{FV}(e_1)$. On peut alors induire sur e_1 qu'il existe $(t_1\,', m_1\,', \mathtt{s}_1\,', \mathtt{c}_1\,')$, $\mathcal{M}_1\,'$ et $\mathcal{S}_1\,'$ tels que :

$$\begin{cases} (t_1\,', m_1\,', \mathtt{s}_1\,', \mathtt{c}_1\,') = \mathtt{R}(\mathtt{s}_0\,'\mathrm{T}, e_1) \\ \mathcal{M}_1\,'\mathcal{S}_1\,'\mathtt{s}_1\,'\mathtt{s}_0\,'\mathrm{T} = \mathcal{M}_0\,''\mathcal{S}_0\mathtt{s}_0\,'\,''\mathrm{T} \ \ \textit{sur}\ \ \mathrm{FV}(e_1) \\ \mathcal{M}_1\,' \models \mathtt{c}_1\,' \\ t_1 \sim \mathcal{M}_1\,'\mathcal{S}_1\,'t_1\,' \\ m_1 \sim \mathcal{M}_1\,'m_1\,' \end{cases}$$

Soient $\mathcal{M}_1\,''$ et $\mathcal{S}_1\,''$ les extensions de $\mathcal{M}_1\,'$ et $\mathcal{S}_1\,'$ telles que $\mathcal{M}_1\,''\mathcal{S}_1\,''\mathrm{T} = \mathcal{M}_0\,''\mathcal{S}_0\mathtt{s}_0\,'\,''\mathrm{T}$ sur $\mathrm{FV}(e_0) \cup \mathrm{FV}(e_1)$. On déduit la suite d'équivalence suivante :

$$\begin{aligned} \mathcal{M}_1\,''\mathcal{S}_1\,''\mathtt{s}_1\,'t_0\,' &\sim \mathcal{M}_1\,''\mathcal{S}_1\,''\mathtt{s}_1\,'\mathtt{s}_0\,'t_0\,' \\ &\sim \mathcal{M}_0\,''\mathcal{S}_0\,''\mathtt{s}_0\,'t_0\,' \\ &\sim \mathcal{M}_0\,''\mathcal{S}_0\,''t_0\,' \\ &\sim t_1 \xrightarrow{m_l} t_0 \\ &\sim (\mathcal{M}_1\,''\mathcal{S}_1\,''t_1\,') \xrightarrow{m_l} t_0 \end{aligned}$$

On construit $\mathcal{M}\,''$ et $\mathcal{S}\,''$ par extension de $\mathcal{M}_1\,''$ et de $\mathcal{S}_1\,''$ avec $[\mu \backslash m_l]$ et $[\tau \backslash t_0]$. On obtient alors $\mathcal{M}\,''\mathcal{S}\,''\mathtt{s}_1\,'t_0\,' \sim \mathcal{M}\,''\mathcal{S}\,''(t_1\,' \xrightarrow{\mu} \tau)$. L'unification $\mathtt{U}(\mathtt{s}_1\,'t_0\,', t_1\,' \xrightarrow{\mu} \tau)$ est donc possible et retourne la substitution $\mathtt{s}_2$. L'algorithme construit le n-uplet $(\mathtt{s}_2\tau, \mathtt{s}_2(\mu \oplus m_1\,' \oplus \mathtt{s}_1\,'m_0\,') \oplus 1, \mathtt{s}_2\mathtt{s}_1\,'\mathtt{s}_0\,', \mathtt{s}_2(\mathtt{c}_1\,' \cup \mathtt{s}_1\,'\mathtt{c}_0\,'))$. La substitution $\mathcal{M}\,''\mathcal{S}\,''$ est un unificateur de $\mathtt{s}_1\,'t_0\,'$ et $t_1\,' \xrightarrow{\mu} \tau$, donc il existe $\mathcal{M}\,'$ et $\mathcal{S}\,'$ tels que $\mathcal{M}\,'\mathcal{S}\,'\mathtt{s}_2 = \mathcal{M}\,''\mathcal{S}\,''$. Le modèle $\mathcal{M}\,'$ et la réalisation $\mathcal{S}\,'$ ont les propriétés requises pour la vérification du théorème.

Cas $\boxed{e = (\mathtt{new}\ e)}$
Supposons que $\mathcal{MS}\mathrm{T} \vdash (\mathtt{new}\ e) : t\,\$\,m$. De la règle S.New, on déduit que $m \sim m_0 \oplus 1$,

$t \sim \texttt{ref}\ t_0$ et $\mathcal{MS}\mathrm{T} \vdash e_0 : t_0 \,\$\, m_0$. Par induction, on sait qu'il existe $(t_0\,', m_0\,', \mathtt{s}_0\,', \mathtt{c}_0\,')$, $\mathcal{M}_0\,'$ et $\mathcal{S}_0\,'$ tels que :

$$\begin{cases} (t_0\,', m_0\,', \mathtt{s}_0\,', \mathtt{c}_0\,') = \mathtt{R}(\mathrm{T}, e) \\ \mathcal{M}_0\,'\mathcal{S}_0\,'\mathtt{s}_0\,'\mathrm{T} = \mathcal{MS}\mathrm{T} \ \ sur\ \ \mathrm{FV}(e) \\ \mathcal{M}_0\,' \models \mathtt{c}_0\,' \\ t_0 \sim \mathcal{M}_0\,'\mathcal{S}_0\,'t_0\,' \\ m_0 \sim \mathcal{M}_0\,'m_0\,' \end{cases}$$

L'algorithme rend en résultat le n-uplet $(\texttt{ref}\, t_0\,', m_0\,' \oplus 1, \mathtt{s}_0\,', \mathtt{c}_0\,')$. Prenons $\mathcal{M}\,'$ et $\mathcal{S}\,'$ respectivement égaux à $\mathcal{M}_0\,'$ et $\mathcal{S}_0\,'$. Le théorème est alors trivialement vérifié.

Cas $\boxed{e = (\texttt{get}\ e)}$
Supposons que $\mathcal{MS}\mathrm{T} \vdash (\texttt{get}\ e) : t \,\$\, m$. De la règle S.Get, on déduit que $m \sim m_0 \oplus 1$, $\texttt{ref}\, t \sim t_0$ et $\mathcal{MS}\mathrm{T} \vdash e : t_0 \,\$\, m_0$. Par induction sur e on déduit qu'il existe un n-uplet $(t_0\,', m_0\,', \mathtt{s}_0\,', \mathtt{c}_0\,')$, un modèle $\mathcal{M}_0\,'$ et une réalisation $\mathcal{S}_0\,'$ tels que :

$$\begin{cases} (t_0\,', m_0\,', \mathtt{s}_0\,', \mathtt{c}_0\,') = \mathtt{R}(\mathrm{T}, e) \\ \mathcal{M}_0\,'\mathcal{S}_0\,'\mathtt{s}_0\,'\mathrm{T} = \mathcal{MS}\mathrm{T} \ \ sur\ \ \mathrm{FV}(e) \\ \mathcal{M}_0\,' \models \mathtt{c}_0\,' \\ \texttt{ref}\, t \sim t_0 \sim \mathcal{M}_0\,'\mathcal{S}_0\,'t_0\,' \\ m_0 \sim \mathcal{M}_0\,'m_0\,' \end{cases}$$

Soient $\mathcal{M}\,'$ et $\mathcal{S}\,'$ respectivement égaux à $\mathcal{M}_0\,'$ et $\mathcal{S}_0\,'$étendue avec $[\tau \backslash t]$. On a l'équivalence $\mathcal{M}\,'\mathcal{S}\,'\texttt{ref}\,\tau \sim \mathcal{M}\,'\mathcal{S}\,'t_0\,'$, donc l'unification $\mathtt{U}(t_0\,', \texttt{ref}\,\tau)$ réussit et retourne une substitution, $\mathtt{s}_1$. Comme résultat, l'algorithme construit le quadruplet $(\mathtt{s}_1\tau, \mathtt{s}_1 m_0\,' \oplus 1, \mathtt{s}_1\mathtt{s}_0\,', \mathtt{s}_1\mathtt{c}_0\,')$. Comme $\mathtt{s}_1$ est l'unificateur le plus général (théorème 3.31), il existe un modèle $\mathcal{M}_0$ et une réalisation $\mathcal{S}_0$ tels que $\mathcal{M}\,'\mathcal{S}\,' \sim \mathcal{M}_0\mathcal{S}_0\mathtt{s}_1$. On vérifie aisément le théorème avec $\mathcal{M}_0$ et $\mathcal{S}_0$.

Cas $\boxed{e = (\texttt{set}\ e_1\ e_0)}$
Supposons que $\mathcal{MS}\mathrm{T} \vdash (\texttt{set}\ e_1\ e_0) : t \,\$\, m$. De la règle S.Set, on déduit que $t \sim \texttt{base}$, $m \sim m_1 \oplus m_0 \oplus 1$, $\mathcal{MS}\mathrm{T} \vdash e_1 : \texttt{ref}\, t_0 \,\$\, m_1$ et $\mathcal{MS}\mathrm{T} \vdash e_0 : t_0 \,\$\, m_0$. Par induction sur e_1 on sait qu'il existe $(t_1\,', m_1\,', \mathtt{s}_1\,', \mathtt{c}_1\,')$, $\mathcal{M}_1\,'$ et $\mathcal{S}_1\,'$ tels que :

$$\begin{cases} (t_1\,', m_1\,', \mathtt{s}_1\,', \mathtt{c}_1\,') = \mathtt{R}(\mathrm{T}, e_1) \\ \mathcal{M}_1\,'\mathcal{S}_1\,'\mathtt{s}_1\,'\mathrm{T} = \mathcal{MS}\mathrm{T} \ \ sur\ \ \mathrm{FV}(e_1) \\ \mathcal{M}_1\,' \models \mathtt{c}_1\,' \\ t_1 \sim \mathcal{M}_1\,'\mathcal{S}_1\,'t_1\,' \sim \texttt{ref}\, t_0 \\ m_1 \sim \mathcal{M}_1\,'m_1\,' \end{cases}$$

Soient $\mathcal{M}_1$ et $\mathcal{S}_1$ les extensions de $\mathcal{M}_1\,'$ et $\mathcal{S}_1\,'$ à l'ensemble $\mathrm{FV}(e_1) \cup \mathrm{FV}(e_0)$. Par induction sur e_0 il existe un n-uplet $(t_0\,', m_0\,', \mathtt{s}_0\,', \mathtt{c}_0\,')$, un modèle $\mathcal{M}_0\,'$ et une réalisation $\mathcal{S}_0\,'$ tels que :

$$\begin{cases} (t_0\,', m_0\,', \mathtt{s}_0\,', \mathtt{c}_0\,') = \mathtt{R}(\mathtt{s}_1\,'\mathrm{T}, e_0) \\ \mathcal{M}_0\,'\mathcal{S}_0\,'\mathtt{s}_0\,'\mathtt{s}_1\,'\mathrm{T} = \mathcal{M}_1\mathcal{S}_1\mathtt{s}_1\,'\mathrm{T} \ \ sur\ \ \mathrm{FV}(e_0) \\ \mathcal{M}_0\,' \models \mathtt{c}_0\,' \\ t_0 \sim \mathcal{M}_0\,'\mathcal{S}_0\,'t_0\,' \\ m_0 \sim \mathcal{M}_0\,'m_0\,' \end{cases}$$

Soient $\mathcal{M}_0$ et $\mathcal{S}_0$ les extensions de $\mathcal{M}_0\,'$ et $\mathcal{S}_0\,'$ à $\mathrm{FV}((\texttt{set}\ e_1\ e_0))$. On déduit la suite d'équivalences suivante :

$$\mathcal{M}_0\mathcal{S}_0(\texttt{ref}\, t_0\,') \quad \sim \quad \texttt{ref}\, t_0$$

$$
\begin{aligned}
&\sim \mathcal{M}_1\mathcal{S}_1 t_1\,' \\
&\sim \mathcal{M}_1\mathcal{S}_1 \mathtt{s}_1\,' t_1\,' \\
&\sim \mathcal{M}_0\mathcal{S}_0 \mathtt{s}_0\,' \mathtt{s}_1\,' t_1\,' \\
&\sim \mathcal{M}_0\mathcal{S}_0 (\mathtt{s}_0 t_1\,')
\end{aligned}
$$

L'unification $\mathtt{U}(\mathtt{s}_0\,' t_1\,', \mathtt{ref}\, t_0\,')$ réussit et retourne la substitution $\mathtt{s}_2$. L'algorithme construit comme résultat le n-uplet $(\mathtt{base}, \mathtt{s}_2(m_0\,' \oplus \mathtt{s}_0\,' m_1\,') \oplus 1, \mathtt{s}_2\mathtt{s}_0\,' \mathtt{s}_1\,', \mathtt{s}_2(\mathtt{c}_0\,' \cup \mathtt{s}_0\,' \mathtt{c}_1\,'))$. La substitution $\mathcal{M}_0\mathcal{S}_0$ est un unificateur des types $\mathtt{s}_0\,' t_1\,'$ et $\mathtt{ref}\, t_0\,'$. Comme l'unification est complète, il existe $\mathcal{M}\,'$ et $\mathcal{S}\,'$ tels que $\mathcal{M}\,'\mathcal{S}\,'\mathtt{s}_2 = \mathcal{M}_0\mathcal{S}_0$. Le théorème est alors vérifié pour ce modèle et cette réalisation. □

7 Résolution des contraintes

Dans cette section, nous présentons l'algorithme de résolution des contraintes générées par l'algorithme de reconstruction. Nous démontrons ensuite que l'algorithme proposé calcule la solution recherchée.

7.1 Algorithme

L'algorithme est composé de trois étapes dont nous ne spécifions pas les détails, étant donné leur simplicité. La première phase normalise l'ensemble de contraintes. La deuxième, constituant la résolution proprement dite, est une itération de point fixe. La dernière extrait les résultats désirés.

Données Un ensemble de contraintes. Chaque contrainte est un couple constitué d'une variable d'unification de temps et d'une expression de temps. Les expressions de temps sont des sommes de variables et de constantes de temps.

Algorithme

1. **Normalisation** `Si` deux contraintes possèdent la même variable d'unification à gauche, `alors` les remplacer par une seule maximisant les deux termes droits.

$$\left.\begin{array}{l}\mu \geq M_1(\mu_1, \ldots, \mu_n) \\ \mu \geq M_2(\mu_1, \ldots, \mu_n)\end{array}\right\} \longrightarrow \mu \geq M_1(\mu_1, \ldots, \mu_n) \sqcup M_2(\mu_1, \ldots, \mu_n)$$

2. **Point fixe** `Itérer jusqu'à` obtenir un point fixe en assignant au départ toutes les variables à la valeur `long`.

3. **Extraction** Extraire les valeurs des variables d'unification de l'ensemble de contraintes obtenu.

Résultats Ensemble d'associations entre variables d'unification et temps fondamentaux.

Algorithme de Résolution

L'opérateur $\sqcup$ est un opérateur de maximisation des temps. Il correspond à la maximisation sur *IN* étendue de façon directe à la valeur `long`. Il correspond à la plus petite borne

supérieure[3] sur le treillis des temps.

La solution utilisée dans cet algorithme est une itération de point fixe maximal. La solution désirée au système de contraintes est la plus petite. Nous allons démontrer maintenant qu'il y a bien correspondance entre les deux.

7.2 Spécifications et définitions

Nous allons maintenant montrer que l'utilisation du point fixe maximal (partant de `long`) est valide et effective. Pour cela, nous commençons par spécifier de façon plus algébrique le système de contraintes à résoudre. Celui-ci est de la forme :

$$\left\{\begin{array}{rcl} \mu_1 & \geq & \mathrm{M}_1(\mu_1, \mu_2, \cdots, \mu_n) \\ \mu_2 & \geq & \mathrm{M}_2(\mu_1, \mu_2, \cdots, \mu_n) \\ & \vdots & \\ \mu_n & \geq & \mathrm{M}_n(\mu_1, \mu_2, \cdots, \mu_n) \end{array}\right.$$

où chaque opérateur de temps M_i est de la forme :

$$\left\{\begin{array}{rccl} \mathrm{M}_i : & \mathrm{Time}^n & \longrightarrow & \mathrm{Time} \\ & \langle \mu_i \rangle_{i=1}^n & \mapsto & \bigsqcup_{j \in I_i} (c_j \oplus \bigoplus_{k=1}^{n} (a_{j,k} . \mu_k)) \end{array}\right.$$

où c_j est une constante de temps. On définit ensuite $\mathrm{U} = \langle \mu_i \rangle_{i=1}^n$ et

$$\left\{\begin{array}{rccl} \mathrm{M} : & \mathrm{Time}^n & \longrightarrow & \mathrm{Time}^n \\ & \mathrm{U} & \mapsto & \langle \mathrm{M}_j(\mathrm{U}) \rangle_{j=1}^n \end{array}\right.$$

La solution recherchée pour l'ensemble de contraintes est la plus petite solution de l'inéquation $\mathrm{U} \geq \mathrm{M}(\mathrm{U})$. Cette solution est le $lpfp(\mathrm{M})$, c.à.d. le plus petit pré-point fixe de l'opérateur M [A90].

Nous définissons maintenant les puissances croissantes et décroissantes de l'opérateur M. Celles-ci, reprises de [A90], permettent de manipuler, de manière algébrique, les itérations de l'opérateur M.

Définition 3.37 (puissances)

$$\begin{array}{rclcrcl} \mathrm{M}{\uparrow_0}(m) & = & m & \qquad & \mathrm{M}{\downarrow_0}(m) & = & m \\ \mathrm{M}{\uparrow_{n+1}}(m) & = & \mathrm{M}(\mathrm{M}{\uparrow_n}(m)) & & \mathrm{M}{\downarrow_{n+1}}(m) & = & \mathrm{M}(\mathrm{M}{\downarrow_n}(m)) \\ \mathrm{M}{\uparrow_\omega}(m) & = & \bigsqcup_{n \leq \omega} \mathrm{M}{\uparrow_n}(m) & & \mathrm{M}{\downarrow_\omega}(m) & = & \bigsqcup_{n \leq \omega} \mathrm{M}{\downarrow_n}(m) \\ \mathrm{M}{\uparrow_\omega} & = & \mathrm{M}{\uparrow_\omega}(\mathbb{1}) & & \mathrm{M}{\downarrow_\omega} & = & \mathrm{M}{\downarrow_\omega}(\mathbb{L}\mathrm{ong}) \\ \mathbb{1} & = & \langle 1 \rangle_{i=1}^n & & \mathbb{L}\mathrm{ong} & = & \langle \texttt{long} \rangle_{i=1}^n \end{array}$$

7.3 Preuve

Le principe de la preuve est simple. La solution désirée au système de contraintes correspond au $lpfp(\mathrm{M})$. L'algorithme, quand à lui, calcule le $\mathrm{M}{\downarrow_\omega}$. Nous devons démontrer que ces deux objets sont égaux.

Théorème 3.38 (Monotonicité) *L'opérateur* M *est monotone.*

[3] On dit "lub" en Anglais.

Preuve Soient $\langle m_i \rangle_{i=1}^n$ et $\langle m'_i \rangle_{i=1}^n$ deux vecteurs de temps appartenant à Time^n tels que $\langle m_i \rangle_{i=1}^n \leq \langle m'_i \rangle_{i=1}^n$.

$$\begin{aligned}
\langle m_i \rangle_{i=1}^n \leq \langle m'_i \rangle_{i=1}^n &\Rightarrow \forall j \ m_j \leq m'_j \\
&\Rightarrow \forall j \ \forall i \ a_{i,j}.m_j \leq a_{i,j}.m'_j \\
&\Rightarrow \forall i \ \bigoplus_{j=1}^{n} a_{i,j}.m_j \leq \bigoplus_{j=1}^{n} a_{i,j}.m'_j \\
&\Rightarrow \forall i \ c_i \oplus \bigoplus_{j=1}^{n} (a_{i,j}.m_j) \leq c_i \oplus \bigoplus_{j=1}^{n} (a_{i,j}.m'_j) \\
&\Rightarrow \bigsqcup_{i \in I} (c_i \oplus \bigoplus_{j=1}^{n} (a_{i,j}.m_j)) \leq \bigsqcup_{i \in I} (c_i \oplus \bigoplus_{j=1}^{n} (a_{i,j}.m'_j)) \\
&\Rightarrow \mathrm{M}(\langle m_i \rangle_{i=1}^n) \leq \mathrm{M}(\langle m'_i \rangle_{i=1}^n)
\end{aligned}$$

□

Corollaire 3.39

$$\begin{aligned}
lpfp(\mathrm{M}) &= lfp(\mathrm{M}) \\
gpfp(\mathrm{M}) &= gfp(\mathrm{M})
\end{aligned}$$

Preuve L'opérateur M est monotone. □

Théorème 3.40 (Continuité) *L'opérateur* M *est (ascendant) continu.*

Preuve Il suffit de montrer que chacune des n restrictions de M au domaine $\text{Time} \longrightarrow \text{Time}^n$ est continue. On définit les restrictions comme suit :

$$\left\{ \begin{array}{lrcl}
\mathrm{M}|_j : & \text{Time} & \longrightarrow & \text{Time}^n \\
& \mu_j & \mapsto & \langle \mathrm{M}|_j^i(\mu_j) \rangle_{i=1}^n \\
\\
\mathrm{M}|_j^i : & \text{Time} & \longrightarrow & \text{Time} \\
& \mu_j & \mapsto & \bigsqcup_{k \in I_i} (c'_k \oplus a_{k,j}.\mu_j) \\
& & & avec \ c'_k = c_k \oplus \bigoplus_{l \neq j} a_{k,l}.\mu_l
\end{array} \right.$$

Soit $m_0 \leq m_1 \leq m_2 \leq \cdots$ une chaîne croissante de valeurs dans Time. Deux cas se présentent :

1. La chaîne est stationnaire à partir d'un certain rang p, $\exists p$ t.q. $\forall i \geq p, m_i = m_p$. On a alors $\bigsqcup_{i=0}^{\infty} m_i = m_p$. Ceci implique que $\mathrm{M}|_j(\bigsqcup_{i=0}^{\infty} m_i) = \mathrm{M}|_j(m_p)$. Comme m_p appartient à la chaîne, on a l'inclusion $\mathrm{M}|_j(m_p) \leq \bigsqcup_{i=0}^{\infty} \mathrm{M}|_j(m_i)$ et donc, l'opérateur $\mathrm{M}|_j$ est finitaire. Comme il est monotone, il est continu.

2. La chaîne est infiniment croissante. Dans ce cas, on a $\bigsqcup_{i=0}^{\infty} m_i = \texttt{long}$[4]. Ceci implique que $\mathrm{M}|_j(\bigsqcup_{i=0}^{\infty} m_i) = \mathrm{M}|_j(\texttt{long})$. Soit L, un ensemble d'indices (notés l) sous-ensemble de $\{1, \cdots, n\}$ tel qu'il existe $k \in I_l$ et que $a_{k,j} \neq 0$. On sait alors que, pour tout l de

[4] On note $\bigsqcup_{i=0}^{\infty} m_i$ la plus petite borne supérieure de la chaîne des m_i.

L, la chaîne $(\mathrm{M}|_j^l(m_i))_i$ croît infiniment, donc $\bigsqcup_{i=0}^{\infty} \mathrm{M}|_j^l(m_i) = \texttt{long}$. Par contre, pour tout $l \notin L$, la chaîne est constante en $\bigsqcup_{k \in I_l} m'_l$. L'union de toutes les images de la chaîne par $\mathrm{M}|_j$ est donc un vecteur composé d'éléments égaux à `long` quand l'indice de l'élément est dans L et égaux aux maximum des termes constants quand l'indice de l'élément n'est pas dans L. Cette valeur est exactement l'image de `long`.

□

Corollaire 3.41

$$lfp(Mc) = \mathrm{M}\uparrow_\omega$$

Preuve L'opérateur M est (ascendant) continu. □

Théorème 3.42 M *est un opérateur descendant continu.*

Preuve Soit $m_0 \geq m_1 \geq m_2 \geq \cdots$ une chaîne décroissante de valeurs dans Time. Cette chaîne est constante à partir d'un certain rang. En effet, soit elle est constante sur `long`, soit elle passe par une valeur strictement inférieure à `long` et termine stationnaire au pire sur 1. Il existe donc p tel que $\forall i \geq p, m_i = m_p$. On en déduit que $\bigcap_{i=0}^{\infty} m_i = m_p$[5] et par conséquent que $\mathrm{M}|_j(\bigcap_{i=0}^{\infty} m_i) = \mathrm{M}|_j(m_p)$. Comme cette dernière est dans l'image de la chaîne, elle est plus grande que $\bigcap_{i=0}^{\infty} \mathrm{M}|_j(m_i)$. L'opérateur est donc descendant finitaire et, comme il est monotone, il est aussi descendant continu. □

Corollaire 3.43

$$gfp(Mc) = \mathrm{M}\downarrow_\omega$$

Preuve L'opérateur M est descendant continu. □

Propriété 3.44 *Il existe un entier p tel que* $\mathrm{M}\downarrow_\omega$ *est égal à* $\mathrm{M}\downarrow_n(\mathbb{L}\mathrm{ong})$.

Preuve Il n'existe pas de chaîne ω-infinie décroissante non stationnaire sur Time. □

Théorème 3.45

$$\mathrm{M}\uparrow_\omega = \mathrm{M}\downarrow_\omega$$

Preuve Il suffit de prouver que le point fixe est unique. Pour cela, on suppose qu'il existe deux valeurs de Time^n, V_1 et V_2, telles que

$$\begin{cases} V_1 & = & \mathrm{M}(V_1) \\ V_2 & = & \mathrm{M}(V_2) \\ V_1 & \neq & V_2 \end{cases}$$

Des deux premières équations, on déduit que pour tout l de $\{1, \cdots, n\}$, on a

$$\begin{cases} \bigsqcup_{i \in I_l}(m_i \oplus \bigoplus_{j=1}^{n} a_{i,j}.m_{1,j}) & = & m_{1,k} \\ \bigsqcup_{i \in I_l}(m_i \oplus \bigoplus_{j=1}^{n} a_{i,j}.m_{2,j}) & = & m_{2,k} \end{cases}$$

[5] On note $\bigcap_{i=0}^{\infty} m_i$ la plus grande borne inférieure de la chaîne des m_i.

On définit un ensemble L_i ($i \in \{1, \cdots, n\}$) comme l'ensemble des indice j tels que tous les m_j sont définis en fonction d'eux-mêmes à travers un cycle de longueur i.

$$\left\{\begin{array}{rcl} L_1 & = & \{l | i \in I_l \wedge a_{i,l} \neq 0\} \\ L_2 & = & \{l | i \in I_l \wedge j \in \{1, \cdots, n\} \wedge i' \in I_j \wedge a_{i,j} \neq 0 \wedge a_{i',l} \neq 0\} \\ & \vdots & \\ L_q & = & \{j_q | \forall p \in \{1, \cdots, q\}, j_p \in \{1, \cdots, n\} \wedge i_{p+1} \in I_{j_p} \wedge i_1 \in I_{j_q} \wedge a_{i_p, j_p} \neq 0\} \\ & \vdots & \\ L_n & = & \{j_n | \forall p \in \{1, \cdots, n\}, j_p \in \{1, \cdots, n\} \wedge i_{p+1} \in I_{j_p} \wedge i_1 \in I_{j_n} \wedge a_{i_p, j_p} \neq 0\} \\ & & \\ L & = & \bigsqcup_{i=1}^{n} L_i \end{array}\right.$$

Nous savons que pour tout $l \in L$ on a $m_{1,l} = m_{2,l} =$ `long`. L'ensemble des variables d'indice non appartenant à L peut être totalement ordonné par les dépendances dues aux définitions exprimées par les équations. Chaque variable est donc uniquement définie. Il s'en suit que $V_1 = V_2$. □

L'effectivité de l'algorithme proposé est confirmée par la propriété 3.44. Comme il n'existe pas de chaîne ω-infinie décroissante et non stationnaire, la chaîne construite par l'algorithme est stable à partir d'un certain nombre d'itérations et donc l'algorithme termine. Il faut remarquer, qu'il n'en serait pas de même si l'on partait du bottom (`1`) du treillis. En pratique, l'algorithme est au pire de complexité linéaire sur le nombre de variables d'unification.

8 Conclusion

Nous avons présenté un système de reconstruction des types et des temps pour un langage implicitement typé. Un temps est un entier majorant statiquement le temps d'exécution ou `long` quand le programme est récursif. Cette simplicité de description est à modérer à la vue des points suivants :

- Fournir des informations très complexe est inutile si le module de gestion du parallélisme n'est pas à même de les exploiter. Les informations doivent rester simples comme celle utilisées par [G83].
- Trouver un système capable de gérer des informations plus riches (comme linéaire sur la taille de la donnée d'entrée, quadratique ...) est tout de suite plus difficile. En effet, cela nécessite d'être capable de décrire la taille (ou une autre caractéristique) des données sur lesquelles sont exprimées les complexités.
- Notre langage est un langage complet aussi bien d'un point de vue applicatif qu'impératif.

L'autre point critiquable touche à la compilation séparée. Le programmeur ne possède pas le moyen d'exprimer un quelconque polymorphisme. Il ne pourra donc pas abstraire ses modules sur divers types de données (Problème analogue à celui du langage *Pascal*). Le polymorphisme s'introduit aisément en utilisant la construction `let` permettant de définir des variables locales à un bloc. La quantification universelle se fait alors implicite et utilise les schémas de type contraints [TJ91a,TJ91b]. Une technique de reconstruction partielle des types et des temps en présence d'abstractions polymorphes explicites est présentée dans le chapitre suivant.

L'intérêt de notre système se situe dans la puissance d'expression de notre langage. Il n'avait jamais été proposé, à notre connaissance, de système d'analyse automatique capable de fournir ce type de résultat avec un langage complet. On peut aussi constater, que les temps et types étant entièrement reconstruits, ils ne gênent pas le programmeur dans son exercice.

Appendice : Exemples

Afin de mieux comprendre le système de reconstruction des types et des temps, voyons quelques exemples à travers une session de travail. Nous nous trouvons sous l'interpréteur *ITL*. Le prompt "`ITL>`" indique l'entrée d'une expression. Le résultat de l'évaluation est fourni deux lignes plus loin avec les types et temps reconstruits. Dans un but pédagogique, la lisibilité des exemples présentés a été améliorée par rapport à l'état actuel de la maquette.

```
ITL> 3

3 : base $ 1

ITL> true

true : base $ 1
```

Les expressions atomiques s'évaluent en elles-mêmes. Le temps de calcul est `1`. Il correspond à l'accès aux valeurs de base. Il faut noter que les booléens et les entiers ont le même type. Cela peut mener à des erreurs au moment de l'exécution. Notre but, n'est pas de reconstruire les types exacts (ce qui a déjà été fait par Tofte dans [T88,T90]). Par contre, le temps inféré est consistant avec le temps effectivement utilisé lors de l'exécution.

```
ITL> (lambda (x) x)

<subr> : 't ---[1]--> 't $ 1

ITL> +

<subr> : base ---[1]--> (base ---[2]--> base) $ 1

ITL> (+ 3)

<subr> : base ---[2]--> base $ 4

ITL> ((+ 3) 4)

7 : base $ 8
```

L'identité est une abstraction. Le temps d'évaluation d'une abstraction est constant et a été fixé arbitrairement à `1`. Le temps latent de l'identité est aussi `1`. Il correspond à l'évaluation de la variable `x` qui aura lieu lors de l'application. L'addition est une fonction curryfiée. Appliquée à un entier elle rend une fermeture qui appliquée à un deuxième entier rend le résultat de l'addition. On peut constater que l'addition prend un temps arbitrairement fixé à `2`. L'application d'une fonction à une expression nécessite d'évaluer la fonction, d'évaluer l'argument, de mettre en place l'application (par défaut, on fixe le temps

pris par cette phase à `1`) et d'appliquer la fermeture à la valeur argument. On obtient `4`, puis `8` lors des évaluations de l'addition.

```
ITL> (rec (f x) x)

<subr> : 't ---[1]--> 't $ 1

ITL> (rec (fact n) (if ((= 0) n) 1 ((* n) (fact ((- n) 1)))))

<subr> : base ---[long]--> base $ 1
```

L'opérateur `rec` permet de définir des fonction récursives. Naturellement, la première n'est que l'identité et n'est donc pas effectivement récursive. Son temps latent est donc, comme précédemment, de `1`. La seconde définit la fonction factorielle. Le nombre d'itérations n'étant pas borné à priori, le temps latent est `long`.

Sur la maquette que nous avons développée en *Common Lisp*, la fonction la plus proche de `fact` est (nous ne disposons pas du test) :

```
CL> (R T (fix (f x) ((* (f x)) x)))

Type: (o ---[m2]--> o)
Time: 1
Substitution:
   t2 / t2 ;
   t3 / t1 ;
   m2 / m1 ;
   o / t3 ;
   (o ---[1]--> o) / t4 ;
   1 / m3 ;
   o / t2 ;
   o / t5 ;
   1 / m4 ;
Constraints:
   m2 >= 9+m2
```

L'algorithme de reconstruction `R`, sur un environnement de type `T` et sur l'expression correspondant à l'approximation de la fonction `fact`, fournit quatre objets :

- Le type de `fact`. Notez le temps latent `m2`.
- Le temps. Ici : 1.
- La substitution construite sur les variables d'unification des types et des temps.
- L'ensemble de contraintes associé au type.

Nous présentons pour terminer la réduction utilisée dans le langage *FP* comme opérateur d'insertion dans les listes[6]. Munie d'une fonction binaire et d'une liste, elle construit une fermeture qui, appliquée à une valeur d'accumulation, rend le résultat de la réduction :

$$\begin{aligned} (insert\ f\ <x>) &= \lambda a.(f\ x\ a) \\ (insert\ f\ <x, y, z, ...>) &= \lambda a.(f\ x\ ((insert\ f\ <y, z, ...>)\ a)) \end{aligned}$$

[6] Exemple dû à P. Fradet.

Le temps latent de la fonction `insert`, ainsi que celui de la fermeture construite, est égal à `long`. Normalement, `insert` s'écrit :

```
(fix (insert f)
   (lambda (l)
      (if (null l) (lambda (x) x)
          (let ((z ((insert f) (cdr l))))
             (lambda (x) (f (car l) (z x)))))))
```

Comme nous ne disposons pas du test, ni des déclaration locales, la version approchée de `insert` sur notre maquette est :

```
CL> (R T (fix (INSERT F)
            (lambda (L) ((lambda (Z) (lambda (X) ((F (CAR L)) (Z X))))
                         ((INSERT F) (CDR L))))))

Type: (o ---[m3]--> (t9 ---[m5]--> t9)) ---[m8]-->
                    (o ---[m12]--> (t5 ---[m6]--> t9))
Time: 1
Substitution:
  o / t3 ;
  o / t6 ;
  ...
  t9 / t8 ;
  m6 / m4 ;
  m12 / m10 ;
Constraints:
  m6 >= 10+m5+m6+m3
  m11 >= 1
  m12 >= 10+m11+m12+m8
  m8 >= 1
```

Les temps correspondant à l'exécution de l'application de la fonction à une liste et au temps latent du résultat sont respectivement `m12` et `m6`. Il est aisé de vérifier que les contraintes les forcent à `long`.

– – –◊◊◊– ––

Chapitre 4

Vérification des temps

1 Introduction

Dans le chapitre précédent, nous avons présenté un système d'inférence des temps pour les langages possédant les concepts applicatifs et impératifs. Nous présentons ici un équivalent utilisant un système de typage explicite. Le langage proposé étend celui présenté dans [DJG91] à la prise en compte des effets de bords. L'échelle des temps choisie est identique à celle du chapitre précédent : une valeur entière pour majorer statiquement le temps de calcul quand cela est possible et la valeur `long` pour symboliser un temps d'exécution non statiquement borné.

Notre langage est explicitement typé. Il obéit à la discipline de typage avec sorte[1] de MacCracken [M79] : le programmeur fait des déclarations sur la validité des types. Notre langage est polymorphe et permet au programmeur de s'abstraire sur les types des objets manipulés. Le polymorphisme utilisé est universel paramétrique : Les types sur lesquels les fonctions sont abstraites n'entrent que dans la description des types des paramètres formels et n'influent pas sur le code exécuté. Reprenant la philosophie appliquée dans la conception du langage *FX* [GJLS87], les déclarations de type incluent des déclarations de temps d'exécution (d'effets de bord dans *FX*). Ces déclarations sont vérifiées à la compilation en même temps que les types. L'échelle des temps choisie est constituée du domaine des entiers auquel on ajoute l'élément `long` représentant la récursion (et la non terminaison éventuelle).

Pourquoi proposer un système de *vérification* des types et des temps alors que l'on dispose déjà d'un système d'*inférence* (cf. chapitre précédent)? On pourrait penser qu'il est préférable de décharger le programmeur de l'écriture des types des objets. De plus, l'inférence permet d'obtenir le type le plus général et par conséquent d'augmenter les possibilités de réutilisation du code. Il faut cependant constater que le langage précédent est moins expressif, et ce à deux points de vue. Premièrement, l'algorithme de Robinson [R65] (avec test d'occurrence) interdit d'unifier une variable avec une expression la contenant et, par conséquent, de manipuler des descriptions récursives. L'écriture de l'opérateur de récursion `Y` est impossible car nécessitant un type récursif pour la variable auto-appliquée. Pour éliminer ce problème, nous avions introduit la forme spéciale de récursion `rec`. Dans le langage présent, cette forme spéciale n'est plus nécessaire. Deuxièmement, l'introduction du polymorphisme à travers la construction `let` pose des problèmes analogues à ceux des langages à la *SML* en imposant l'utilisation de *schémas de type*[2] contraints (voir [TJ91b]

[1] Dû à l'Anglais, nous utilisons le domaine Kind et la méta-variable k.

[2] *type schemes* en Anglais.

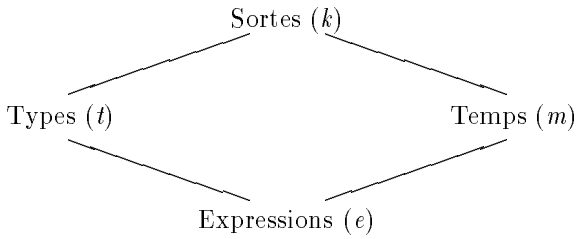
pour un problème identique avec les effets de bord). Le polymorphisme exprimé en est limité, ce qui complique l'introduction de modules pour la compilation séparée. A l'inverse, la quantification explicite du langage *FX* s'est montrée très performante pour l'introduction de modules de première classe [SG89]. Nous verrons, à la fin de ce chapitre, que l'on peut libérer quelque peu le programmeur en reconstruisant une partie des types (types des abstractions) en utilisant un algorithme d'unification ACZ.

Nous présentons (section 2), la syntaxe du langage accompagné de sa sémantique dynamique. Les informations de temps présentes dans le langage sont vérifiées (section 3) en même temps que les types. Comme précédemment, chapitre 3, la consistance de la sémantique statique par rapport au schéma d'évaluation est prouvé (section 4). Le système de typage étant désormais valide, l'algorithme de vérification est introduit et prouvé (section 5). Les expressions du langage n'étant pas pourvues de constructions pour exprimer la récursion, on montre (section 6) comment introduire l'opérateur paradoxal `Y` sans perdre les propriétés du système de typage. Certaines informations, comme le type des fonctions, sont reconstructible (section 7) d'une manière similaire à [JG91]. Nous concluons (section 8) avant de donner quelques exemples en appendice.

2 Définition du langage

Le mini-langage *ETL*[3] contient toutes les primitives nécessaires à l'expression d'un langage de programmation fortement typé combinant les paradigmes des langages applicatifs et impératifs. Le schéma d'évaluation est l'appel par valeur. Le système de typage obéit à la discipline de typage avec *sorte* de MacCracken [M79]. Le polymorphisme est générique (universel paramétrique).

Le langage *ETL* possède deux niveaux hiérarchisés de description. Les temps et les types décrivent statiquement les temps d'exécution et les valeurs associés aux expressions du langage. Les sortes (types des descriptions) permettent une vérification de la validité des descriptions données par le programmeur. Sur le schéma suivant, ne sont symbolisées que les descriptions de type et de temps. Les descriptions restantes, abstraction (`dlambda`) et application d'un constructeur de type à un type, ne figurent pas.



Le langage *ETL* est parenthésé à la *LisP* par simple convention. Sa syntaxe est fournie sous forme d'une grammaire BNF. Il n'y a qu'un seul espace de nom pour les identificateurs. Les méta-variables (symboles non terminaux) sont écrites en *italique* tandis que les mots clés du langage (terminaux) le sont en `télétype`.

Une expression (non terminal e) peut être un identificateur (i), une abstraction, une application, l'abstraction d'une expression par rapport à une description de type ou de temps (`plambda`) ou la projection d'une expression sur une description (`proj`). Les expressions ne comprennent pas d'opérateur de récursion. On montrera, section 6, que notre langage

[3] pour Explicitly Typed Language

est assez puissant pour pouvoir l'introduire. Les primitives nécessaires aux effets de bord ne sont pas présentes. Elles sont introduites à travers des identificateurs spéciaux dans l'environnement initial.

Une description (non terminal d) peut être un temps (m) ou un type (t). Un temps peut être un entier positif non nul, `long` ou la maximisation de deux temps ($\oplus$). Un type peut être un type récursif (`drec`), le type d'une abstraction (`subr`), un type polymorphe (`poly`), le type d'une référence en mémoire (`refof`) ou un identificateur. Une description peut être une abstraction sur un type ou un temps (`dlambda`) ou une application d'une description à une autre. Ces deux dernières permettent la manipulation de constructeurs de type. Le constructeur de type `refof` n'est qu'un cas particulier d'application de descriptions mais nous le conservons dans une but pédagogique.

Une sorte (non terminal k) peut être les identificateurs `type` et `time` ou la sorte associé à une abstraction de description (`dfunc`).

$k \in$	Kind	
$k ::=$	`type` | `time`	
	(`dfunc` (k) k)	Sorte d'abstraction de description
$d \in$	Descr	
$d ::=$	$m \mid t$	
	(`dlambda` (i k) d)	Abstraction de description
	(d d)	Application de descriptions
$m \in$	Time	
$m ::=$	`1` | `2` | `3` | ... | `long`	
	($\oplus$ m m)	Accumulation de temps
	i	
$t \in$	Type	
$t ::=$	(`subr` m (t) t)	Type d'abstraction
	(`refof` t)	Type d'une référence
	(`drec` i t)	Type récursif
	(`poly` (i k) t)	Type Polymorphe
	i	
$e \in$	Expr	
$e ::=$	i	
	(`lambda` (i t) e)	Abstraction
	(e e)	Application
	(`plambda` (i k) e)	Abstraction Polymorphe
	(`proj` e d)	Projection
$i \in$	Id	Identificateurs

Syntaxe du langage

En appendice de ce chapitre, quelques exemples simples de programmes *ETL* donnent une idée de la manipulation des constructeurs de types et du typage des expressions. Pour compléter cette définition d'*ETL*, il ne reste qu'à spécifier la sémantique dynamique de notre langage.

Nous utilisons les quatre domaines sémantiques standards Loc, Val, Env et Store. A ceux-ci, nous ajoutons le domaine Nat^+ symbolisant l'horloge de l'évaluateur. Le domaine BVal contient les valeurs élémentaires comme les entiers et les booléens. Le domaine BClos contient les opérateurs primitifs comme l'addition, la multiplication ou les primitives de manipulation de la mémoire.

$$
\begin{array}{rrll}
l \in & \mathrm{Loc} & & \text{Location} \\
b \in & \mathrm{BVal} & = \mathrm{Bool} + \mathrm{Int} + \mathrm{BClos} + \{\mathsf{unit}\} & \text{Valeurs de base} \\
 & \mathrm{BClos} & = \{\mathsf{new}, \mathsf{get}, \mathsf{set}, +, \ldots\} & \text{Opérateurs primitifs} \\
v \in & \mathrm{Val} & = \mathrm{Loc} + \mathrm{BVal} + \mathrm{Clos} & \text{Valeurs} \\
\langle i, e, \mathrm{E}\rangle \in & \mathrm{Clos} & = (\mathrm{Id} \times \mathrm{Expr} \times \mathrm{Env}) & \text{Fermetures} \\
\mathrm{st} \in & \mathrm{Store} & = \mathrm{Loc} \to \mathrm{Val} & \text{Mémoires} \\
\mathrm{E} \in & \mathrm{Env} & = \mathrm{Id} \to \mathrm{Val} & \text{Environnements} \\
n \in & \mathrm{Nat}^+ & & \text{Naturels}
\end{array}
$$

L'évaluation d'une expression quelconque nécessite un environnement initial (E_0). Ce dernier contient les liaison des identificateurs aux valeurs de base (entiers, booléens et opérateurs primitifs) :

$$
\begin{aligned}
\mathrm{E}_0 = [&\mathsf{new}{\leftarrow}\mathsf{new}, \mathsf{get}{\leftarrow}\mathsf{get}, \mathsf{set}{\leftarrow}\mathsf{set}, +{\leftarrow}+, \ldots \\
&\mathsf{unit}{\leftarrow}\mathsf{unit}, \#\mathsf{t}{\leftarrow}\#\mathsf{t}, \#\mathsf{f}{\leftarrow}\#\mathsf{f}, \ldots \\
&\mathsf{0}{\leftarrow}\mathsf{0}, \mathsf{1}{\leftarrow}\mathsf{1}, \mathsf{2}{\leftarrow}\mathsf{2}, \ldots]
\end{aligned}
$$

Le schéma d'évaluation des expressions est spécifié par le jeu de règles suivant. Les règles sont constituées de jugements :

$$
\mathrm{st}, \mathrm{E} \vdash e \to v,\ n,\ \mathrm{st}\,' \ \subseteq\ \mathrm{Store} \times \mathrm{Env} \times \mathrm{Expr} \times \mathrm{Val} \times \mathrm{Nat}^+ \times \mathrm{Store}
$$

signifiant *Dans un environnement* E *et munie d'une mémoire* st*, l'expression e s'évalue en la valeur v en un temps n et modifie la mémoire* st *en* st′.

$$
\frac{[i{\leftarrow}v] \subseteq \mathrm{E}}{\mathrm{st}, \mathrm{E} \vdash i \to v,\ \mathsf{1},\ \mathrm{st}}\ [\text{D.Env}]
$$

$$
\frac{}{\mathrm{st}, \mathrm{E} \vdash (\mathsf{lambda}\ (i\ t)\ e) \to \langle i, e, \mathrm{E}\rangle,\ \mathsf{1},\ \mathrm{st}}\ [\text{D.Lambda}]
$$

$$
\frac{\begin{array}{r}\mathrm{st}, \mathrm{E} \vdash e_0 \to \langle i, e\,', \mathrm{E}\,'\rangle,\ n_0,\ \mathrm{st}_0 \\ \mathrm{st}_0, \mathrm{E} \vdash e_1 \to v_1,\ n_1,\ \mathrm{st}_1 \\ \mathrm{st}_1, \mathrm{E}\,'[i{\leftarrow}v_1] \vdash e\,' \to v,\ n,\ \mathrm{st}\,'\end{array}}{\mathrm{st}, \mathrm{E} \vdash (e_0\ e_1) \to v,\ 1 + n + n_0 + n_1,\ \mathrm{st}\,'}\ [\text{D.Apply}]
$$

$$
\frac{\mathrm{st}, \mathrm{E}_i \vdash e \to v,\ n,\ \mathrm{st}\,'}{\mathrm{st}, \mathrm{E} \vdash (\mathsf{plambda}\ (i\ k)\ e) \to v,\ n,\ \mathrm{st}\,'}\ [\text{D.Plambda}]
$$

$$
\frac{\mathrm{st}, \mathrm{E} \vdash e \to v,\ n,\ \mathrm{st}\,'}{\mathrm{st}, \mathrm{E} \vdash (\mathsf{proj}\ e\ d) \to v,\ n,\ \mathrm{st}\,'}\ [\text{D.Proj}]
$$

$$\frac{\begin{array}{c}\mathrm{st},\,\mathrm{E}\vdash e_0 \rightarrow \mathtt{new},\ n_0,\ \mathrm{st}_0\\ \mathrm{st}_0,\,\mathrm{E}\vdash e_1 \rightarrow v,\ n,\ \mathrm{st}_1\\ [l\leftarrow v'] \not\subseteq \mathrm{st}_1\end{array}}{\mathrm{st},\mathrm{E}\vdash (e_0\ e_1)\rightarrow l,\ n_0+n_1+1,\ \mathrm{St}_1[l\leftarrow v]}\ [\text{D.New}]$$

$$\frac{\begin{array}{c}\mathrm{st},\,\mathrm{E}\vdash e_0 \rightarrow \mathtt{get},\ n_0,\ \mathrm{st}_0\\ \mathrm{st}_0,\,\mathrm{E}\vdash e \rightarrow l,\ n_1,\ \mathrm{st}_1\\ [l\leftarrow v] \subseteq \mathrm{st}_1\end{array}}{\mathrm{st},\mathrm{E}\vdash (e_0\ e_1)\rightarrow v,\ n_0+n_1+1,\ \mathrm{st}_1}\ [\text{D.Get}]$$

$$\frac{\begin{array}{l}\mathrm{st},\mathrm{E}\vdash e_0 \rightarrow \mathtt{set},\ n_0,\ \mathrm{st}_0\\ \mathrm{st}_0,\mathrm{E}\vdash e_1 \rightarrow l,\ n_1,\ \mathrm{st}_1\\ \mathrm{st}_1,\mathrm{E}\vdash e_2 \rightarrow v,\ n_2,\ \mathrm{st}_2\end{array}}{\mathrm{st},\mathrm{E}\vdash (e_0\ e_1\ e_2)\rightarrow \mathtt{unit},\ n_0+n_1+n_2+1,\ \mathrm{St}_2[l\leftarrow v]}\ [\text{D.Set}]$$

Les temps d'exécutions des primitives ont été fixés arbitrairement à un top d'horloge. Toute autre option peut être prise pour modéliser une architecture particulière sans changer le principe de vérification des temps présenté dans ce chapitre. L'application d'une fonction à ses paramètres coûte un top (règle D.Lambda). La relation d'inclusion ($\subseteq$) est étendue aux environnements et mémoires vus comme des fonctions finies. Les abstractions et projections sur les types (règles D.Plambda et D.Proj) ne coûtent pas de temps puisque traitées à la compilation. Enfin, afin de diminuer le nombre de règles et de simplifier les preuves, la fonction de modification de la mémoire (règle D.Set) est d'arité deux. Notre langage étant unaire, nous donnons en temps utile la règle binaire associée.

3 Sémantique statique

Le système de vérification des types et des temps valide les descriptions fournies par le programmeur. Les types et temps sont plongés dans une algèbre (souvent appelée algèbre d'effet). Il s'agit, pour les types, du respect de l'alpha-convertibilité au sens du lambda-calcul (règles E.Drec, E.Subr et E.Poly) et de l'interconvertibilité d'un type récursif avec lui-même *déroulé* d'un cran (E.Fold).

$$\frac{j \notin \mathrm{FV}(d)}{(\mathtt{dlambda}\ (i\ k)\ d) \sim (\mathtt{dlambda}\ (j\ k)\ d[i\backslash j])}\ [\text{E.Dlambda}]$$

$$\frac{d_0 \sim d_0\,' \ \wedge\ d_1 \sim d_1\,'}{(d_0\ d_1) \sim (d_0\,'\ d_1\,')}\ [\text{E.Dapply}]$$

$$((\mathtt{dlambda}\ (i\ k)\ d_0)\ d_1) \sim d_0[i\backslash d_1] \qquad [\text{E.Betaconv}]$$

$$(\mathtt{drec}\ i\ t) \sim t[i\backslash(\mathtt{drec}\ i\ t)] \qquad [\text{E.Fold}]$$

$$\frac{j \notin \mathrm{FV}(t)}{(\mathtt{drec}\ i\ t) \sim (\mathtt{drec}\ j\ t[i\backslash j])}\ [\text{E.Drec}]$$

$$\frac{t_0 \sim t_0' \ \wedge \ t_1 \sim t_1' \ \wedge \ m \sim m'}{(\texttt{subr}\ m\ (t_1)\ t_0) \sim (\texttt{subr}\ m'\ (t_1')\ t_0')}\ [\text{E.Subr}]$$

$$\frac{j \notin \mathrm{FV}(t)}{(\texttt{poly}\ (i\ k)\ t) \sim (\texttt{poly}\ (j\ k)\ t[i \backslash j])}\ [\text{E.Poly}]$$

$$\frac{t \sim t'}{(\texttt{refof}\ t) \sim (\texttt{refof}\ t')}\ [\text{E.Refof}]$$

L'algèbre induite par $\oplus$ sur l'ensemble des temps est ACZ (Associative, commutative avec un élément absorbant, un zéro). La loi $\oplus$ est l'addition sur les entiers positifs étendue à l'élément `long`.

$$\begin{array}{rl} (\oplus\ m_1\ (\oplus\ m_2\ m_3)) \sim (\oplus\ (\oplus\ m_1\ m_2)\ m_3) & \text{Associativité} \\ (\oplus\ m_1\ m_2) \sim (\oplus\ m_2\ m_1) & \text{Commutativité} \\ (\oplus\ m\ \texttt{long}) \sim \texttt{long} & \text{Elément absorbant} \end{array}$$

$$(\oplus\ m_1\ m_2) \sim m_1 + m_2 \ \textit{si}\ m_i \neq \texttt{long} \quad \text{Additivité}$$

L'aspect important de cette algèbre est illustré par l'équation suivante :

$$m = (\oplus\ m\ m_0)$$

Quelque soit la constante de temps m_0, l'unique solution de l'équation pour la variable m est `long`. Nous ne disposerions pas de cette propriété si le domaine Time possédait un élément neutre. Les fonctions récursives ont des temps latents obéissant à des équations de ce style et qui seront donc contraints à `long`.

L'expression du système de vérification des sortes, des types et des temps nécessite un environnement de typage (TK). Comme notre langage ne possède qu'un seul espace de nom, TK appartient au domaine des fonction finies à valeur dans l'union des types et des sortes :

$$\mathrm{TK} \in \mathrm{TKenv} \ = \ \mathrm{Id} \rightarrow (\mathrm{Kind} \uplus \mathrm{Type})$$

L'environnement de typage initial maintient les types de toutes les valeurs de base et opérateurs primitifs ainsi que les sortes des types et temps de base. Il peut paraître surprenant que les identificateurs `1`, `2`, etc. ... soient liés à la fois aux valeurs de type entier et aux descriptions de temps. L'environnement TK_0 n'est cependant pas incohérent car il s'agit là d'objets différents (des temps ou des entier). Il est toujours possible de les différencier suivant le contexte. La valeur `unit` retournée par la fonction de modification de la mémoire est définie comme ayant le type `stm` (pour *statement*).

$$\begin{array}{rl} \mathrm{TK}_0 = [& \texttt{1} :: \texttt{time}, \texttt{2} :: \texttt{time}, \ldots, \texttt{long} :: \texttt{time}, \\ & \texttt{bool} :: \texttt{type}, \texttt{int} :: \texttt{type}, \ldots \\ & \texttt{0} : \texttt{int}, \texttt{1} : \texttt{int}, \texttt{2} : \texttt{int}, \ldots \\ & \texttt{unit} : \texttt{stm}, \#\texttt{t} :: \texttt{bool}, \#\texttt{f} :: \texttt{bool}, \\ & \texttt{new} : (\texttt{poly}\ (t\ \texttt{type})\ (\texttt{subr}\ \texttt{1}\ (t)\ (\texttt{refof}\ t))), \\ & \texttt{get} : (\texttt{poly}\ (t\ \texttt{type})\ (\texttt{subr}\ \texttt{1}\ ((\texttt{refof}\ t))\ t)), \\ & \texttt{set} : (\texttt{poly}\ (t\ \texttt{type})\ (\texttt{subr}\ \texttt{1}\ ((\texttt{refof}\ t)\ \ t)\ \texttt{stm})), \\ & + : (\texttt{subr}\ \texttt{2}\ (\texttt{int}\ \ \texttt{int})\ \texttt{int}), \\ & \ldots] \end{array}$$

Nous commençons par donner la spécification de la vérification des sortes des descriptions. Un jugement $\mathrm{TK} \vdash d :: k$ comme :

$$\mathrm{TK} \vdash d :: k \subseteq \mathrm{TKenv} \times \mathrm{Descr} \times \mathrm{Kind}$$

signifie *La description d possède pour sorte k dans l'environnement* TK. Les règles pour les identificateurs, les types récursifs, les type d'abstraction, les types polymorphes et les maximisations de temps suivent. Dans la règle K.Subr, le temps latent m dans `(subr` m `(`t_1`)` t_0`)` symbolise le temps d'exécution des fonctions de ce type.

$$\frac{[i :: k] \subseteq \mathrm{TK}}{\mathrm{TK} \vdash i :: k}\ [\text{K.Env}]$$

$$\frac{\mathrm{TK}[i :: k_1] \vdash d :: k_0}{\mathrm{TK} \vdash (\texttt{dlambda}\ (i\ k_1)\ d) :: (\texttt{dfunc}\ (k_1)\ k_0)}\ [\text{K.Dlambda}]$$

$$\frac{\begin{array}{l}\mathrm{TK} \vdash d_0 :: (\texttt{dfunc}\ (k_1)\ k_0) \\ \mathrm{TK} \vdash d_1 :: k_1\end{array}}{\mathrm{TK} \vdash (d_0\ d_1) :: k_0}\ [\text{K.Dapply}]$$

$$\frac{\mathrm{TK} \vdash m_i :: \texttt{time}}{\mathrm{TK} \vdash (\oplus\ m_0\ m_1) :: \texttt{time}}\ [\text{K.Sumtime}]$$

$$\frac{\begin{array}{l}\mathrm{TK} \vdash m :: \texttt{time} \\ \mathrm{TK} \vdash t_i :: \texttt{type}\end{array}}{\mathrm{TK} \vdash (\texttt{subr}\ m\ (t_1)\ t_0) :: \texttt{type}}\ [\text{K.Subr}]$$

$$\frac{\mathrm{TK} \vdash t :: \texttt{type}}{\mathrm{TK} \vdash (\texttt{refof}\ t) :: \texttt{type}}\ [\text{K.Refof}]$$

$$\frac{\mathrm{TK}[i :: \texttt{type}] \vdash t :: \texttt{type}}{\mathrm{TK} \vdash (\texttt{drec}\ i\ t) :: \texttt{type}}\ [\text{K.Drec}]$$

$$\frac{\mathrm{TK}[i :: k] \vdash t :: \texttt{type}}{\mathrm{TK} \vdash (\texttt{poly}\ (i\ k)\ t) :: \texttt{type}}\ [\text{K.Poly}]$$

Les règles spécifiant la vérification des types utilisent des jugements $\mathrm{TK} \vdash e : t\$\, m$ de la forme :

$$\mathrm{TK} \vdash e : t\$\, m \subseteq \mathrm{TKenv} \times \mathrm{Expr} \times \mathrm{Type} \times \mathrm{Time}$$

qui signifient *l'expression e est associée au type t et au temps m dans l'environnement* TK. Les règles de typage pour les identificateurs, les abstractions, les applications, les quantification universelles et les projections suivent. Comme précédemment pour la sémantique dynamique, les temps des étapes de base sont tous fixés arbitrairement à `1`.

$$\frac{\mathrm{TK} \vdash e : t \,\$\, m \qquad t \sim t' \ \wedge \ m \sim m'}{\mathrm{TK} \vdash e : t' \,\$\, m'} \text{[S.Equiv]}$$

$$\frac{[i : t] \subseteq \mathrm{TK}}{\mathrm{TK} \vdash i : t \,\$\, \mathtt{1}} \text{[S.Env]}$$

$$\frac{\mathrm{TK}[i : t_1] \vdash e : t_0 \,\$\, m \qquad \mathrm{TK} \vdash t_1 :: \mathtt{type}}{\mathrm{TK} \vdash (\mathtt{lambda}\ (i\ t_1)\ e) : (\mathtt{subr}\ m\ (t_1)\ t_0) \,\$\, \mathtt{1}} \text{[S.Lambda]}$$

$$\frac{\mathrm{TK} \vdash e_0 : (\mathtt{subr}\ m_l\ (t_1)\ t_0) \,\$\, m_0 \qquad \mathrm{TK} \vdash e_1 : t_1 \,\$\, m_1}{\mathrm{TK} \vdash (e_0\ e_1) : t_0 \,\$\, (\oplus\ (\oplus\ m_l\ \mathtt{1})\ (\oplus\ m_0\ m_1))} \text{[S.Apply]}$$

$$\frac{\mathrm{TK}[i :: k] \vdash e : t \,\$\, m}{\mathrm{TK} \vdash (\mathtt{plambda}\ (i\ k)\ e) : (\mathtt{poly}\ (i\ k)\ t) \,\$\, m} \text{[S.Plambda]}$$

$$\frac{\mathrm{TK} \vdash e : (\mathtt{poly}\ (i\ k)\ t) \,\$\, m \qquad \mathrm{TK} \vdash d :: k}{\mathrm{TK} \vdash (\mathtt{proj}\ e\ d) : t[i \backslash d] \,\$\, m} \text{[S.Proj]}$$

Noter le *va-et-vient* du temps latent entre les règles S.Lambda (où il est défini) et S.Apply (où il est utilisé). L'abstraction et la projection d'une expression sur une description n'interfèrent pas avec le temps associé à l'expression. Cette propriété est due au caractère générique du typage. En effet, le code exécuté par une fonction ne dépend pas des types sur lesquels elle est abstraite. Le temps d'exécution suit donc le même principes.

4 Consistance

Traditionnellement, la preuve de consistance entre les sémantiques statique et dynamique affirme la cohérence du type de l'expression avec la valeur en laquelle l'expression s'évalue. Dans notre cas, deux cohérences sont à vérifier. Premièrement, le temps d'exécution de l'expression doit être consistant avec le (inférieur au) temps vérifié statiquement. Deuxièmement, la valeur fournie par l'exécution du programme doit être consistante avec son type (et ce, en tenant compte des temps latent contenus dans les types). Le langage *ETL* contenant des constructions impératives, nous utilisons, comme dans le chapitre précédent, la technique d'induction de point fixe maximal.

4.1 Consistance entre les valeurs et les types

L'expression du théorème de consistance nécessite une propriété de consistance entre les valeurs et les types ainsi qu'une autre entre les temps d'exécution (n) et les temps (m). Les valeurs pouvant dépendre d'un état mémoire, quelques définitions sont nécessaires.

Définition 4.46 (Mémoire abstraite) *Une mémoire abstraite* ST *est une fonction finie associant un type aux locations mémoires :*

$$\mathrm{ST} \in \mathrm{AbStore} = \mathrm{Loc} \rightarrow \mathrm{Type}$$

Définition 4.47 (Mémoire typée) *Une mémoire typée est une couple composé d'une mémoire* st *et d'une mémoire abstraite* ST *telles que* dom(st) = dom(ST)*. Elle est notée* st : ST*.*

La consistance des valeurs conservées en mémoire avec les types gardés dans la mémoire abstraite n'est pas imposée. Cette condition, nécessaire lors de la démonstration, n'est requise que dans la relation de consistance suivante.

Nous pouvons maintenant énoncer la relation de consistance entre les valeurs et les types. Cette relation est définie par induction sur la structure des types. Sur l'ensemble des valeurs de base, la relation suppose les deux environnements initiaux (E_0 et TK_0) comme consistant.

Définition 4.48 *Nous définissons la consistance entre les valeurs et les types par rapport à une mémoire typée de la façon suivante :*

$\mathrm{st}{:}\mathrm{ST} \models v : t \iff$

si $v = b$ *alors* $[i \leftarrow v] \subseteq \mathrm{E}_0$ et $[i : t] \subseteq \mathrm{TK}_0$

si $t \sim (\mathtt{subr}\ m\ (t_1)\ t_0)$ *alors* $v = \langle i, e, \mathrm{E}\rangle$ *et*

$$\exists\, \mathrm{TK}\ s.t. \begin{cases} \mathrm{st} : \mathrm{ST} \models \mathrm{E} : \mathrm{TK} \\ \mathrm{TK}[i : t_1] \vdash e : t_0\ \$\ m \\ \mathrm{TK} \vdash t_1 :: \mathtt{type} \end{cases}$$

si $t \sim (\mathtt{refof}\ t')$

alors $v = l$ *et* $[l \leftarrow v'] : [l : t'] \subseteq \mathrm{st} : \mathrm{ST}$ *et* $\mathrm{st}{:}\mathrm{ST} \models v' : t'$

si $t \sim (\mathtt{drec}\ i\ t')$

alors $\mathrm{st}{:}\mathrm{ST} \models v : t'[i \backslash (\mathtt{drec}\ i\ t')]$

si $t \sim (\mathtt{poly}\ (i\ k)\ t_0)$

alors $\forall\, d \in \mathrm{Descr},\ \mathrm{TK}_0 \vdash d :: k \Rightarrow \mathrm{st}{:}\mathrm{ST} \models v : t_0[i \backslash d]$

De la même manière, nous définissons la consistance entre un environnement E *et un environnement de typage* TK *:*

$\mathrm{st}{:}\mathrm{ST} \models \mathrm{E} : \mathrm{TK} \iff$

$$\begin{cases} \mathrm{TK}_0 = \mathrm{TK} \quad sur \quad \mathrm{Id} \rightarrow \mathrm{Kind} \\ \mathrm{dom}(\mathrm{E}) \cup \mathrm{dom}(\mathrm{TK}_0) = \mathrm{dom}(\mathrm{TK}) \\ \forall i \in \mathrm{dom}(\mathrm{E}),\ \mathrm{st} : \mathrm{ST} \models \mathrm{E}(i) : \mathrm{TK}(i) \end{cases}$$

La consistance entre les temps d'exécution et les temps n'est que l'extension à ω de l'inégalité sur les entiers positifs.

Définition 4.49 ($\models \leq$)

$$\models n \leq m \Leftrightarrow \begin{array}{l} si\ \ m \sim \mathtt{long}\ \ alors\ \ true \\ si\ \ m \not\sim \mathtt{long}\ \ alors\ \ n \leq m \end{array}$$

4.2 Induction de point fixe maximal

Il faut redéfinir la relation de consistance comme une fonction monotone de $\mathcal{U}$ à valeur dans $\mathcal{U}$. Sur l'univers $\mathcal{U}$ des quadruplets :

$$\begin{array}{rcl} \mathcal{U} & = & \mathcal{U}_1 \uplus \mathcal{U}_2 \\ \mathcal{U}_1 & \subseteq & \text{Store} \times \text{AbStore} \times \text{TKenv} \times \text{Val} \times \text{Type} \\ \mathcal{U}_2 & \subseteq & \text{Store} \times \text{AbStore} \times \text{TKenv} \times \text{Env} \times \text{TKenv} \end{array}$$

on définit la fonction $\mathcal{F}$ par deux fonctions adjointes :

$$\begin{array}{rccc} \mathcal{F} : & \mathcal{P}(\mathcal{U}) & \longrightarrow & \mathcal{P}(\mathcal{U}) \\ & \mathcal{Q} = \mathcal{Q}_1 \uplus \mathcal{Q}_2 & \longmapsto & \mathcal{F}_1(\mathcal{Q}) \uplus \mathcal{F}_2(\mathcal{Q}) \end{array}$$

correspondant aux deux parties de la relation de consistance :

$$\begin{array}{l} \mathcal{F}_1(\mathcal{Q}) = \{(\text{st}, \text{ST}, v, t) \mid \\ \quad \text{si } v = b \text{ alors } [i{\leftarrow}v] \subseteq \text{E}_0 \text{ et } [i : t] \subseteq \text{TK}_0 \\ \quad \text{si } t \sim (\texttt{subr}\ m\ (t_1)\ t_0) \text{ alors} \\ \qquad v = \langle i, e, \text{E} \rangle \text{ et} \\ \qquad \exists\ \text{TK s.t.} \left\{ \begin{array}{l} (\text{st}, \text{ST}, \text{E}, \text{TK}) \in \mathcal{Q}_2 \\ \text{TK}[i : t_1] \vdash e : t_0\ \$\ m \\ \text{TK} \vdash t_1 :: \texttt{type} \end{array} \right. \\ \quad \text{si } t \sim (\texttt{refof}\ t') \\ \qquad \text{alors } v = l \text{ et } [l{\leftarrow}v'] : [l : t'] \subseteq \text{st} : \text{ST} \text{ et } (\text{st}, \text{ST}, v', t') \in \mathcal{Q}_1 \\ \quad \text{si } t \sim (\texttt{drec}\ i\ t') \\ \qquad \text{alors } (\text{st}, \text{ST}, v, t'[i \backslash (\texttt{drec}\ i\ t')]) \in \mathcal{Q}_1 \\ \quad \text{si } t \sim (\texttt{poly}\ (i\ k)\ t_0) \\ \qquad \text{alors } \forall\ d \in \text{Descr}, \text{TK}_0 \vdash d :: k \Rightarrow (\text{st}, \text{ST}, v, t_0[i \backslash d]) \in \mathcal{Q}_1\ \} \end{array}$$

$$\begin{array}{l} \mathcal{F}_2(\mathcal{Q}) = \{(\text{st}, \text{ST}, \text{E}, \text{TK}) \mid \\ \quad \text{TK}_0 = \text{TK} \ \textit{sur}\ \text{Id} \rightarrow \text{Kind} \\ \quad \text{dom}(\text{E}) \cup \text{dom}(\text{TK}_0) = \text{dom}(\text{TK}) \\ \quad \forall i \in \text{dom}(\text{E}),\ (\text{st}, \text{ST}, \text{E}(i), \text{TK}(i)) \in \mathcal{Q}_1\} \end{array}$$

L'opérateur $\mathcal{F}$ ainsi défini est monotone et peut donc être utilisé par l'induction de point fixe maximal.

Propriété 4.50 (monotonicité) *L'opérateur $\mathcal{F}$ est monotone sur le treillis formé de l'ensemble $\mathcal{P}(\mathcal{U})$ ordonné par l'inclusion $\subseteq$.*

Preuve Par disjonction des cas. Soient deux ensembles $\mathcal{Q}$ et $\mathcal{Q}'$ tels que $\mathcal{Q} \subseteq \mathcal{Q}'$.

Cas $\boxed{q = (\text{st}, \text{ST}, v, t)}$
On sait que q appartient à $\mathcal{F}_1(\mathcal{Q})$

- Si $v = b$ alors il existe un identificateur i tel que $[i{\leftarrow}v] \subseteq \text{E}_0$ et $[i : t] \subseteq \text{TK}_0$. On en déduit immédiatement que q appartient à $\mathcal{F}(\mathcal{Q}')$.
- Si $t \sim (\texttt{subr}\ m\ (t_1)\ t_0)$ alors $v = \langle i, e, \text{E} \rangle$ et il existe TK tel que $\text{TK} \vdash t_1 :: \texttt{type}$, $\text{TK}[i : t_1] \vdash e : t_0\ \$\ m$ et $(\text{st}, \text{ST}, \text{E}, \text{TK}) \in \mathcal{Q}_2$. On en déduit que $(\text{st}, \text{ST}, \text{E}, \text{TK}) \in \mathcal{Q}'_2$ et que q appartient à $\mathcal{F}(\mathcal{Q}')$.
- Si $t \sim (\texttt{refof}\ t')$ alors $v = l$, $[l{\leftarrow}v'] : [l : t'] \subseteq \text{st} : \text{ST}$ et $(\text{st}, \text{ST}, v', t') \subseteq \mathcal{Q}_1$. On en déduit que $(\text{st}, \text{ST}, v', t') \subseteq \mathcal{Q}'_1$ et que q appartient à $\mathcal{F}(\mathcal{Q}')$.

- Si $t \sim (\texttt{poly}\ (i\ k)\ t')$ alors pour toute description d, on a l'implication $\mathrm{TK} \vdash d :: k \Rightarrow (\mathrm{st}, \mathrm{ST}, v, t'[i \backslash d]) \in \mathcal{Q}_1$. Comme $\mathcal{Q} \subseteq \mathcal{Q}'$, on a $\mathrm{TK} \vdash d :: k \Rightarrow (\mathrm{st}, \mathrm{ST}, v, t'[i \backslash d]) \in \mathcal{Q}'_1$ et q appartient à $\mathcal{F}(\mathcal{Q}')$.
- Si $t \sim (\texttt{drec}\ i\ t')$ alors $(\mathrm{st}, \mathrm{ST}, v, t'[i \backslash t])$ appartient à $\mathcal{Q}_1$, donc appartient à $\mathcal{Q}'_1$.

Cas $\boxed{q = (\mathrm{st}, \mathrm{ST}, e, \mathrm{TK})}$
On sait que q appartient à $\mathcal{F}_2(\mathcal{Q})$. De la définition de $\mathcal{F}_2$, on a $\mathrm{TK}_0 = \mathrm{TK}$ sur $\mathrm{Id} \rightarrow \mathrm{Kind}$, $\mathrm{dom(E)} \cup \mathrm{dom}(\mathrm{TK}_0) = \mathrm{dom(TK)}$ et pour tout identificateur i appartenant à dom(E), $(\mathrm{st}, \mathrm{ST}, \mathrm{E}(i), \mathrm{TK}(i)) \in \mathcal{Q}_1$. On en déduit que $(\mathrm{st}, \mathrm{ST}, \mathrm{E}(i), \mathrm{TK}(i)) \in \mathcal{Q}'_1$ et donc que q appartient à $\mathcal{F}(\mathcal{Q}')$. □

4.3 Lemmes annexes

Avant d'énoncer le théorème de consistance, quelques définitions et lemmes sont nécessaires. Nous commençons par définir la notion de succession de mémoires typées.

Définition 4.51 (Succession) *On dit qu'une mémoire typée,* $\mathrm{st}' : \mathrm{ST}'$*, succède à une autre* $\mathrm{st} : \mathrm{ST}$*, ce que l'on note* $\mathrm{st} : \mathrm{ST} \sqsubseteq \mathrm{st}' : \mathrm{ST}'$*, ssi* $\mathrm{st} \subseteq \mathrm{st}'$ *et pour toute valeur* v*, pour tout type* t *et pour tout environnement* k*, l'implication suivante est vérifiée :*

$$\mathrm{st} : \mathrm{ST} \models v : t \Rightarrow \mathrm{st}' : \mathrm{ST}' \models v : t$$

Le premier lemme n'est qu'une autre formulation de la définition précédente.

Lemme 4.52

$$\left.\begin{array}{l} \mathrm{st} : \mathrm{ST} \sqsubseteq \mathrm{st}' : \mathrm{ST}' \\ \mathrm{st} : \mathrm{ST} \models v : t \end{array}\right\} \Rightarrow \mathrm{st}' : \mathrm{ST}' \models v : t$$

Preuve Directe. □

Le deuxième n'est que l'extension de la définition aux environnements d'évaluation et de typage.

Lemme 4.53

$$\left.\begin{array}{l} \mathrm{st} : \mathrm{ST} \sqsubseteq \mathrm{st}' : \mathrm{ST}' \\ \mathrm{st} : \mathrm{ST} \models \mathrm{E} : \mathrm{TK} \end{array}\right\} \Rightarrow \mathrm{st}' : \mathrm{ST}' \models \mathrm{E} : \mathrm{TK}$$

Preuve Directe par la définition 4.48. □

La succession des états mémoires est transitive.

Lemme 4.54 (Transitivité)

$$\left.\begin{array}{l} \mathrm{st} : \mathrm{ST} \sqsubseteq \mathrm{st}' : \mathrm{ST}' \\ \mathrm{st}' : \mathrm{ST}' \sqsubseteq \mathrm{st}'' : \mathrm{ST}'' \end{array}\right\} \Rightarrow \mathrm{st} : \mathrm{ST} \sqsubseteq \mathrm{st}'' : \mathrm{ST}''$$

Preuve Directe en s'appuyant sur la propriété de transitivité de l'implication. □

Vient ensuite un lemme lié à la création d'une nouvelle location en mémoire. Cette dernière n'atteint pas à la propriété de succession :

Lemme 4.55 (Expansion mémoire)

$$\left.\begin{array}{l}\mathrm{st} : \mathrm{ST} \models v : t \\ [l \leftarrow v'] \not\subseteq \mathrm{st}\end{array}\right\} \Rightarrow \mathrm{st} : \mathrm{ST} \sqsubseteq \mathrm{St}[l \leftarrow v] : \mathrm{ST}[l : t]$$

Preuve La proposition $\mathrm{st} : \mathrm{ST} \sqsubseteq \mathrm{St}[l \leftarrow v] : \mathrm{ST}[l : t]$ est équivalente à l'inclusion $\mathrm{st} \subseteq \mathrm{St}[l \leftarrow v]$ (évidente) et à

$$\forall v', t', \mathrm{st} : \mathrm{ST} \models v' : t' \Rightarrow \mathrm{St}[l \leftarrow v] : \mathrm{ST}[l : t] \models v' : t'$$

On construit l'ensemble $\mathcal{Q}$ comme :

$$\begin{array}{rcl}\mathcal{Q} & = & \mathcal{Q}_1 \uplus \mathcal{Q}_2 \\ \mathcal{Q}_1 & = & \{(\mathrm{st}', \mathrm{ST}', v', t') \mid \mathrm{st} : \mathrm{ST} \models v' : t'\} \\ \mathcal{Q}_2 & = & \{(\mathrm{st}', \mathrm{ST}', \mathrm{E}', \mathrm{TK}') \mid \mathrm{st} : \mathrm{ST} \models \mathrm{E}' : \mathrm{TK}'\}\end{array}$$

où $\mathrm{st}' = \mathrm{St}[l \leftarrow v]$ et $\mathrm{ST}' = \mathrm{ST}[l : t]$. On doit maintenant montrer que $\mathcal{Q}$ est inclus dans $\mathcal{F}(\mathcal{Q})$.

Cas $\boxed{q = (\mathrm{st}', \mathrm{ST}', v', t')}$
On sait que q appartient à $\mathcal{Q}_1$ et que $\mathrm{st} : \mathrm{ST} \models v' : t'$.

- Si $v' = b$ alors $[i \leftarrow v'] \subseteq \mathrm{E}_0$ et $[i : t'] \subseteq \mathrm{TK}_0$. On en déduit directement que q appartient à $\mathcal{F}(\mathcal{Q})$.
- Si $t' \sim (\texttt{subr}\ m\ (t_1)\ t_0)$ alors $v' = \langle i, e, \mathrm{E}'' \rangle$ et il existe un environnement TK'' tel que $\mathrm{TK}'' \vdash t_1 :: \texttt{type}$, $\mathrm{TK}''[i : t_1] \vdash e : t_0\ \$\ m$ et $\mathrm{st} : \mathrm{ST} \models \mathrm{E}'' : \mathrm{TK}''$. On en déduit que $(\mathrm{st}', \mathrm{ST}', \mathrm{E}'', \mathrm{TK}'') \in \mathcal{Q}_2$ et donc que q appartient à $\mathcal{F}(\mathcal{Q})$.
- Si $t' \sim (\texttt{refof}\ t'')$ alors $v' = l''$, $[l'' \leftarrow v''] : [l'' : t''] \subseteq \mathrm{st} : \mathrm{ST}$ et $\mathrm{st} : \mathrm{ST} \models v'' : t''$. On en déduit que $(\mathrm{st}', \mathrm{ST}', v'', t'')$ appartient à $\mathcal{Q}_1$ et que $[l'' \leftarrow v''] : [l'' \leftarrow t''] \subseteq \mathrm{st}' : \mathrm{ST}'$ donc que q appartient à $\mathcal{F}(\mathcal{Q})$.
- Si $t' \sim (\texttt{poly}\ (i\ k)\ t'')$ alors pour toute description d, on a l'implication $\mathrm{TK}_0 \vdash d :: k \Rightarrow \mathrm{st} : \mathrm{ST} \models v' : t''[i \backslash d]$. On en déduit $\mathrm{TK}_0 \vdash d :: k \Rightarrow (\mathrm{st}', \mathrm{ST}', v', t''[i \backslash d]) \in \mathcal{Q}_1$. On conclut que q appartient à $\mathcal{F}(\mathcal{Q})$.
- Si $t' \sim (\texttt{drec}\ i\ t'')$ alors $\mathrm{st} : \mathrm{ST} \models v' : t''[i \backslash t']$. On en déduit que $(\mathrm{st}', \mathrm{ST}', v', t''[i \backslash t'])$ appartient à $\mathcal{Q}_1$ et donc que q appartient à $\mathcal{F}(\mathcal{Q})$.

Cas $\boxed{q = (\mathrm{st}', \mathrm{ST}', \mathrm{E}', \mathrm{TK}')}$
On sait que q appartient à $\mathcal{Q}_2$ et que $\mathrm{st} : \mathrm{ST} \models \mathrm{E}' : \mathrm{TK}'$. De la relation de consistance, on sait que $\mathrm{dom}(\mathrm{E}') \cup \mathrm{dom}(\mathrm{TK}_0) = \mathrm{dom}(\mathrm{TK}')$, $\mathrm{TK}' = \mathrm{TK}_0$ sur $\mathrm{Id} \rightarrow \mathrm{Kind}$ et que pour tout identificateur i de $\mathrm{dom}(\mathrm{E}')$ on a $\mathrm{st} : \mathrm{ST} \models \mathrm{E}'(i) : \mathrm{TK}'(i)$. On en déduit que pour tout i, $(\mathrm{st}', \mathrm{ST}', \mathrm{E}'(i), \mathrm{TK}'(i)) \in \mathcal{Q}_1$ et donc que q appartient à $\mathcal{F}(\mathcal{Q})$. □

Toujours pour les effets de bord, nous avons besoin d'un lemme concernant la modification d'une valeur en mémoire.

Lemme 4.56 (Modification)

$$\left.\begin{array}{l}\mathrm{st} : \mathrm{ST} \models v : t \\ [l \leftarrow v'] : [l : t] \subseteq \mathrm{st} : \mathrm{ST}\end{array}\right\} \Rightarrow \mathrm{st} : \mathrm{ST} \sqsubseteq \mathrm{St}[l \leftarrow v] : \mathrm{ST}$$

Preuve La preuve est directe et suit le même principe que pour le lemme précédent. □

Nous terminons sur un lemme lié à l'expansion commune des environnements de typage et d'évaluation.

Lemme 4.57 (Expansion d'environnement) *Soient une valeur v, un type t, un environnement* E *et un TK-environnement* TK*. L'implication suivante est vérifiée :*

$$\left.\begin{array}{l} \mathrm{st} : \mathrm{ST} \models v : t \\ \mathrm{st} : \mathrm{ST} \models \mathrm{E} : \mathrm{TK} \\ \mathrm{TK} \vdash t :: \mathrm{type} \end{array}\right\} \Rightarrow \mathrm{st} : \mathrm{ST} \models \mathrm{E}[i \leftarrow v] : \mathrm{TK}[i : t]$$

Preuve Directe, par disjonction des cas sur la structure des types dans la définition 4.48. □

4.4 Preuve

Nous pouvons maintenant énoncer et prouver le théorème principal de ce chapitre :

Théorème 4.58 (Consistance) *Soient* st *une mémoire,* E *un environnement, e une expression, v une valeur, n un entier positif,* TK *un environnement de typage, t un type, m un temps et* ST *une mémoire abstraite. L'implication suivante est vérifiée :*

$$\left.\begin{array}{l} \mathrm{st}, \mathrm{E} \vdash e \rightarrow v, n, \mathrm{st}' \\ \mathrm{TK} \vdash e : t \$ m \\ \mathrm{st} : \mathrm{ST} \models \mathrm{E} : \mathrm{TK} \end{array}\right\} \Rightarrow \exists \mathrm{ST}' \ s.t. \left\{\begin{array}{l} \mathrm{st} : \mathrm{ST} \sqsubseteq \mathrm{st}' : \mathrm{ST}' \\ \mathrm{st}' : \mathrm{ST}' \models v : t \\ \models n \leq m \end{array}\right.$$

et se lit *si l'on prend deux environnements (typage et évaluation) consistants, si l'expression e a pour type t et pour temps m, enfin si l'expression e s'évalue en la valeur v en n étapes avec une mémoire* st′*, alors, il existe une mémoire abstraite* ST′ *telle que la mémoire typée* st′:ST′ *succède à celle de départ, la valeur v est consistante avec le type t et le temps de calcul n est inférieur au temps m.*

Preuve Par disjonction des cas sur la structure des expressions. Par induction sur la longueur de dérivation (d'évaluation) n et sur la taille de l'expression quand n est constant. Les hypothèses de la preuve sont :

$$\left\{\begin{array}{ll} \mathrm{st} : \mathrm{ST} \models \mathrm{E} : \mathrm{TK} & (a) \\ \mathrm{TK} \vdash e : t \$ m & (b) \\ \mathrm{st}, \mathrm{E} \vdash e \rightarrow v, n, \mathrm{st}' & (c) \end{array}\right.$$

Cas $\boxed{e = i}$
Des règles D.Env et S.Env, on sait que $[i \leftarrow v] : [i : t] \subseteq \mathrm{E} : \mathrm{TK}$ et que $n \sim m \sim \mathtt{1}$. De l'hypothèse (a), on déduit que st:ST $\models v$: t.

Cas $\boxed{e = (\mathtt{lambda}\ (i\ t)\ e)}$
Direct, en s'appuyant sur la relation de consistance.

Cas $\boxed{e = (e_0\ e_1)}$
Des règles D.Apply et S.Apply, on déduit que :

$$\left\{\begin{array}{ll} v = v' & (1) \\ n = n_0 + n_1 + n' + 1 & (2) \\ \mathrm{st}_1, \mathrm{E}'[i \leftarrow v_1] \vdash e' \rightarrow v', n', \mathrm{st}' & (3) \\ \mathrm{st}_0, \mathrm{E} \vdash e_1 \rightarrow v_1, n_1, \mathrm{st}_1 & (4) \\ \mathrm{st}, \mathrm{E} \vdash e_0 \rightarrow \langle i, e', \mathrm{E}' \rangle, n_0, \mathrm{st}_0 & (5) \end{array}\right.$$

et

$$\begin{cases} t \sim t_0 & (6) \\ m \sim (\oplus\ (\oplus\ m_0\ m_1)\ (\oplus\ m_l\ \mathtt{1})) & (7) \\ \mathrm{TK} \vdash e_1\ :\ t_1\ \$\ m_1 & (8) \\ \mathrm{TK} \vdash e_0\ :\ (\mathtt{subr}\ m_l\ (t_1)\ t_0)\ \$\ m_0 & (9) \end{cases}$$

Depuis les propositions (5), (9), (a) et par induction sur n_0, on sait qu'il existe ST_0 telle que :

$$\begin{cases} \mathrm{st}\ :\ \mathrm{ST} \sqsubseteq \mathrm{st}_0\ :\ \mathrm{ST}_0 & (10) \\ \mathrm{st}_0\ :\ \mathrm{ST}_0 \models \langle i, e', \mathrm{E}' \rangle\ :\ (\mathtt{subr}\ m_l\ (t_1)\ t_0) & (11) \\ \models n_0 \leq m_0 \end{cases}$$

Depuis (10) et le lemme 4.53, on sait que $\mathrm{st}_0 : \mathrm{ST}_0 \models \mathrm{E} : \mathrm{TK}$. Donc par induction sur n_1 avec les propositions (4) et (8), il existe ST_1 telle que :

$$\begin{cases} \mathrm{st}_0\ :\ \mathrm{ST}_0 \sqsubseteq \mathrm{st}_1\ :\ \mathrm{ST}_1 & (12) \\ \mathrm{st}_1\ :\ \mathrm{ST}_1 \models v_1\ :\ t_1 & (13) \\ \models n_1 \leq m_1 \end{cases}$$

Depuis la proposition (11) et le lemme 4.52, on a $\mathrm{st}_1 : \mathrm{ST}_1 \models \langle i, e', \mathrm{E}' \rangle : (\mathtt{subr}\ m_l\ (t_1)\ t_0)$. De la relation de consistance, il existe TK' tel que :

$$\begin{cases} \mathrm{st}_1\ :\ \mathrm{ST}_1 \models \mathrm{E}'\ :\ \mathrm{TK}' & (14) \\ \mathrm{TK}'[i\ :\ t_1] \vdash e'\ :\ t_0\ \$\ m_l & (15) \\ \mathrm{TK}' \vdash t_1\ ::\ \mathtt{type} & (16) \end{cases}$$

Depuis les propositions (13), (14), (16) et le lemme 4.57, on sait que $\mathrm{st}_1 : \mathrm{ST}_1 \models \mathrm{E}'[i \leftarrow v] : \mathrm{TK}'[i : t_1]$. Par induction sur n' avec les propositions (3) et (15), il existe ST' telle que :

$$\begin{cases} \mathrm{st}_1\ :\ \mathrm{ST}_1 \sqsubseteq \mathrm{st}'\ :\ \mathrm{ST}' \\ \mathrm{st}'\ :\ \mathrm{ST}' \models v'\ :\ t_0 \\ \models n' \leq m_l \end{cases}$$

De (2) et (7), on a $\models n \leq m$ et par transitivité (lemme 4.54) $\mathrm{st} : \mathrm{ST} \sqsubseteq \mathrm{st}' : \mathrm{ST}'$.

Cas $\boxed{e = (\mathtt{plambda}\ (i\ k)\ e)}$
Direct.

Cas $\boxed{e = (\mathtt{proj}\ e\ d)}$
Des règles D.Proj et S.Proj, on a $\mathrm{st}, \mathrm{E} \vdash e \rightarrow v, n, \mathrm{st}'$ (1) et :

$$\begin{cases} t \sim t'[i \backslash d] & (2) \\ \mathrm{TK} \vdash d\ ::\ k & (3) \\ \mathrm{TK} \vdash e\ :\ (\mathtt{poly}\ (i\ k)\ t')\ \$\ m & (4) \end{cases}$$

Par induction sur e (n est constant) avec les propositions (1) et (4), il existe ST' telle que :

$$\begin{cases} \mathrm{st}\ :\ \mathrm{ST} \sqsubseteq \mathrm{st}'\ :\ \mathrm{ST}' & (5) \\ \mathrm{st}'\ :\ \mathrm{ST}' \models v\ :\ (\mathtt{poly}\ (i\ k)\ t') & (6) \\ \models n \leq m \end{cases}$$

De l'hypothèse (a) et par la relation de consistance, on sait que $\mathrm{TK}_0 = \mathrm{TK}$ sur $\mathrm{Id} \rightarrow \mathrm{Kind}$. De (6), on déduit l'implication pour tout d', $\mathrm{TK} \vdash d'\ ::\ k \Rightarrow \mathrm{st}'\ :\ \mathrm{ST}' \models v\ :\ t'[i \backslash d']$. De la proposition (3), on a $\mathrm{st}' : \mathrm{ST}' \models v : t'[i \backslash d]$ et donc $\mathrm{st}' : \mathrm{ST}' \models v : t$ par (2).

Cas `new`
De la règle D.New, on a :

$$\begin{cases} v = l & (1)\\ n = n_0 + n_1 + 1 & (2)\\ \mathrm{st}' = \mathrm{St}_1[l \leftarrow v] & (3)\\ [l \leftarrow v'] \not\subseteq \mathrm{st}_1 & (4)\\ \mathrm{st}_0, \mathrm{E} \vdash e_1 \rightarrow v,\ n,\ \mathrm{st}_1 & (5)\\ \mathrm{st}, \mathrm{E} \vdash e_0 \rightarrow \mathtt{new},\ n_0,\ \mathrm{st}_0 & (6) \end{cases}$$

Pour le typage, on utilise la règle S.Apply et on obtient :

$$\begin{cases} t \sim t_0 & (7)\\ m \sim (\oplus\ (\oplus\ m_0\ m_1)\ (\oplus\ m_l\ \mathtt{1})) & (8)\\ \mathrm{TK} \vdash e_1\ :\ t_1 \,\$\, m_1 & (9)\\ \mathrm{TK} \vdash e_0\ :\ (\mathtt{subr}\ m_l\ (t_1)\ t_0) \,\$\, m_0 & (10) \end{cases}$$

Par induction sur n_0 avec les proposition (6) et (10), on sait qu'il existe ST_0 telle que :

$$\begin{cases} \mathrm{st}\ :\ \mathrm{ST} \sqsubseteq \mathrm{st}_0\ :\ \mathrm{ST}_0\\ \mathrm{st}_0 : \mathrm{ST}_0 \models \mathtt{new} : (\mathtt{subr}\ m_l\ (t_1)\ t_0)\\ \models n_0 \leq m_0 \end{cases}$$

Par la relation de consistance sur les valeur de base, on sait que $\mathrm{st}_0 : \mathrm{ST}_0 \models$ `new`: `(poly (`t `type) (subr 1 (`t`) (refof` t`)))` et donc que pour toute description d, l'implication suivante est vérifiée :

$$\mathrm{TK} \vdash d :: \mathtt{type} \Rightarrow \mathrm{st}_0 : \mathrm{ST}_0 \models \mathtt{new} : (\mathtt{subr}\ \mathtt{1}\ (d)\ (\mathtt{refof}\ d))$$

En fait, le programmeur a projeté explicitement la fonction `new` sur le bon type. De la proposition (10) et par la relation de consistance, on sait que $\mathrm{TK} \vdash t_1 ::$ `type`. On en déduit que $m_l \sim 1$ et $t_0 \sim$ `(refof` t_1`)`. Par induction sur n_1 avec les propositions (5) et (9), on sait qu'il existe ST_1 telle que :

$$\begin{cases} \mathrm{st}_0\ :\ \mathrm{ST}_0 \sqsubseteq \mathrm{st}_1\ :\ \mathrm{ST}_1\\ \mathrm{st}_1 : \mathrm{ST}_1 \models v :\ t_1\\ \models n_1 \leq m_1 \end{cases}$$

Soit $\mathrm{ST}' = \mathrm{ST}_1[l : t_1]$. Par le lemme d'expansion (4.57), on a $\mathrm{st}_1 : \mathrm{ST}_1 \sqsubseteq \mathrm{st}' : \mathrm{ST}'$ et par la relation de consistance, $\mathrm{st}' : \mathrm{ST}' \models l$: `(refof` t_1`)`.

Cas `get`
Direct, même principe que précédemment.

Cas `set`
De la règle D.Set, on a :

$$\begin{cases} v = \mathsf{unit}\\ n = n_0 + n_1 + n_2 + 1\\ \mathrm{st}' = \mathrm{St}_2[l \leftarrow v]\\ \mathrm{st}_1, \mathrm{E} \vdash e_2 \rightarrow v,\ n_2,\ \mathrm{st}_2\\ \mathrm{st}_0, \mathrm{E} \vdash e_1 \rightarrow l,\ n_1,\ \mathrm{st}_1\\ \mathrm{st}, \mathrm{E} \vdash e_0 \rightarrow \mathtt{set},\ n_0,\ \mathrm{st}_0 \end{cases}$$

Nous étendons trivialement le langage des types et expressions pour admettre des fonctions binaires :

$$\frac{\begin{array}{l}\mathrm{TK} \vdash\ e_0\ :\ (\texttt{subr}\ m_l\ (t_1\ \ t_2)\ t_0)\,\$\,m_0\\ \mathrm{TK} \vdash\ e_1\ :\ t_1\,\$\,m_1\\ \mathrm{TK} \vdash\ e_2\ :\ t_2\,\$\,m_2\\ m \sim (\oplus\ (\oplus\ m_0\ (\oplus\ m_1\ m_2))\ (\oplus\ m_l\ \texttt{1}))\end{array}}{\mathrm{TK} \vdash\ (e_0\ e_1\ \ e_2)\ :\ t_0\,\$\,m}\ [\text{S.Apply2}]$$

On en déduit que :

$$\begin{cases} m \sim (\oplus\ (\oplus\ m_0\ (\oplus\ m_1\ m_2))\ (\oplus\ m_l\ \texttt{1})) \\ t \sim t_0 \end{cases}$$

Par induction sur n_0, on sait qu'il existe ST_0 telle que :

$$\begin{cases} \mathrm{st}\ :\ \mathrm{ST} \sqsubseteq \mathrm{st}_0\ :\ \mathrm{ST}_0 \\ \mathrm{st}_0 : \mathrm{ST}_0 \models \texttt{set} : (\texttt{subr}\ m_l\ (t_1\ \ t_2)\ t_0) \quad (1) \\ \models n_0 \leq m_0 \end{cases}$$

Par la relation de consistance sur les valeurs de base, on sait que $\mathrm{st}_0 : \mathrm{ST}_0 \models \texttt{set}$: `(poly (`$t$ `type) (subr 1 ((refof` t`) `t`) stm))` et donc que pour toute description d, l'implication suivante est vérifiée :

$$\mathrm{TK} \vdash d\ ::\ \texttt{type} \Rightarrow \mathrm{st}_0\ :\ \mathrm{ST}_0 \models \texttt{set} : (\texttt{subr 1}\ ((\texttt{refof}\ d)\ \ d)\ \texttt{stm})$$

De la relation de consistance et de la proposition (1), on sait que $\mathrm{TK} \vdash t_2 :: \texttt{type}$. On en déduit que $m_l \sim 1$, que $t_0 \sim \texttt{stm}$ et que $t_1 \sim (\texttt{refof}\ t_2)$. Par induction sur n_1 et n_2 on sait qu'il existe ST_1 et ST_2 telles que

$$\begin{cases} \mathrm{st}_0\ :\ \mathrm{ST}_0 \sqsubseteq \mathrm{st}_1\ :\ \mathrm{ST}_1 \\ \mathrm{st}_1 : \mathrm{ST}_1 \models l : (\texttt{refof}\ t_2) \quad (2) \\ \models n_1 \leq m_1 \end{cases} \qquad \text{et} \qquad \begin{cases} \mathrm{st}_1\ :\ \mathrm{ST}_1 \sqsubseteq \mathrm{st}_2\ :\ \mathrm{ST}_2 \\ \mathrm{st}_2 : \mathrm{ST}_2 \models v : t_2 \quad (3) \\ \models n_2 \leq m_2 \end{cases}$$

De la proposition (2), on sait que l appartient à $\mathrm{dom}(\mathrm{st}_1)$. Par le lemme de modification (4.56) et la proposition (3), on a $\mathrm{st}_2 : \mathrm{ST}_2 \sqsubseteq \mathrm{St}_2[l \leftarrow v] : \mathrm{ST}_2$. Enfin, par définition, on a $\mathrm{st}' : \mathrm{ST}_2 \models \mathsf{unit}$: `stm`. □

5 Algorithme de vérification

La sémantique statique est spécifiée par deux jeux de règles (vérification des sortes et vérification des types). Les algorithmes associés sont directs. L'algorithme de vérification des sortes est de type :

`KCA` $\in$ `(TKenv` $\times$ `Descr)` $\longrightarrow$ `Kind`

et fonctionne par induction sur la structure des descriptions :

```
KCA(TK,d) = case d in
i          => if [i :: k] ⊆ TK then k else fail
(dlambda (i k1) d)
           => let k0 = KCA(TK[i :: k1],d)
              (dfunc (k1) k0)
(d0 d1)    => let k = KCA(TK,d0)
              let k1 = KCA(TK,d1)
```

```
            if k = (dfunc (k_1) k_0) then k_0 else fail
(drec i t)  if KCA(TK[i::type],t) = type then type else fail
(subr m (t_1) t_0)
        => if KCA(TK,m) = time
              and KCA(TK,t_0) = type
              and KCA(TK,t_1) = type
           then type else fail
(poly (i k) t)
        => if KCA(TK[i::k],t) = type then type else fail
(refof t) => if KCA(TK,t) = type then type else fail
(⊕ m_0 m_1)
        => if KCA(TK,m_0) = time
              and KCA(TK,m_1) = time
           then time else fail
else    => fail
```

L'algorithme de vérification des sortes jouit de bonnes propriétés.

Théorème 4.59 (Terminaison) *L'algorithme* `KCA` *termine.*

Preuve L'algorithme `KCA` fonctionne par induction sur la structure des descriptions qui sont de degré d'imbrication fini. □

Théorème 4.60 (Correction des sortes) *Soient* TK *un environnement de typage, d une description et k une sorte. L'équivalence suivante est vérifiée :*

$$\mathrm{TK} \vdash d :: k \Longleftrightarrow \texttt{KCA}(\mathrm{TK}, d) = k$$

Preuve Par induction et disjonction des cas sur la structure des expressions. La preuve étant simple, nous ne donnons qu'un cas.

Cas $\boxed{d = (d_0\ d_1)}$
Soit un environnement de typage TK,

$\mathrm{TK} \vdash (d_0\ d_1) :: k_0 \Longleftrightarrow$

$\mathrm{TK} \vdash d_1 :: k_1$ et $\mathrm{TK} \vdash d_0 ::$ `(dfunc (`k_1`) `k_0`)`	Par la règle K.Dapply
$k_1 = \texttt{KCA}(\mathrm{TK}, d_1)$ et `(dfunc (`k_1`) `k_0`)` $= \texttt{KCA}(\mathrm{TK}, d_0)$	Par induction sur d_1 et d_0
$k_0 = \texttt{KCA}(\mathrm{TK}, (d_0\ d_1))$	Par l'algorithme

□

L'algorithme de vérification des types `TTCA` :

`TTCA` $\in$ (TKenv $\times$ Expr) $\longrightarrow$ Type $\times$ Time

calcule une description de type et de temps pour toute expression dans un environnement de typage TK :

```
TTCA(TK,e) = case e in
i       => if [i : t] ⊆ TK then (t,1) else fail
(lambda (i t_1) e)
        => if KCA(TK,t_1) = type then
             let (t_0,m) = TTCA(TK[i : t_1],e)
```

```
                ((subr m (t1) t0), 1)
             else fail
(e0 e1)   => let (t, m0) = TTCA(TK, e0)
             let (t1, m1) = TTCA(TK, e1)
             if sim(t,(subr ml (t1) t0))
                then (t0,(⊕ (⊕ m0 m1) (⊕ 1 ml)))
                else fail
(plambda (i k) e)
          => let (t, m) = TTCA(TK[i :: k], e)
             ((poly (i k) t), m)
(proj e d)
          => let (t, m) = TTCA(TK, e)
             let k = KCA(TK, d)
             if sim(t,(poly (i k) t0)) then (t0[i\d], m) else fail
else      => fail
```

La fonction de pattern-matching `sim` vérifie si deux descriptions sont similaires (interconvertible par la relation $\sim$). Elle est définie par induction structurelle sur le domaine des descriptions. Dans le cas des temps, elle effectue une normalisation à `long` ou, à défaut, à une somme de variables de temps triée lexicalement et d'une constante entière.

L'algorithme `TTCA` jouit de bonnes propriétés.

Théorème 4.61 (Terminaison) *L'algorithme* `TTCA` *termine.*

Preuve L'algorithme `TTCA` fonctionne par induction structurelle sur les expressions qui sont de degré d'imbrication fini. □

Théorème 4.62 (Correction) *Soient* TK *environnement de typage, e une expression,* t_0 *et* t_1 *deux types et* m_0 *et* m_1 *deux temps.*

$$\left.\begin{array}{l} \mathrm{TK} \vdash e \,:\, t \,\$\, m \\ (t', m') = \texttt{TTCA}(\mathrm{TK}, e) \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} t \sim t' \\ m \sim m' \end{array}\right.$$

Preuve Par induction structurelle et par disjonction des cas sur les expressions. Nous ne présentons que deux cas intéressants.

Cas $\boxed{e = (e_0\ e_1)}$
De la règle S.Apply, on déduit que :

$$\left\{\begin{array}{l} \mathrm{TK} \vdash e_0 \,:\, (\texttt{subr}\ m_l\ (t_1)\ t) \,\$\, m_0 \\ \mathrm{TK} \vdash e_1 \,:\, t_1 \,\$\, m_1 \\ m \sim (\oplus\ (\oplus\ m_0\ m_1)\ (\oplus\ m_l\ \texttt{1})) \end{array}\right.$$

De l'algorithme, on sait que :

$$\left\{\begin{array}{l} \texttt{TTCA}(\mathrm{TK}, e_0) = ((\texttt{subr}\ m'_l\ (t'_1)\ t'), m'_0) \\ \texttt{TTCA}(\mathrm{TK}, e_1) = (t'_1, m'_1) \\ m' \sim (\oplus\ (\oplus\ m'_0\ m'_1)\ (\oplus\ m'_l\ \texttt{1})) \end{array}\right.$$

Par induction sur e_0 et e_1, on obtient les égalités :

$$\left\{\begin{array}{l} (\texttt{subr}\ m_l\ (t_1)\ t) \sim (\texttt{subr}\ m'_l\ (t'_1)\ t') \\ m_0 \sim m'_0 \end{array}\right. \quad \text{et} \quad \left\{\begin{array}{l} t_1 \sim t'_1 \\ m_1 \sim m'_1 \end{array}\right.$$

ce qui suffit à montrer que $t \sim t'$ et $m \sim m'$.

Cas $\boxed{e = (\texttt{plambda}\ (i\ k)\ e_0)}$
De la règle S.Plambda et de l'algorithme, on déduit que :

$$\begin{cases} t \sim (\texttt{poly}\ (i\ k)\ t_0) \\ \mathrm{TK}[i :: k] \vdash e_0 : t_0\ \$\ m \end{cases} \quad \text{et} \quad \begin{cases} t' \sim (\texttt{poly}\ (i\ k)\ t'_0) \\ \texttt{TTCA}(\mathrm{TK}[i :: k], e_0) = (t'_0, m'_0) \end{cases}$$

Par induction sur e_0, on obtient les égalités suffisantes pour conclure. □

6 L'opérateur de point fixe

Comme nous l'avons dit précédemment, l'opérateur de point fixe, `Y`, est exprimable dans notre langage. Quand `Y` est appliqué à une fonction potentiellement récursive, il retourne une fonction de temps latent `long`. Par contre, si `Y` est appliqué à une fonction non potentiellement récursive, il retourne une fonction de temps latent identique à celui de la fonction d'entrée. Avec un peu d'édulcorant syntaxique, l'opérateur de point fixe est intégré à notre langage de la manière suivante :

$$\begin{aligned} e ::= &\ \ldots \\ &\texttt{(rec}\ (f\ i)\ e\texttt{)} \end{aligned}$$

On peut ainsi écrire la fonction factorielle :

$$\texttt{FACT} \equiv \texttt{(rec}\ (f\ i)\ \texttt{(if (=}\ i\ \texttt{0) 1 (*}\ i\ \texttt{((}f\texttt{) (-}\ i\ \texttt{1)))))}$$

Les opérateurs de point fixe sont généralement donnés pour un schéma d'évaluation paresseux. Notre langage utilisant l'appel par valeur, nous devons explicitement retarder l'évaluation des auto-applications. Ceci est fait en enfermant les expressions $(x\ x)$ dans une fermeture.

$$\texttt{Y} \equiv \lambda f.(\lambda x.(f\ \lambda.(x\ x))\ \lambda x.(f\ \lambda.(x\ x)))$$

et

$$\texttt{(rec}\ (f\ i)\ e\texttt{)} \equiv (\texttt{Y}\ \lambda f.\lambda i.e)$$

En *ETL*, l'opérateur `Y` précédemment défini s'écrit :

```
Y ≡ (plambda ((t type)(m1 time)(m2 time))
       (lambda ((f (subr m1
                         ((subr (⊕ m1 6) () (subr m2 (t) t)))
                         (subr m2 (t) t))))
          ((lambda ((x (drec tx
                              (subr (⊕ m1 3) (tx) (subr m2 (t) t)))))
              (f (lambda () (x x))))
           (lambda ((x (drec tx
                              (subr (⊕ m1 3) (tx) (subr m2 (t) t)))))
              (f (lambda () (x x)))))))
```

Dans cette définition, nous avons généralisé l'abstraction et l'application à de multiples arguments. Le type de l'identificateur `x` est récursif car celui-ci est appliqué à lui même. Si nous ne disposions pas du constructeur de type `drec`, l'écriture de l'opérateur `Y` deviendrait impossible et notre langage fortement normalisable. Le type de `Y` suit :

```
Y : (poly ((t type)(m1 time)(m2 time))
       (subr (⊕6 m1)
             ((subr m1
                    ((subr (⊕6 m1) () (subr m2 (t) t)))
                    (subr m2 (t) t)))
             (subr m2 (t) t)))
```

Pour obtenir le type de la fonction `FACT` ci-dessous, nous avons supposé que les primitives `=`, `*` et `-` avaient pour temps latents `1`. Le temps pris par un test `if` est le $\oplus$ des temps du prédicat, de la branche vraie et de la branche fausse. Une meilleure approximation consisterait à prendre la somme des temps du test et du maximum des deux branches. L'introduction d'un opérateur `maxtime` sur les temps ne poserait pas de difficultés théoriques majeures.

```
FACT : (poly ((m3 time)(m4 time))
          (subr 1
                ((subr m3 () (subr m4 (int) int)))
                (subr (⊕16 m3 m4) (int) int)))
```

La fonction factorielle s'obtient en appliquant l'opérateur `Y` à l'expression `FACT`. Pour cela, quelques projections de type et temps sont nécessaires :

```
((proj Y int 1 long) (proj FACT 7 long)) :  (subr long (int) int)
```

Les projections de `Y` et `FACT` sont contraintes par les équations suivantes. Le programmeur doit les résoudre au moment de la programmation. Le lecteur peut vérifier que, dans ce système, la valeur `long` est la seule solution possible pour les variables m_2 et m_4.

$$\begin{aligned} m_1 &\sim 1 \\ (\oplus\ 6\ m_1) &\sim m_3 \\ m_2 &\sim m_4 \\ t &\sim \texttt{int} \\ m_2 &\sim (\oplus\ 16\ m_3\ m_4) \end{aligned}$$

7 Reconstruction partielle

La programmation dans le langage précédent est un exercice parfois rébarbatif; cela est dû aux nombreuses déclarations. A noter que les types des résultats des fonctions ne sont pas requis. Nous allons présenter une méthode reprise de [OG89] qui autorise le programmeur à éluder les types de certains paramètres. On pourrait penser à faire plus mais la reconstruction partielle du polymorphisme est indécidable [B85] et la reconstruction totale est un problème ouvert.

7.1 Système de type

Le langage utilisé est décrit par la grammaire suivante. Celle-ci reprend les principales fonctionnalités de la précédente en y ajoutant le minimum nécessaire à la reconstruction.

Le non terminal r correspond aux types reconstruits par le système. L'abstraction est autorisée sans spécification du type de l'argument. Le domaine des identificateurs inclut des variables d'unification non accessibles au programmeur et utilisées pour les types non spécifiés.

$k \in$ Kind		
$k ::=$ `time` | `type`		
$d \in$ Descr		
$d ::= m \mid t$		
$m \in$ Time		
$m ::=$ `1` | `2` | `3` | ... | `long`		
	$(\oplus\ m\ m)$	
	i	
$t \in$ Type		
$t ::=$	(`subr` m (t) t)	Type d'abstraction explicite
	(`poly` (i k) t)	
	r	
$r ::=$	i	
	(`subr` m (r) r)	Type d'abstraction implicite
$e \in$ Expr		
$e ::=$	i	
	(`lambda` (i t) e)	Abstraction explicitement typée
	(`lambda` (i) e)	Abstraction implicitement typée
	(e e)	
	(`plambda` (i k) e)	
	(`proj` e d)	
$i \in$ Id		
$i ::=$	$j \mid f \mid x$	Variables et constantes
	$\delta \mid \tau \mid \mu$	Variables d'unification

La vérification des classes reste inchangée. En effet, nous n'avons pas introduit ou modifié les divers constructeurs de type. Les règles d'équivalence entre les types et entre les temps sont donc identiques et l'algèbre induite sur les temps par $\oplus$ reste ACZ. Le système des types et des temps est mis à jour par l'introduction d'une règle R.ILambda spécifiant la vérification des abstraction implicitement typées.

$$\frac{\mathrm{TK} \vdash\ e\ :\ t \$ m \qquad t \sim t'\ \wedge\ m \sim m'}{\mathrm{TK} \vdash\ e\ :\ t' \$ m'}\ [\text{R.Equiv}]$$

$$\frac{[i\ :\ t] \subseteq \mathrm{TK}}{\mathrm{TK} \vdash\ i\ :\ t \$ \mathtt{1}}\ [\text{R.Env}]$$

$$\frac{\mathrm{TK}[i : t_1] \vdash e : t_0 \,\$\, m \qquad \mathrm{TK} \vdash t_1 :: \texttt{type}}{\mathrm{TK} \vdash (\texttt{lambda}\ (i\ t_1)\ e) : (\texttt{subr}\ m\ (t_1)\ t_0) \,\$\, \texttt{1}}\ [\text{R.Lambda}]$$

$$\frac{\mathrm{TK}[i : r] \vdash e : t \,\$\, m \qquad \mathrm{TK} \vdash r :: \texttt{type}}{\mathrm{TK} \vdash (\texttt{lambda}\ (i)\ e) : (\texttt{subr}\ m\ (r)\ t) \,\$\, \texttt{1}}\ [\text{R.ILambda}]$$

$$\frac{\mathrm{TK} \vdash e_0 : (\texttt{subr}\ m_l\ (t_1)\ t_0) \,\$\, m_0 \qquad \mathrm{TK} \vdash e_1 : t_1 \,\$\, m_1}{\mathrm{TK} \vdash (e_0\ e_1) : t_0 \,\$\, (\oplus\ (\oplus\ m_l\ \texttt{1})\ (\oplus\ m_0\ m_1))}\ [\text{R.Apply}]$$

$$\frac{\mathrm{TK}[i :: k] \vdash e : t \,\$\, m}{\mathrm{TK} \vdash (\texttt{plambda}\ (i\ k)\ e) : (\texttt{poly}\ (i\ k)\ t) \,\$\, m}\ [\text{R.Plambda}]$$

$$\frac{\mathrm{TK} \vdash e : (\texttt{poly}\ (i\ k)\ t) \,\$\, m \qquad \mathrm{TK} \vdash d :: k}{\mathrm{TK} \vdash (\texttt{proj}\ e\ d) : t[i \backslash d] \,\$\, m}\ [\text{R.Proj}]$$

La consistance avec la sémantique dynamique est préservée. L'abstraction, qu'elle soit typée implicitement ou explicitement, s'évalue en une fermeture. La relation de consistance entre les valeurs et les types s'appuie sur le nouveau système de typage mais reste inchangée dans sa formulation. La preuve s'étend sans difficultés au cas supplémentaire de l'abstraction implicite.

7.2 Reconstruction

L'algorithme de reconstruction partielle des types doit aussi reconstruire les temps latents des fonctions. Si l'équivalence sur les types est purement structurelle, il n'en est pas de même pour les temps. Considérons l'exemple suivant :

```
g :  (subr (⊕ m1 m4) (int) int)
h :  (subr (⊕ m2 m3) (int) int)

(lambda (f) (if BoolExp (f g) (f h))
```

où `g` et `h` sont des fonctions ayant les types cités. Dans l'abstraction sur `f`, les deux applications de `f` aux fonctions `g` et `h` imposent l'égalité de leurs types et donc des temps latents $(\oplus\ \texttt{m}_1\ \texttt{m}_4)$ et $(\oplus\ \texttt{m}_2\ \texttt{m}_3)$. Dû aux propriétés ACZ des temps, l'ensemble des solutions de l'équation $(\oplus\ \texttt{m}_1\ \texttt{m}_4) \sim (\oplus\ \texttt{m}_2\ \texttt{m}_3)$ est infini et n'admet pas de temps principal (au sens d'une substitution). Le principe d'unification à la Robinson [R65] est donc inadapté.

L'algorithme de reconstruction utilise l'algorithme d'unification `U` qui unifie deux types t_1 et t_2 en calculant une substitution et un ensemble de contraintes. L'ensemble de contraintes assure l'égalité des temps latents contenus dans les deux types à unifier. La substitution est unificatrice des types dépourvus de leurs temps latents :

```
U∈ (Type × Type) → (Subst × Const)

U(t1,t2) = case (t1,t2) in
(i,i)      => ([],∅)
(τ,r) or (r,τ)
           => if τ ∉ r then ([τ\r],∅) else fail
((subr m (t1) t0),(subr m' (t1') t0'))
           let (s1,c1) = U(t1,t1')
           let (s0,c0) = U(s1t0,s1t0')
           (s0s1,c1∪c0∪{(m,m')})
((poly (i k) t),(poly (i' k) t'))
           U([i\j]t,[i'\j]t') where j is fresh
```

Comme le montre [JG91], l'unification et correcte. Si `U` unifie deux types en fournissant une substitution et un ensemble de contraintes, alors les types obtenus en appliquant la substitution sont équivalents modulo les équivalences contenues dans l'ensemble de contraintes.

L'algorithme de reconstruction `RA` rend un type t, un temps m, une substitution s et ensemble de contraint c. Le type t est celui de l'expression e dans l'environnement TK et m est son temps. En fait, t et m ne sont valides que si l'ensemble de contraintes admet une solution.

```
RA ∈ (TKenv × Expr) → (Type × Time × Subst × Const)

RA(TK,e) = case e in
i          => if [i:t] ⊆ TK then (t,1,[],∅) else fail
(lambda (i t) e)
           => let (t0,m0,s0,c0) = RA(TK[i:t],e)
              ((subr m0 (t) t0),1,s0,c0)
(lambda (i) e)
           => let (t0,m0,s0,c0) = RA(TK[i:τ],e) where τ is fresh
              ((subr m0 (sτ) t0),1,s0,c0)
(e0 e1)    => let (t0,m0,s0,c0) = RA(TK,e0)
              let (t1,m1,s1,c1) = RA(s0TK,e1)
              let (s,c) = U(s1t0,(subr μ (t1) τ)) where μ and τ are fresh
              (sτ,(⊕ (⊕ m0 m1) (⊕ μ 1)),ss1s0,c0∪c1∪c)
(plambda (i k) e)
           => let (t,m,s,c) = RA(TK[i::k],e)
              let {ii} = FV(c) − FV(s(TK[i::k]))
              ((poly (i k) t),m,s,c∪c[i\j][ii\δi]) where j and δi are fresh
(proj e d)
           => let (t,m,s,c) = RA(TK,e)
              if t = (poly (i k) t0) and KCA(TK,d) = k
                 then (t0[i\d],m,s,[i\d]c)
              else fail
```

Cet algorithme est simple à appréhender sauf dans le cas de l'abstraction polymorphe (`plambda`). L'ensemble de contrainte construit est double. La première partie (c) propage les contraintes sur les paramètres de l'abstraction et ainsi sur les descriptions sur lesquelles ces paramètres seront projetés. La deuxième ($c[i\backslash j][i_i\backslash d_i]$) assure que l'ensemble de contraintes est réellement polymorphe sur les paramètres de l'abstraction en simulant une projection explicite sur j.

L'algorithme `RA` jouit de bonnes propriétés. Il est juste et complet par rapport à la spécification sémantique [JG91].

7.3 Résolution

Lorsque l'algorithme fournit un quadruplet (t,m,s,c), le type t et le temps m ne sont valides que si l'ensemble de contraintes possède une solution. Ce dernier se normalise sous la forme suivante :

$$\begin{array}{rcl} (\oplus\ m_{1,1}\ m_{1,2}\ \cdots\ m_{1,p_1}) & \sim & (\oplus\ m'_{1,1}\ m'_{1,2}\ \cdots\ m'_{1,q_1}) \\ & \vdots & \\ (\oplus\ m_{n,1}\ m_{n,2}\ \cdots\ m_{n,p_n}) & \sim & (\oplus\ m'_{n,1}\ m'_{n,2}\ \cdots\ m'_{n,q_n}) \end{array}$$

où $m_{i,j}$ est une variable d'unification de temps, la constante `long` ou une variable de programmation vue comme une constante. Résoudre une équation de ce système revient à effectuer l'unification des deux termes. L'algorithme d'unification utilisé doit intégrer les propriétés ACZ de l'algèbre des temps. Nous proposons une unification ACZ adaptée de l'algorithme d'unification AC proposé par [S75]. Ce dernier simplifié pour le domaine des temps et l'opérateur $\oplus$ est le suivant :

`UAC` $\in$ (Time $\times$ Time) $\rightarrow$ Subst

```
UAC(m,m') = case (m,m') in
(μ,μ) => []
(μ,m) or (m,μ) =>
          if μ ∈ m then fail else [μ\m]
((⊕ m1 ... mp),(⊕ m'1 ... m'q)) =>
          let M = solve([m1,...,mp],[m'1,...,m'q])
          choose-sol(M)
```

Les deux premiers cas sont standards. Pour le cas d'une somme de temps, toute variable peut s'unifier avec n'importe quel sous-terme de l'autre somme (commutativité) ou même avec une partie de cette autre somme (associativité). La matrice `M` retournée par la fonction `solve` est la solution de l'équation diophantienne linéaire homogène associée à l'équation. Cette matrice codant pour toutes les solutions, la fonction `choose-sol` en choisit une de façon non déterministe et la fournit sous forme d'une substitution. Résoudre le système consiste alors à appliquer successivement cet algorithme à toutes les équations du système de contraintes. Deux étapes restent à mettre en œuvre avant cela. On doit d'abord réduire notre système ACZ en un système AC. Pour cela, il suffit de choisir de façon non déterministe les variables égales à `long`, et ce en respectant les équations. En effet une variable μ obéissant à l'équation $(\oplus\ \mu\ \mu_0) \sim$ `10` ne peut pas être `long`. Ensuite, il faut normaliser le système en éliminant les équations où apparaît `long` (propriété d'absorption) et éliminer les doubles dans les équations restantes :

$$(\oplus\ \mu\ m_0) \sim (\oplus\ \mu\ m_1)\ \Rightarrow\ m_0 \sim m_1$$

L'unification AC est NP-complète [KN86]. Stickel [S75] a proposé un algorithme dont la terminaison ne fut prouvée que par Fages [F83]. Ceci dit, nous n'en utilisons pas la pleine

puissance puisque la seule fonctions AC que nous ayons à gérer est $\oplus$. La complexité de notre algorithme est néanmoins déplorable. Celle-ci est exponentielle sur le nombre de variables (choix des variables égales à `long`) et exponentielle sur les ensembles de solutions de chaque équation (un appel à `choose-sol` pour chaque équation). Malgré les diverses optimisations possibles, comme l'utilisation de la méthode du point fixe (chapitre 3) dès que la forme du système s'y prête, cette complexité intrinsèque reste problématique. Jouvelot et Gifford [JG91] ont proposé une solution polynomiale à un problème similaire. Il s'agissait pour eux de résoudre les contraintes d'effet dont l'algèbre est ACUI (Associative, Commutative, Unitaire et Idempotente). Une solution suivant la même démarche reste envisageable dans le cas ACZ. Cependant, il est utile de noter que pour la plupart des programmes, le système de contrainte est petit, voir vide [J92].

8 Conclusion

Les gains de notre système sont ceux apportés par le caractère explicite du typage (vs. typage implicite du chapitre précédent). La compilation séparée est facilitée par le caractère explicite du polymorphisme. La simplicité de la description des temps d'exécution n'est compensée que par la richesse du langage traité (applicatif et impératif). Notre système est prouvé consistant avec un schéma d'évaluation standard et les algorithmes sont prouvés corrects.

Appendice : Exemples

L'intérêt de l'opérateur de point fixe (section 6) est théorique. Voici deux programmes plus pragmatiques : la fonction `map` de *Lisp* et la fonction `reduce` à la *FP*. Pour finir, nous discutons des interférences possibles entre les effets de bord et la récursion.

Les abstractions sont étendues aux arités multiples. Le `let`, le `if` et l'opérateur de point fixe `rec` sont introduits dans la syntaxe des expressions. Dans l'expression `rec`, l'identificateur i est la variable locale à la définition récursive. Le temps m est le temps de la fonction définie (généralement `long`). Ensuite, viennent le type du paramètre, le type du résultat et l'expression définissant la fonction.

e ::= ...

`(let (`i e`)` e`)`	Définition locale
`(if` e e e`)`	Test
`(rec` i m `(`t`)` t e`)`	Opérateur de point fixe

Les règles de typage associées à ces nouvelles expressions sont directes. La seule particularité réside dans la règle de typage du test où le temps total de l'expression est la somme des temps du test, de la branche vraie et de la branche fausse (cf. section 6).

Enfin, nous supposons disposer du type `(pairof` t_1 t_2`)` décrivant les paires d'objets de types t_1 et t_2. Nous disposons aussi des constructeurs et accesseurs associés : `cons`, `car` et `cdr`. Les temps d'exécution de ces primitives sont fixés à `1`.

```
cons : (poly ((t1 type)(t2 type)) (subr 1 (t1 t2) (pairof t1 t2)))
car  : (poly ((t1 type)(t2 type)) (subr 1 ((pairof t1 t2)) t1))
cdr  : (poly ((t1 type)(t2 type)) (subr 1 ((pairof t1 t2)) t2))
```

Les fonctions `map` et `reduce` travaillant sur des listes, nous définissons le constructeur de type `(listof` t`)` pour décrire les listes d'objets de type t. La sorte `(dfunc (type) type)`

indique qu'appliqué à une description de sorte `type`, le constructeur `listof` retourne un type.

```
listof = (dlambda ((t type)) (drec l (pairof t l)))
       :: (dfunc (type) type)

listofint = (listof int)
          :: type
```

La fonction `map` est polymorphe sur deux types et un temps. Le premier type est celui des objets de la liste prise en paramètre. Le second est celui des objets de la liste résultat. Le temps est le temps latent de la fonction appliquée aux éléments de la liste. Le temps latent de la fonction `map` est `long`.

```
map = (plambda ((t1 type)(t2 type)(m time))
         (rec map-itern long
              ((f (subr m (t1) t2))
               (l (listof t1)))
              (listof t2)
           (if (null? l)
                     (proj nil t2 (listof t2))
               ((proj cons t2 (listof t2))
                        (f ((proj car t1 (listof t1)) l))
                        (map-intern f ((proj cdr t1 (listof t1)) l))))))
    : (poly ((t1 type)(t2 type)(m time))
         (subr long ((subr m (t1) t2) (listof t1)) (listof t2)))
```

Nous présentons maintenant la réduction utilisée dans le langage *FP* comme opérateur d'insertion dans les listes [4]. Munie d'une fonction binaire et d'une liste, elle construit une fermeture qui, appliquée à une valeur d'accumulation, rend le résultat de la réduction :

$$\begin{aligned} (insert\ f\ <x>) &= \lambda a.(f\ x\ a) \\ (insert\ f\ <x,y,z,...>) &= \lambda a.(f\ x\ ((insert\ f\ <y,z,...>)\ a)) \end{aligned}$$

La fonction `insert` est polymorphe sur le type des objets contenus dans la liste et sur le temps latent de la fonction réductrice. Le temps latent de `insert`, ainsi que celui de la fermeture construite, est égal à `long`.

```
insert = (plambda ((t type)(m time))
            (rec insert-bis long
                           ((f (subr m (t t) t))
                            (l (listof t)))
                           (subr long (t) t)
               (if (null? l)
                         (lambda ((x t)) x)
                   (let ((z (insert-bis f (cdr l))))
                      (lambda ((x t)) (f (car l) (z x)))))))
          : (poly ((t type)(m time))
               (subr long
                     ((subr m (t t) t)
```

[4] Merci à P. Fradet pour cet exemple original.

```
                (listof t))
               (subr long (t) t)))
```

Il est clair que pour créer une récursion il faut effectuer une auto-application, soit en redéfinissant `Y`, soit à travers la forme spéciale `rec`. Dans les deux cas, le système de typage impose un temps `long`. On peut néanmoins s'interroger sur les interférences possibles avec les primitives d'effets de bord. Même si cela est impossible (Le système est prouvé; section 4), nous allons tenter de créer une récursion à travers une référence en mémoire. Le principe est de définir la fonction `mapfirst` qui construit une liste dont le premier élément est obtenu par calcul direct et dont la queue l'est en appliquant une fonction conservée en mémoire à une adresse précise.

Supposons qu'une fonction polymorphe du type adéquat soit conservée en mémoire et associée à la variable globale $\texttt{ref}_0$. Celle-ci a été définie grâce à une fonction de même type déjà existante.

```
ref0 : (refof (poly ((m time)(t1 type)(t2 type))
                (subr m ((listof t1)) (listof t2))))
```

Nous pouvons maintenant définir la fonction `mapfirst` utilisant la référence $\texttt{ref}_0$ et en supposant l'existence d'une fonction `map-function` :

```
mapfirst = (plambda ((m time)(t1 type)(t2 type))
              (lambda ((l (listof t1)))
                 (cons (map-function (car l))
                       ((proj (get ref0) m t1 t2) (cdr l)))))
         : (poly ((m time)(t1 type)(t2 type))
              (subr (⊕ 18 m m-map-function)
                    ((listof t1))
                    (listof t2)))
```

L'affectation de la référence $\texttt{ref}_0$ avec la fonction `mapfirst` est interdite. En effet, le temps latent de la fonction `mapfirst` est la somme de `m` (le temps latent de la fonction résidant à l'adresse $\texttt{ref}_0$) et d'une constante (`18`). Les deux types sont donc incompatibles. L'affectation devient possible à condition de réviser les types en supprimant l'abstraction sur le temps `m` et en le remplaçant par `long` :

```
ref0 : (refof (poly ((t1 type)(t2 type))
                (subr long ((listof t1)) (listof t2))))

mapfirst = (plambda ((t1 type)(t2 type))
              (lambda ((l (listof t1)))
                 (cons (map-function (car l))
                       ((proj (get ref0) t1 t2) (cdr l)))))
         : (poly ((t1 type)(t2 type))
              (subr long ((listof t1)) (listof t2))))
```

et l'affectation devient possible :

```
(set ref0 mapfirst) : stm

(get ref0) : (poly ((t1 type)(t2 type))
               (subr long ((listof t1)) (listof t2)))
```

– – –◇◇◇– – –

Conclusion

Bien que motivée par de nombreux intérêts pratiques, l'analyse automatique de complexité en temps des programmes n'a pas eu un développement rapide. La majorité des solutions proposées dans la littérature possèdent deux bases communes : restriction du langage à un sous-ensemble analysable automatiquement et calcul d'un résultat aussi précis que possible. Ces systèmes sont en deux étapes. La phase dynamique construisant un système d'équations récursives à partir du programme ne possède généralement pas de preuve de correction. La phase statique résout le système récursif par des méthodes au cas par cas ou en utilisant l'appariement de formes. Elle n'est ni générale, ni totalement automatique.

Bien que ces méthodes soient attrayantes d'un point de vue théorique, elles ne peuvent guère être intégrées de manière pleinement automatique dans un compilateur. Dans ce cas (ex : placement statique de tâches sur les machines parallèles), une complexité en temps très précise est le plus souvent inutile. Une information moins riche mais obtenue automatiquement est préférable et suffisante.

Nous avons donc pris les hypothèses inverses, à savoir : un langage réaliste contenant tous les principes de la programmation applicative et impérative sur lequel un système réellement automatique calcule des informations de temps simples. De plus, nous voulions être à même de prouver la validité de nos systèmes. Pour cela, nous avons utilisé le cadre du typage non standard, plus précisément du système d'effets, où les déclarations ne concernent pas que les types des expressions mais aussi diverses propriétés comme les effets de bord (le langage *FX*), le flot de contrôle ou, dans notre cas, les temps d'exécution.

Les langages, support de l'analyse de temps, autorisent les fonctions d'ordre supérieur et les effets de bord. Les systèmes d'inférence et de vérification de type associés fournissent, par une voie nouvelle, des informations de temps sur les programmes. Les déclarations de temps utilisent l'ensemble des entiers (qui bornent statiquement le temps d'exécution) augmenté d'une valeur spéciale représentant la récursion. Le premier système de typage (chapitre 3) est un reconstructeur pour un langage implicitement typé. Les deuxième et troisième (chapitre 4) sont respectivement un vérificateur pour un langage totalement explicite et un reconstructeur partiel.

La simplicité des informations de temps obtenues est à considérer à la vue des points suivants. D'abord, nos systèmes sont totalement automatiques et les langages analysés sont réalistes (d'un point de vue programmation). Ensuite, l'obtention de résultats plus précis nécessite l'introduction (ou la reconstruction) des *mesures*[5]. Cette introduction ne doit pas empêcher les déclarations de rester simples et aisément compréhensibles pour le programmeur. Enfin, tous les systèmes présentés sont prouvé cohérents avec un modèle standard d'exécution et les algorithmes proposés sont prouvé corrects.

Parmi les extensions possibles de ces systèmes de typage, la première venant à l'esprit est d'augmenter le langage avec le test (`if`). Celui-ci est direct à introduire puisqu'il peut être

[5] Paramètres significatifs sur lesquels on exprime les coûts.

exprimé par expansion de macro-expression en imbrications d'abstractions. Il ne nécessite donc pas de modification des systèmes de typage mais implique un temps global du test égale à la somme des temps de l'expression booléenne et des deux branches. Cela n'est gênant que si l'on désire avoir une borne entière précise et non plus une taxonomie récursive/non récursive. Dans ce cas, on introduit un opérateur de maximisation des temps. Le temps d'un test devient la somme du temps de l'expression booléenne et du maximum des temps des deux branches. L'algèbre des temps est alors plus complexe. Elle doit toujours tenir compte des propriétés ACZ de la sommation, mais aussi des propriétés ACUIZ (ACZ avec idempotence et élément unitaire) de la maximisation. Pour l'inférence, la méthode de résolution présentée au chapitre 3 reste valable et le prédicat d'équivalence des temps utilisé, chapitre 4, pour la vérification s'étend simplement. Par contre la reconstruction partielle suppose un algorithme d'unification sur cette nouvelle algèbre. La deuxième extension simple consiste à ajouter au langage les déclarations locales (construction `let`). Dans le cadre du typage explicite, cette construction est déjà inclue dans le noyau applicatif et ne demande aucun travail. Pour l'implicite, elle introduit aisément un polymorphisme à la Standard ML.

Une autre voie d'extension est de chercher à enrichir le domaine des temps pour obtenir une information plus riche sur le déroulement du programme. L'idée immédiate est un domaine des temps à trois niveaux : un niveau pour les temps constants ($\not\sim$ `long` dans le treillis actuel), un pour des expressions symboliques exprimant un caractère polynomial, logarithmique ou exponentiel du programme et un troisième niveau pour exprimer l'absence d'information de temps. Cette vision pose un problème difficile à résoudre concernant la notion de mesure avec laquelle seront construites les expressions symbolique. En effet, la mesure symbolisant le critère significatif de la notion de coût n'est pas simple à formaliser ou à manipuler automatiquement. Il apparaît qu'en fait, une information indiquant quelles structures de données récursives ayant induit la récursion du programme serait plus facile à obtenir et pourrait constituer un premier pas vers la manipulation de mesures.

La constructions des systèmes présentés nous a permis d'appréhender l'intérêt et la puissance du typage non standard. Nous espérons continuer dans cette voie et proposer des systèmes d'analyse plus puissant. La notion de *mesure* peut être abstraite, par exemple, par l'ensemble des structures de données récursives utilisées lors de la récursion et permettre d'obtenir des résultats plus précis. La taille mémoire utile à l'exécution d'une expression ou au stockage des données pourra être obtenue par une méthode analogue. Le lancement en parallèle d'une tâche pourrait ainsi ne plus être uniquement pondéré par le temps d'exécution mais aussi par la taille de la mémoire du processeur distant et par le coût de transfert des données.

– – –◇◇◇– ––

Bibliographie

[AS85] Abelson, H., et Sussman, G. J., *Structure and Interpretation of Computer Programs. The MIT Press, McGraw-Hill Book Company* (1985), 293-324.

[AH87] Abramsky, S., et Hankin, C., *Abstract Interpretation of Declarative Languages. Ellis Horwood Limited, Chichester, England* (1987).

[A90] Apt, K. R., *Elsevier, Handbook of TCS, Vol B, Formal Models and Semantics, Chp 10 : Logic Programming, (1990) 516-518*

[B78] Backus, J. W., Can Programming be Liberated from Von Neumann Style? A Functionnal Style and its Algebra of Programs. *CACM 21,* 8 (Aout 1978), 613-641.

[B81] Barendregt, H. P., *The lambda-Calculus. Its Syntax and Semantics. North-Holland Publishing Company,* (1981).

[BH90] Barendregt, H. P., et Hemerik K., Types in Lambda Calculi and Programming Languages. *ESOP'90, LNCS,* (1990).

[B80] Beckman, F. S., *Mathematical Foundations of Programming. Addison-Wesley Publishing Company Inc.* (1980).

[B85] Boehm, H. J., Partial Polymorphic Type Inference is Undecidable. *Proceedings du 26eme FOCS Symposium, IEEE,* (1985).

[CGGW85] Char, B. W., Geddes, K. O., Gonnet, G. H. et Watt, S. M., MAPLE: Reference Manual. *University of Waterloo,* (1985).

[CJ91] Consel C. et Jouvelot P., Communication privée. (1991).

[CC77] Cousot, P. et Cousot, R., Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *ACM PoPL, Los Angeles,* (January 1977), 238-252.

[Deu89] Deutsch, A., Génération automatique d'interpréteurs et compilation à partir de définitions dénotationnelles. *Mémoire de DEA, Université Paris 6, LITP 89-17 RXF,* (Janvier 1989).

[Dew89] Dewdney, A. K., *The Turing Omnibus. Computer Science Press, 1803 Research Boulevard, Rockville, MD 20850,* (1989).

[D91] Dornic, V., Vérification des types, tailles et temps dans un langage applicatif typé polymorphe. *Journées Francophones des Langages Applicatifs, JFLA'91, BIGRE 72,* (Janvier 1991).

[DJG91] Dornic, V., Jouvelot, P. et Gifford, D. K., Polymorphic Time Systems for Estimating Program Complexity. *JTASPEFL'91, Bordeaux, France,* (Octobre 1991). *A paraître dans ACM LOPLAS,* (1992).

[F83] Fages, F., Formes canoniques dans les algèbres booléennes et applications à la démonstration automatique en logique du premier ordre. *Thèse, LITP, Université Paris VII,* (Novembre 1983).

[FLD83] Fortune, S., Leivant, D., et O'Donnell, M., The expressiveness of Simple and Second-Order Type Structures. *JACM 30,* 1 (Janvier 1983) 151-185.

[FV87] Flajolet, P., et Vitter, J. S., Average-Case Analysis of Algorithms and Data Structures. *Rapport de recherche INRIA 718,* (Aout 1987).

[FSZ88] Flajolet, P., Salvy, B. et Zimmermann, P., Lambda-Upsilon-Omega : An Assistant Algorithms Analyser. *Rapport de recherche INRIA 876 et Proceedings of AECC'6, Lect. Notes in Computer Science,* (Juillet 1988).

[F88] Fradet, P., Compilation des langages fonctionnels par transformation de programmes. *Thèse, Université de Rennes I,* (Novembre 1988).

[GL86] Gifford, D. K., Integrating Functional and Imperative Programming. *ACM Conference on Lisp and Functional Programming,* (Aout 1986) 28-38.

[GJLS87] Gifford, D. K., Jouvelot, P., Lucassen, J. M., and Sheldon, M. A., *The FX-87 Reference Manual. Massachusetts Institute of Technology, LCS/TR-407,* (1987).

[G72] Girard, J-Y., Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur. *Thèse, Paris,* (1972).

[GLT89] Girard, J-Y., Lafont, Y. et Taylor, P., *Proofs and Types. Cambridge Tracts in Theoretical Computer Science,* 7 (1989), 114-119.

[GSS90] Girard, J-Y., Scedrov A., et Scott P. J., Bounded Linear Logic: A Modular Approch to Polynomial Time Computability. *Feasible Mathematics Proceedings, Math. Sci. Institute Workshop, Cornell Univ., Buss S. R. et Scott P. J. ed., Birkhauser,* (Juin 1990).

[G83] Gray, S. L., Using Futures to Exploit Parallelism in Lisp. *MIT SB Master Thesis,* (1983).

[G88] Goldberg F. B., *Multiprocessor Execution of Functional Programs. Research Report YALEU/DCS/RR-618,* (April 1988).

[HMT89] Harper, R., Milner, R. et Tofte, M., The definition of Standard ML. Version 3. *LFCS Report, DCS, University of Edinburgh,* (Mai 1989).

[HC88] Hickey, T., et Cohen, J., Automating Program Analysis. *JACM 35,* 1 (Janvier 1988) 185-220.

[HU79] Hopcroft, J. E. et Ullman, J. D., *Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Publishing Company Inc.* (1979).

[JD89] Jouvelot, P., et Dornic, V., FX-87, or What Comes After Scheme. *BIGRE 65,* (Juillet 1989), 55-65.

[JG89] Jouvelot, P., et Gifford, D. K., Reasoning about Continuations with Control Effects. *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation,* ACM, New York, (1989).

[JG91] Jouvelot, P. et Gifford, D. K., Algebraic Reconstruction of Types and Effects. *Proceedings of the PoPL'91 Conference, Orlando Fl.,* (Janvier 1991) 303-310.

[J92] Jouvelot, P., Communication privée. (1992).

[KN86] Kapur, et Narendran, NP-Completeness of the Set Unification and Matching Problems. *Proc. od the 8th International Conference on Automated Deduction, Springer LNCS 230,* (Juillet 1986) 489-495.

[Kn73] Knuth, D. E., *The Art of Computer Programming. Vol 1 : Fundamental Algorithm. Addison-Wesley, Reading Mass* (1973).

[Kn81] Knuth, D. E., *The Art of Computer Programming. Vol 2 : Semi-Numerical Algorithms. Addison-Wesley, Reading Mass* (1981).

[Ko81] Kozen, D., Semantics of Probabilistic Programs. *J. Comput. Syst. Sci. 22* (1981), 328-350.

[KM89] Kuo T. M., et Mishra P., Strictness Analysis: A New Perspective Based on Type Inference. *FPCA'89, Londres, ACM Press* (Septembre 1989) 260-273.

[LP87] Lee P., et Pleban U., A Realistic Compiler Generator Based on High-Level Semantics. *Proceedings of the PoPL'87 Conference, Munich Germany.,* (January 1987) 284-295.

[L85] Le Métayer, D., Mechanical Analysis of Program Complexity. *ACM SIGPLAN 85 Symposium, SIGPLAN Notices, 20,* 7 (July 85) 69-73.

[L87] Le Métayer, D., Analysis of Functional Programs by Program Transformation. *Proceedings of France-Japan Artificial Intelligence and Computer Science Symposium, North-Holland, Amsterdam,* (1987).

[L88] Le Métayer, D., ACE: An Automatic Complexity Evaluator. *TOPLAS ACM 10,* 2 (Avril 1988), 248-266.

[LW90] Leroy X., et Weis P., Polymorphic Type Inference and Assignment. *Rapport de Recherche INRIA, 1327,* (Novembre 1990).

[Lu87] Lucassen, J. M., Types and Effects. Towards the Integration of functional and imperative programming. *MIT/LCS/TR-408,* (Aout 1987).

[LG88] Lucassen, J. M., et Gifford D. K., Polymorphic Effect Systems. *PoPL'88, San Diego, ACM,* (January 1988).

[M63] MacCarthy, J., A Basis for Mathematical Theory of Computation. *Computer Programming and Formal Systems, Braffort and Hirsberg Eds, North-Holland Amsterdam,* (1963).

[M79] McCracken, N. J., An Investigation of a Programming Language with a Polymorphic Type Structure. *PhD Dissertation, Syracuse University,* (1979).

[MS82] MacQueen D. et Sethi R., A Semantic Model of Types for Applicative Languages. *LFP'82, Pittsburgh, Pennsylvania, ACM,* (August 1982) 243-252.

[MPS84] MacQueen D., Plotkin G., et Sethi R., An Ideal Model for Recursive Polymorphic types. *PoPL'84, ACM,* (1984) 165-174.

[M76] Mosses, P. D., Compiler Generation using Denotational Semantics. *Mathematical Foundations of Computer Science,* Springer Verlag, (1976), 436-441.

[OG89] O'Toole, J., et Gifford, D. K., Type Reconstruction with First-Class Polymorphic Values. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, New York,* (1989) 207-217.

[PB85] Purdom, P. W. Jr., et Brown, C. A., *The Analysis of Algorithms. Holt, Rinehart et Winston. CBS Publishing,* (1985).

[R79] Ramshaw, L. H., Formalizing the Analysis of Algorithms. *Rapport SL-79-5, Xerox Palo Alto Research Center, Palo Alto, Calif.* (1979).

[R84] Reynolds J. C., Polymorphism is Not Set-Theoretic. *Kahn, McQueen and Plotkin ed., Semantics of Data Types, Springer LNCS* 173 (1984) 145-156.

[R65] Robinson, J. A., A Machine-oriented Logic Based on the Resolution Principle. *JACM 12,* 1 (1965) 23-41.

[R86] Rosendahl, M., Automatic Program Analysis. *Master's Thesis. Institute of Datalogy, University of Copenhagen,* (1986).

[R89] Rosendahl, M., Automatic Complexity Analysis. *Proceedings de FPCA'89, ACM,* (1989) 144-156.

[SJ87] Saint-James, E., De la méta-recursivité comme outil d'implémentation. *Thèse d'état, Université Paris 6,* (Décembre 1987).

[S91] Salvy, B., Asymptotique automatique et fonctions génératrices. *Rapport de Thèse, Ecole Polytechnique,* (Avril 1991).

[S88] Sands, D., Complexity Analysis for Higher Order Language. *Rapport DOC 88/14, Imperial College, Londre.* (Octobre 1988).

[SG89] Sheldon, M., Static Dependent Types for First Class Modules. *ACM Conference on Lisp and Functional Programming,* (1990) 20-29.

[Si89] Siekmann, J., Unification Theory. *Journal of Symbolic Computation,* 7 (1989), 207-274.

[S75] Stickel, M. E., A Complete Unification Algorithm for Associative-Commutative functions. 4^{th} *International Joint Conference on Artificial Intelligence, Tbilisi,* (1975) *and JACM 28,* 3 (1981), 423-434.

[TJ91a] Talpin, J-P., et Jouvelot, P., On Reconstructing Type, Effect and Region in Polymorphic Functional Languages and its Applications. *Rapport de recherche E-149, Ecole des Mines de Paris, Présenté au workshop CPC'91 sous le titre* Applications of types and effect inference, (Février 1991).

[TJ91b] Talpin, J-P., et Jouvelot, P., Polymorphic Type Region and Effect Inference. *Rapport de recherche E-150, Ecole des Mines de Paris, Accepté à Journal of Functional Programming, Cambrigde Press, Présenté aux JTASPEFL'91,* (Février 1991).

[T88] Tofte M. Operational Semantics and Polymorphic Type Inference. *University of Edinburgh, THESIS CST-52-88,* (1988).

[T90] Tofte M. Type Inference for Polymorphic References. *Academic Press, Information and Computation 89,* 5 (1990) 1-34.

[T36] Turing, A. M., On Computable Numbers, with an Application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society,* ser 2., vol. 42, 230-265; vol. 43, 544-546, (1936).

[W75] Wegbreit, B., Mechanical Program Analysis. *CACM 18,* 9 (Septembre 1975), 528-539

[Z89] Zimmermann, P., ALAS : Un Système d'Analyse Algébrique. *Rapport de Recherche, INRIA 968* (Janvier 1989).

[Z91] Zimmermann, P., Séries génératrices et analyse automatique d'algorithmes. *cd Thèse, Ecole Polytechnique,* (Mars 1991).