Récursion généralisée et inférence de types avec intersection

Pascal Zimmer

Université de Nice – Sophia Antipolis

UFR Sciences **École doctorale STIC**

Récursion généralisée et inférence de types avec intersection

THÈSE

présentée et soutenue publiquement le 29 Avril 2004

pour obtenir le titre de

Docteur en Sciences
de l'université de Nice – Sophia Antipolis
(spécialité informatique)

par

Pascal ZIMMER

Thèse dirigée par Gérard BOUDOL

et préparée à l'INRIA Sophia Antipolis, projet Mimosa

Composition du jury

Président :	M. Charles André
Rapporteurs :	Mme Simonetta Ronchi della Rocca
	M. Pierre-Louis Curien
	M. Pierre Lescanne

Résumé

Dans une première partie, nous définissons un nouveau langage à base fonctionnelle et avec récursion généralisée, en utilisant le système de types avec degrés de Boudol pour éliminer les récursions dangereuses. Ce langage est ensuite étendu par des enregistrements récursifs, puis par des mixins, permettant ainsi de mêler totalement les paradigmes fonctionnels et objets. Nous présentons également une implémentation, MlObj, ainsi que la machine abstraite servant à son exécution.

Dans une deuxième partie, nous présentons un nouvel algorithme d'inférence pour les systèmes de types avec intersection, dans le cadre d'une extension du λ-calcul. Après avoir prouvé sa correction, nous étudions sa généralisation aux références et à la récursion, nous le comparons aux algorithmes d'inférence déjà existants, notamment à celui de Système $\mathbb{I}$, et nous montrons qu'il devient décidable à rang fini.

Mots-clés: λ-calcul, ML, récursion, programmation orientée objet, mixins, types avec intersection, inférence de types, calcul de Klop

Abstract

In the first part, we define a new programming language with a functional core and generalised recursion, by using Boudol's type system with degrees to rule out unsafe recursions. The language is extended first with recursive records, then with mixins, allowing the programmer to fully mix functional and object-oriented paradigms. We also present an implementation, MlObj, and an abstract machine for execution.

In a second part, we design a new inference algorithm for intersection type systems, on an extension of the λ-calculus. After proving its correctness, we study its generalisation to references and recursion, we compare it with existing inference algorithms, mainly System $\mathbb{I}$, and we show that its finite rank version becomes decidable.

Keywords: lambda-calculus, ML, recursion, object-oriented programming, mixins, intersection types, type inference, Klop calculus

Remerciements

Je tiens à remercier les personnes suivantes, sans qui ce travail ne serait pas ce qu'il est :
– mon directeur de thèse, Gérard Boudol, dont les idées sont à l'origine de ces pages et qui m'a fait partager ses connaissances inépuisables en matière d'étude formelle des langages de programmation ;
– Davide Sangiorgi, qui m'a également supervisé pendant les premiers temps, alors que je travaillais sur les ambients ;
– Simona Ronchi della Rocca, Pierre-Louis Curien et Pierre Lescanne, qui ont gentiment accepté d'être les rapporteurs de ma thèse, et dont les commentaires ont contribué à son amélioration ;
– Charles André, qui a accepté de présider le jury de soutenance ;
– tous les chercheurs du projet Mimosa, ainsi que ceux de Tick et du CMA, pour leurs encouragements et leurs commentaires réguliers au fil des ans ;
– plus généralement, tous les habitants de Fermat-RDC, pour la bonne ambiance de travail qu'on y trouve et pour leurs nombreuses discussions aussi diverses que passionnées ;
– ma famille, mes amis, tous ceux qui m'ont apporté leur aide au cours de cette aventure, et notamment Audrey, qui s'est attachée à me soutenir durant les derniers mois de rédaction.

Table des matières

Introduction

La conception de nouveaux langages et paradigmes de programmation sont des sujets de recherche actifs depuis les débuts de l'informatique. Plus particulièrement, nous nous intéressons à la formalisation des concepts qui sous-tendent ces langages, afin qu'ils proposent davantage d'expressivité, de sûreté d'exécution et/ou de systèmes d'analyse statique (comme les systèmes de types).

Notre but est d'aboutir à un langage de base possédant les caractéristiques suivantes :

- opérationnellement, un cœur fonctionnel enrichi d'un calcul d'objets, muni d'une sémantique précise et implémentable de façon réaliste ;
- que ce langage possède un système de typage à la ML [DM82, Mil78], autrement dit un système de types pour lequel il existe un algorithme d'inférence totale de types principaux.

A notre connaissance, peu de langages actuels remplissent toutes ces conditions ; le langage OCaml [LDG+00, RV98] en est une exception, mais comme nous le verrons dans la suite, il n'est pas complètement satisfaisant. Cependant, on peut affirmer que notre proposition de langage et d'implémentation s'inspireront largement d'OCaml.

Examinons nos conditions ci-dessus. Si la notion de typage à la ML est maintenant bien établie, celle d'objet, bien qu'implémentée et utilisée dans de nombreux langages généralistes, prête encore à débat parmi la communauté scientifique. De notre point de vue, un objet consiste en l'encapsulation de données en un *état* et de *méthodes* pour y accéder et les modifier. De plus, un mécanisme d'*héritage* doit permettre de créer de nouveaux objets à partir d'autres par ajout ou redéfinition de méthodes. Ce dernier point n'est toujours pas clairement fixé (voir [Tai96] par exemple) et varie fortement d'un langage à un autre. Une conséquence est qu'un grand nombre de modèles théoriques pour la programmation orientée objet, celui d'OCaml y compris, proposent un calcul spécifique [AC96, Wan87, FHM93, RV98]. A l'inverse, nous pensons qu'il vaut mieux partir de bases plus simples et préexistantes (en l'occurrence ML), et les utiliser pour construire une couche objet. Parmi les bénéfices d'une telle approche, on peut citer :

- l'existence de systèmes de types simples et puissants, reconnus et étudiés, ainsi que d'algorithmes d'inférence de types principaux ;
- au niveau opérationnel, un seul mécanisme de calcul, au lieu d'avoir une réduction spécifique pour la partie objet ;
- les objets et les classes étant des valeurs de premier ordre, elles bénéficient de la pleine fonctionnalité (autrement dit, tout comme ML permet de programmer naturellement des fonctions d'ordre supérieur, il est ici tout aussi naturel de créer des constructions d'objets et de classes d'ordre supérieur).

Précisons enfin que le but ultime d'un tel langage est de servir de base à l'ajout d'autres paradigmes de programmation, notamment la mobilité et la programmation réactive (sujets que nous n'aborderons pas dans cette thèse).

Formalisation des langages orientés objets

On peut grossièrement classifier les modèles orientés objets suivant deux types de sémantique :

la sémantique par enregistrement récursif :

Dans ce modèle, initié par Cardelli [Car88, FM95], une *classe* est une fonction paramétrée par des *variables d'instance* et par un paramètre `self`, qui retourne un enregistrement extensible de *méthodes*. Un *objet* est alors défini comme le *point fixe* d'une classe instanciée par ses paramètres d'instance, autrement dit il s'agit d'un enregistrement récursif dans lequel la variable `self` est lié à l'enregistrement lui-même. L'appel de méthode revient alors à sélectionner simplement le champ correspondant dans l'enregistrement.

Wand [Wan94] propose un calcul utilisant cette sémantique, basé sur la notion de *variables de rangée* [Wan87], celles-ci permettant l'ajout et la redéfinition de méthodes, c'est-à-dire d'héritage à la SMALLTALK, de façon élégante. Malheureusement, son modèle n'est pas suffisamment expressif pour nos besoins : en effet, il ne permet pas de modifier l'état d'un objet une fois celui-ci créé ! Plus précisément, le paramètre `self` étant lié au moment de la création de l'objet, celui-ci continue à faire référence à l'objet de départ, et donc à l'état initial, et non pas à l'objet au moment de l'appel de la méthode.

Cook [CHC94] a proposé un modèle plus élaboré afin de résoudre ce problème : il permet la mise à jour de l'état en créant tout simplement un nouvel objet, instance de la même classe, avec un état modifié. Dans ce modèle, les classes elles-mêmes sont donc récursives puisqu'elles peuvent être amenées à appeler leur propre constructeur. D'un point de vue opérationnel, cette technique, bien qu'elle engendre de nombreuses créations d'objets, est satisfaisante. Cependant, elle fait appel à un système de typage trop complexe vis-à-vis de nos critères, puisqu'il n'existe pas d'algorithme d'inférence de types principaux pour ce langage. Le même défaut s'applique d'ailleurs à tous les autres modèles basés sur une théorie de types d'ordre supérieur [AC96, Bru93, EST95, FHM93, PT94].

la sémantique par auto-application :

Dans ce modèle, proposé par Kamin [Kam88], un objet est une collection de *pré-méthodes*, une pré-méthode étant une fonction prenant un paramètre `self`. Au contraire de la sémantique par enregistrement récursif, ce dernier n'est lié qu'au moment de l'appel de la méthode. Ainsi, si l'on note $o = [\ldots, l = \zeta(s)b, \ldots]$ un tel objet, l'appel de la méthode l, qui se note $o.l$, est défini comme $b\{s \leftarrow o\}$. Ayant accès à l'objet au moment de l'appel, il est possible de modifier son état ; opérationnellement ce modèle est donc satisfaisant. Cependant, du point de vue du typage, il faut s'assurer que les paramètres `self` de toutes les pré-méthodes d'un objet ont tous le même type. Il n'est donc pas possible d'utiliser les types des enregistrements pour de tels objets et des systèmes spécifiques doivent être construits (ce qui explique que l'on utilise la notation ζ comme lieur pour `self` au lieu de l'habituel λ car aussi bien la réduction que le typage sont spécifiques).

Des calculs d'objets basés sur la sémantique par auto-application, et leurs systèmes de types *ad hoc*, ont été proposés, notamment par Fisher et Mitchell [Fis96, FHM93, FM95] et Abadi et Cardelli [Aba94, AC96]. Cependant, toutes les propositions qui intègrent la possibilité d'étendre les objets par héritage réclament des systèmes de types trop complexes, pour lesquels l'inférence de types principaux n'est pas possible.

Une proposition de langage

Poursuivant l'approche de Wand à partir d'enregistrements récursifs, Boudol [Bou04] reprend l'idée des variables de rangée qui permettent l'héritage de façon élégante. Afin de résoudre le problème lié à la modification de l'état, il propose de remplacer la vision purement fonctionnelle de Wand par un mécanisme de modification impératif [BPSM99b, EST95, RV98], basé sur un langage ML avec *références*. Si le paramètre `self` est toujours lié au moment de la création de l'objet, son état est stocké dans une (ou plusieurs) référence(s), elle(s)-même(s) champ(s) de l'enregistrement récursif. Pour modifier l'état, il suffit alors d'accéder à ces références et de les mettre à jour, sans qu'il soit nécessaire de modifier les champs de l'objet lui-même. Cette proposition de langage, telle que nous la reprenons dans cette thèse, est basée sur le noyau "Reference ML" tel que considéré par Wright et Felleisen [WF94], auquel on ajoute la possibilité de point fixe par `let rec` et des opérateurs sur les enregistrements similaires à ceux de Cardelli et Mitchell [CM94].

Un nouveau problème se pose alors suite à l'ajout du point fixe : si celui-ci permet bien de construire des valeurs récursives *fonctionnelles*, il n'est en principe pas permis de l'utiliser pour construire des valeurs récursives *non fonctionnelles*. Or ceci nous est indispensable dans le cas des enregistrements avec références, qui sont des points fixes de classes instanciées. Dans les langages en appel par valeur simplement typés (par exemple le langage OCaml), seule la construction (`let rec` $x = V$ `in` M) où V est une valeur (essentiellement une fonction) est autorisée. Nous aimerions pouvoir écrire (`let rec` $x = N$ `in` M), où N s'évalue en un enregistrement, éventuellement avec création de nouvelles références, donc d'effets de bord. En particulier, il nous faut pouvoir évaluer (`let rec` $x = G\ x$ `in` M) où G est une classe (instanciée) qui s'évalue en un générateur λ`self` R [CP89]. De façon plus générale, d'autres constructions intéressantes comme les modules mutuellement récursifs [CHP99, HL02] sont susceptibles de bénéficier d'un tel point fixe généralisé.

Comme le fait remarquer Rémy [Rém94], dans le cas des enregistrements récursifs (convenablement formés), le `let rec` tel qu'il existe déjà donne une évaluation correcte ; il n'est pas besoin d'en définir une nouvelle sémantique. Bien entendu, il n'est pas non plus possible de permettre toutes les constructions : si l'évaluation de N réclame une valeur pour x, nous sommes bloqués puisque celle-ci n'est pas encore calculée. Par exemple, l'évaluation de (`let rec` $x = \mathbf{F}\ (x\ V)$ `in` M) où $\mathbf{F}$ est le combinateur $\lambda x \lambda y\ y$ provoque une erreur. Il nous faut donc un mécanisme d'analyse statique capable de détecter quand une variable récursive est *dangereuse* (i.e. quand l'évaluation du terme est susceptible de requérir sa valeur).

Les degrés

Boudol [Bou04] propose un système de types avec *degrés* pour répondre à cette question. Intuitivement, un degré est une information booléenne 0 ou 1, attachée à chaque variable apparaissant dans le contexte de typage et à l'argument d'un type fonction $\theta^d \to \tau$. Le degré 1 s'interprète comme "sûr" (i.e. la valeur de la variable n'est pas requise lors de l'évaluation), tandis que 0 s'interprète comme "possiblement dangereux". Pour qu'une récursion (`let rec` $x = N$ `in` M) soit admissible, il faut et il suffit que N soit typable avec un degré 1 pour x. Ceci est toujours le cas pour les abstractions $N = \lambda y\ N'$, car la récursion est alors "gardée". Une fonction qui a pour type $\theta^1 \to \tau$ est dite *protectrice*, autrement dit c'est une fonction qui protège son argument. Pour que (`let rec` $x = G\ x$ `in` M) soit admissible, il faut et il suffit de vérifier que G est une fonction protectrice.

Dans son article, Boudol montre qu'un terme typable dans le système avec degrés ne peut pas conduire à une erreur d'exécution. Il y définit également un algorithme d'inférence qui retourne un type principal pour tout terme typable.

Contribution

Mon apport personnel a consisté à participer à la mise au point du langage et du système de types avec degrés, par l'implémentation d'un interpréteur pour le langage avec enregistrements défini dans [Bou04], ainsi que de l'algorithme d'inférence de types avec degrés. Il a contribué à grandement faciliter la mise au point du système de types et des nouvelles constructions, en permettant l'expérimentation rapide de nos idées, par aller-retour entre la théorie et la pratique, et s'est avéré très utile puisque les premières versions du système de types n'étaient pas assez puissantes pour typer toutes les constructions d'objets nécessaires à notre langage. Dans sa version finale, cet interpréteur, baptisé MlObj et disponible en ligne, permet l'évaluation de termes et l'inférence statique de leurs types principaux, comme le fait le système Objective Caml. Au-delà de la réécriture d'un mini-ML complet, la difficulté principale fut le développement d'un algorithme d'inférence de types (récursifs) qui soit à la fois efficace et sûr, à partir des méthodes déjà existantes, l'ajout des degrés apportant une complexité supplémentaire.

Dans un deuxième temps, nous avons ajouté une couche syntaxique au langage (et à l'interpréteur), sous la forme d'un calcul de *mixins*, afin d'étendre encore l'expressivité des constructions orientés objets. Celle-ci permet un style de programmation plus puissant, notamment par héritage multiple, ou même de définir des classes non instanciables directement, mais qui peuvent servir à étendre d'autres classes par héritage (des exemples de telles constructions seront donnés au Chapitre 4).

Enfin la question de l'évaluation efficace du point fixe généralisé s'est posée ; en effet, notre calcul d'objets nous amène à devoir évaluer des termes de la forme $(\lambda \texttt{self} R)\ o$, où o est une variable non encore évaluée (sachant que `self` ne sera pas demandé lors de l'évaluation de R), et ceci sans que soit levée une exception, comme c'est le cas dans [Rus01]. Habituellement, les machines abstraites d'évaluation pour le λ-calcul ne considèrent pas les variables comme des valeurs. De plus, alors qu'on pourrait être tenté d'utiliser la technique du *point fixe impératif* de Landin [Lan64], en traduisant le `let rec` par :

$$\begin{array}{rcl} [\![\texttt{let rec}\ x = N\ \texttt{in}\ M]\!] & = & \texttt{let}\ r = \texttt{ref ?} \\ & & \texttt{in}\ (r := \{x \mapsto !r\}\ [\![N]\!]; \texttt{let}\ x = !r\ \texttt{in}\ [\![M]\!]) \end{array}$$

où `?` dénote une valeur spéciale "indéfinie", celle-ci n'est pas toujours valide lorsque N n'est pas une valeur, en particulier lorsque $N = G\ x$. Pour résoudre ce problème, nous avons donc mis au point une nouvelle machine abstraite pour la récursion et nous avons prouvé sa correction vis-à-vis de la sémantique du langage. Cette machine utilise une propriété essentielle, vérifiée par les termes typables, qui affirme que les variables récursives qui n'ont pas encore reçu de valeur peuvent toujours être utilisées et passées en argument d'autres fonctions, sans que jamais leur valeur explicite ne soit requise. Cette machine a également été implémentée dans MlObj en remplacement de la sémantique big-step d'origine du langage.

A l'usage, certaines fonctionnalités avancées que nous souhaitions pouvoir proposer dans le calcul de MlObj se sont révélées impossibles à encoder de façon satisfaisante tout en restant typables, notamment le *clonage d'objets* ou les *méthodes binaires*. Toujours dans le but d'améliorer l'expressivité de notre langage, nous avons cherché s'il était possible de faire mieux sans sacrifier pour autant nos conditions, à savoir l'existence d'un algorithme d'inférence de types principaux. Si le Système $\mathcal{F}$ [Gir86] permettrait peut-être de typer nos constructions en théorie, il est bien évidemment exclu d'y faire appel, puisque l'inférence de types y est indécidable [Wel99]. Une autre possibilité serait d'autoriser les méthodes polymorphes [Dug94] dans les enregistrements, une possibilité qui n'a été ajoutée au système Objective Caml [LDG+00] que très récemment. A nouveau, l'inférence complète de types polymorphes étant indécidable, il faudrait, comme c'est le cas en OCaml, aider l'algorithme d'inférence en annotant explicitement chaque méthode polymorphe avec son type, ce que nous ne souhaitons pas. De plus, quelques tentatives en ce sens ont démontré que ceci ne permettrait pas forcément de typer toutes les constructions souhaitées.

Nous nous sommes alors intéressé à une solution intermédiaire : les *types avec intersection*, qui constituent une forme différente de polymorphisme, mais pour lesquels des résultats récents ont prouvé que l'inférence de types devenait décidable à condition de limiter le rang des types inférés, et pour lesquels il existe une notion claire de typage principal. Néanmoins, les algorithmes trouvés dans la littérature existante se sont révélés difficiles à comprendre, complexes et lourds à mettre en oeuvre. C'est pourquoi nous avons souhaité en donner une version plus simple et concise, en mettant en lumière certains points importants.

Systèmes de types avec intersection

Nous commençons par donner quelques éléments généraux concernant l'inférence de types pour le λ-calcul. Classiquement, quel que soit le système, celle-ci se résume aux étapes suivantes :

- donner un type à chaque expression et sous-expression, en particulier une variable de type pour les variables et les expressions composées ;
- générer un ensemble de contraintes entre les types, reflétant le fait que si une fonction est appliquée à un argument, alors le type de l'argument doit correspondre au type du domaine de la fonction ;
- résoudre ces contraintes.

Le point crucial qui distingue les divers systèmes existants réside dans la façon de donner un type aux abstractions λxM lors de la première étape. Si $t_1, \ldots, t_n$ désignent les variables de types assignées aux diverses occurrences de x dans M, plusieurs possibilités existent pour le type de λxM :

- types simples (monomorphes) : les occurrences de x doivent toutes avoir le même type, on a donc la contrainte que tous les t_i doivent être égaux ;
- sous-typage : chacun des t_i doit être un *sous-type* du type du domaine de λxM ;
- types généralisés (polymorphes) : chacun des t_i doit être une *instance* du type du domaine de λxM ;
- types avec intersection : la séquence $t_1, \ldots, t_n$ est vue comme un type, appelé la *conjonction* des t_i, et le type de λxM a la forme $t_1, \ldots, t_n \to \theta$;

sachant que, suivant le cas, la résolution des contraintes correspondantes est plus ou moins "facile" [1]. C'est la dernière approche, celle des types avec intersection, que nous allons privilégier.

[1] Pour un aperçu et une comparaison des différents systèmes de types obtenus dans chaque cas, on pourra se référer à l'excellent [Bar92].

Les systèmes de types avec intersection, introduits par Coppo, Dezani et Sallé [Sal78, CDCS79, CDC80], et indépendamment par Pottinger [Pot80], permettent de réunir deux types donnés à un même terme en un seul. Ainsi, si M a les types A et B, il aura également le type $A \wedge B$ (noté aussi A, B suivant les auteurs, ou lorsqu'il apparaît à gauche d'une flèche). Par conséquent, si le type d'un terme M est de la forme $A, B \to C$, il faudra exhiber deux typages d'un argument N avec les types A et B pour que l'application $M\ N$ soit bien typée. Ce système de types, baptisé $\mathcal{D}$ dans [Kri90], vérifie les propriétés suivantes :

- La typabilité est équivalente à la normalisation forte [Kri90, Bou03]. Une conséquence immédiate est que la typabilité est indécidable.
- Si un terme est typable, il est possible de construire explicitement un *typage principal* pour ce terme [CDCV80, RV84, Kri90]. Par "principal", il faut entendre ici par un jeu de transformations plus étendues que celui de la simple substitution comme pour le système de Hindley/Milner, la plus importante étant l'*expansion* (i.e. la capacité à remplacer un type par une conjonction de types équivalents).

Revenons au problème de l'inférence de types pour ce système. Une solution évidente consiste à essayer de normaliser le terme puis à typer la forme normale, mais ceci n'est valable que dans le cadre de programmes qui terminent toujours. Ronchi [Roc88] a proposé un mécanisme direct basé sur une unification généralisée, et plus récemment Kfoury et Wells [KW04, Kfo99] en ont proposé une nouvelle version, appelée Système $\mathbb{I}$, basée sur l'ajout de *variables d'expansion* permettant de formaliser plus explicitement le mécanisme d'expansion utilisé. Ils ont ainsi pu montrer la principalité de l'arbre de typage obtenu et, plus important, que l'inférence de types devenait décidable à rang fini [KW99].

Il existe également une autre famille de systèmes de types avec intersection, dont le représentant principal est appelé $\mathcal{D}\Omega$ dans [Kri90]. Ces systèmes permettent de donner comme type à n'importe quel terme la conjonction vide, généralement notée ω ou Ω. Bien entendu, ils ne vérifient plus la propriété de normalisation forte, puisque n'importe quelle expression est typable. Néanmoins, on peut donner une caractérisation des termes typables (par un type non trivial) : ce sont ceux qui admettent une forme normale de tête. Nous mentionnons ce système car nous serons amenés à le rencontrer pour une variante de l'algorithme d'inférence.

Contribution

Nous donnons une nouvelle méthode pour résoudre les contraintes relatives aux types avec intersection dans $\mathcal{D}$, par le biais d'un algorithme d'unification spécifique. Soient $M\ N$ une application, τ le type de N, σ celui de M, et t la variable de type assignée au noeud de l'application lors de la première étape de l'inférence de types. La contrainte que nous devons résoudre est alors :

$$\tau \to t \ = \ \sigma$$

Etant donné que ces contraintes ont une forme particulière et qu'elles sont non commutatives, nous remplacerons le symbole d'égalité par $\stackrel{\perp}{=}$. Lorsque M est une fonction $\lambda x M'$, autrement dit lorsqu'on a affaire à un *redex*, le type σ prend la forme $t_1, \ldots, t_n \to \theta$ où les t_i sont les variables de types assignées aux occurrences de x dans M' et θ est le type de M'. La contrainte s'écrit alors :

$$\tau \to t \ \stackrel{\perp}{=} \ t_1, \ldots, t_n \to \theta$$

Alors que la β-réduction nous amène à transformer le redex $(\lambda x M'\ N)$ en $M'\{x \mapsto N\}$, l'algorithme d'unification va reproduire ce comportement en unifiant t avec θ, en créant n copies distinctes des contraintes associées à l'argument N et en unifiant chaque t_i avec la copie correspondante de τ.

Dans le cas où $n = 0$, c'est-à-dire dans le cas d'un β_K-redex $(\lambda x M'\ N)$ pour lequel x n'apparaît pas libre dans M', nous devons procéder différemment, car sinon nous serions amenés à ne pas vérifier que l'argument N est bien typable (en d'autres termes, nous serions en train de typer dans $\mathcal{D}\Omega$ au lieu de $\mathcal{D}$). Dans le cas où $n = 0$ donc, nous unifions simplement t avec θ, mais nous gardons les contraintes associées à l'argument N. Nous nous démarquons donc de la β-réduction qui ici "oublie" N.

Comme nous venons de le faire informellement, nous montrerons de façon précise qu'il existe une correspondance exacte entre les étapes de résolution de notre algorithme d'unification et la réduction non pas dans le λ-calcul, mais dans un λ-*calcul étendu*, le $\Lambda_\mathcal{K}$-calcul, introduit dans [Bou03] et basé sur une idée de Klop [Klo80], qui n'efface pas les arguments des β_K-redex. A notre connaissance, c'est la première fois que ce calcul est mis en lumière comme le calcul sous-jacent au problème de l'inférence de types avec intersection. De plus, nous montrerons ensuite que notre algorithme termine si et seulement si le terme analysé est typable dans un système équivalent à $\mathcal{D}$.

Par ailleurs, l'algorithme d'unification des contraintes conduit naturellement à proposer un algorithme de typage, retournant un *arbre de typage* pour le terme lorsque celui-ci est typable. Nous montrerons que cet arbre est bien un arbre de typage valide dans le système de types, et que de plus il fournit un typage *principal* pour le terme. Cet algorithme a été implémenté dans un interpréteur, baptisé TypI, également disponible en ligne ; celui-ci permet de suivre et de diriger pas à pas l'exécution de l'algorithme de typage pour n'importe quel terme du $\Lambda_\mathcal{K}$-calcul, et d'en obtenir l'arbre de typage.

Dans un deuxième temps, nous explicitons le parallèle entre l'algorithme pour le Système $\mathbb{I}$ de Kfoury et Wells et le nôtre. Nous montrons qu'ils effectuent en réalité les mêmes étapes de calcul, et que les variables d'expansion introduites par ces derniers ne sont donc pas nécessaires à l'inférence. En revanche, nous utilisons ce parallèle pour reprendre deux résultats que nous n'avons pas démontrés directement, à savoir la principalité de l'arbre de typage inféré et surtout la décidabilité du typage à rang fini. En conséquence, même si notre algorithme nous paraît plus simple à comprendre, il n'est intrinsèquement ni plus ni moins complexe que celui de Kfoury et Wells. En fait, comme le montre [NM03], l'algorithme d'inférence du Système $\mathbb{I}$ est intrinsèquement aussi complexe que la normalisation forte, ce que nous avons établi par la correspondance entre résolution des contraintes de typage et réduction dans le $\Lambda_\mathcal{K}$-calcul, sans avoir besoin de recourir aux graphes partagés ni aux réseaux de preuves, comme c'est le cas dans [NM03].

Alors que la rédaction de cette thèse s'achevait, Carlier et Wells [CW04] ont proposé un résultat similaire au nôtre, dans le cadre du Système E [CPWK04], un descendant direct de Système $\mathbb{I}$, toujours basé sur les variables d'expansion. Ils y montrent une correspondance opérationnelle entre la résolution des contraintes et la β-réduction. Cependant, on pourra noter un certain nombre de différences : ils utilisent cette fois-ci un système de types à la $\mathcal{D}\Omega$ et ne montrent une correspondance que pour la stratégie de réduction la plus à gauche, là où nous autorisons n'importe quelle stratégie. Ceci est cohérent : utilisant toujours le λ-calcul et non le $\Lambda_\mathcal{K}$-calcul, ils doivent se référer à $\mathcal{D}\Omega$ pour rendre compte de l'oubli des termes pour les β_K-redex. Et, ce faisant, ils perdent la propriété de normalisation forte, qui est remplacée par la caractérisation par les formes normales de tête, d'où l'obligation d'imposer une stratégie de réduction la plus à gauche possible...

Enfin, en prévision des futurs développements à venir sur notre langage, nous nous intéressons dans la dernière partie de cette thèse à l'extension de notre algorithme d'inférence lorsque les références ou la récursion sont ajoutées au $\Lambda_\mathcal{K}$-calcul, introduisant ainsi de la non-terminaison (typable !) dans le langage. Dans le premier cas, tout comme l'ajout des références en ML conduit à restreindre la règle d'introduction du polymorphisme du `let`, Davies et Pfenning [DP00] ont montré que l'introduction de la conjonction devait être limitée aux valeurs. Ceci revient à ne pas effectuer

certaines duplications dans l'algorithme d'inférence ; nous montrerons comment transformer celui-ci afin de vérifier les conditions adéquates. Cela nous amènera également à imposer un ordre de résolution des contraintes, correspondant grossièrement à une β-réduction en appel par valeur, alors que jusqu'ici elles pouvaient être résolues dans un ordre quelconque. Concernant la récursion, nous montrons comment il est possible de typer un opérateur de point fixe $\mu x\ M$ (celui de MlObj) par l'intermédiaire d'une dernière étape d'unification simple après l'algorithme principal.

Plan

La thèse s'articule autour de deux parties principales : la première traite des sujets relatifs à la récursion généralisée et à MlObj, la deuxième des types avec intersection. Au Chapitre 1, on définit le langage d'objets à base d'enregistrements récursifs, ainsi que le système de types avec degrés et les résultats théoriques qui l'accompagnent. Le Chapitre 2 décrit les algorithmes d'unification et d'inférence utilisés dans MlObj de façon pratique. Le Chapitre 3 présente l'interpréteur MlObj proprement dit, ainsi que de nombreux exemples d'interaction. Au Chapitre 4, on décrit le paradigme de programmation objet à base de mixins, d'abord de manière syntaxique, puis sur des exemples en MlObj. Enfin, la machine abstraite pour la récursion généralisée ainsi que ses preuves de correction sont présentées au Chapitre 5.

Le Chapitre 6 présente le $\Lambda_{\mathcal{K}}$-calcul, issu du calcul de Klop, le système de types avec intersection et les résultats théoriques afférents. Le Chapitre 7 constitue le cœur de cette partie puisqu'il donne toutes les définitions relatives à l'algorithme d'inférence proprement dit, ainsi que de nombreux exemples. Les résultats de correction de cet algorithme ainsi que leurs preuves sont énoncés au Chapitre 8. Le Chapitre 9 présente l'implémentation de l'algorithme, TypI. Le Chapitre 10 traite de la variante de l'algorithme correspondant au système de types $\mathcal{D}\Omega$. Le lien avec le Système $\mathbb{I}$ de Kfoury et Wells est traité au Chapitre 11, et les nouveaux résultats théoriques obtenus sont énoncés. Enfin, les Chapitres 12 et 13 traitent respectivement de l'ajout des références et de la récursion à l'algorithme d'inférence.

En Annexe A se trouve un mini-manuel de référence du langage MlObj et de l'interpréteur. A titre d'information, l'Annexe B présente brièvement quelques autres travaux effectués pendant la période de thèse, relatifs au formalisme des ambients. La bibliographie est donnée en page 169.

Première partie

Récursion généralisée

Chapitre 1

Un langage à objets avec récursion généralisée

Ce chapitre est essentiellement un résumé du travail décrit dans [Bou04]. Nous y décrirons un langage d'objets à base d'enregistrements récursifs, ainsi que le système de types avec degrés qui l'accompagnent ; ceux-ci nous accompagneront tout au long de la première partie.

1.1 Le calcul

Nous commençons par définir la syntaxe et la sémantique du langage.

1.1.1 Syntaxe

On suppose donnés trois ensembles distincts de *variables*, notées $x, y, z...$, de *labels*, notés $l, l'...$, et d'adresses mémoire, notées $u, v, w....$ La syntaxe du langage est alors définie par les règles grammaticales suivantes :

M, N	::=	V	valeur
	|	(MN)	application
	|	**let** $x = M$ **in** N	liaison locale
	|	**let rec** $x = M$ **in** N	liaison locale récursive
	|	$\{M, l = N\}$	extension d'enregistrement
	|	$(M.l)$	sélection
	|	$(M \setminus l)$	restriction
V	::=	x	variable
	|	**ref**	création d'une référence
	|	u	adresse mémoire
	|	**!**	déréférencement
	|	**set**	assignation
	|	(**set** V)	assignation bis
	|	$\lambda x M$	fonction
	|	()	unit
	|	R	enregistrement

$$\begin{array}{rcll} R & ::= & x & \text{variable d'enregistrement} \\ & | & \{\} & \text{enregistrement vide} \\ & | & \{R, l = V\} & \text{enregistrement composé} \end{array}$$

Ce langage se compose :

- d'un cœur fonctionnel à la ML, autrement dit de l'application (MN), des fonctions $\lambda x M$, des liaisons locales introduites par **let** et des liaisons locales récursives introduites par **let rec** ;
- de références ou adresses mémoire, et des constructions pour les créer, déréférencer et assigner : **ref**, **!** et **set** ;
- d'*enregistrements*. Ce sont des suites (éventuellement vides) de champs introduits par des labels, du type $l = V$, séparés par des virgules et placés entre accolades. En tête, peut éventuellement se trouver une variable dite *variable de rangée* (on parle alors d'un *enregistrement ouvert*, sinon il s'agit d'un *enregistrement clos*). La construction $\{M, l = N\}$ permet d'étendre un enregistrement avec un nouveau champ l ; $(M.l)$ permet d'obtenir la valeur du champ l, et $(M \setminus l)$ de supprimer le champ l dans M.

Comme c'est l'usage, on notera $\lambda x_1 \dots x_n.M$ à la place de $\lambda x_1 \dots \lambda x_n M$, et $M N_1 \cdots N_k$ à la place de $(\cdots (M N_1) \cdots N_k)$. On notera $fv(M)$ et $bv(M)$ les ensembles de variables libres et liées dans M respectivement. Dans toute la suite, on identifiera les termes obtenus par α-conversion sur les variables des fonctions. La substitution sans capture d'une variable libre x par N dans M sera notée $\{^N/_x\}(M)$.

Concernant les enregistrements, on notera $\{l_1 = M_1, \dots, l_n = M_n\}$ à la place de $\{\dots\{\{\}, l_1 = M_1\} \dots, l_n = M_n\}$, et $\{N, l_1 = M_1, \dots, l_n = M_n\}$ à la place de $\{\dots\{N, l_1 = M_1\} \dots, l_n = M_n\}$. L'opération de modification de champ $\{M, l \leftarrow N\}$ peut être définie comme une abréviation pour $\{M \setminus l, l = N\}$. De même, il est aussi possible de définir une opération de renommage $M[l \leftarrow l']$ par $(\textbf{let } x = M \textbf{ in } \{x \setminus l, l' = x.l\})$.

Enfin, la définition suivante nous sera utile pour le système de types :

Définition 1.1 *On appelle* expressions pures *le sous-ensemble de termes définis par la grammaire suivante (intuitivement, des termes dont l'évaluation ne provoque pas d'effet de bord) :*

$$U ::= x \mid \lambda x M \mid \textbf{\textit{let}}\ x = U\ \textbf{\textit{in}}\ U' \mid \textbf{\textit{let rec}}\ x = U\ \textbf{\textit{in}}\ U' \mid \{U, l = U'\} \mid (U.l) \mid (U \setminus l)$$

1.1.2 Sémantique opérationnelle

La sémantique de ce langage est donnée par des règles de sémantique opérationnelle. Tout d'abord, la grammaire des *contextes d'évaluation*, donnée ci-dessous, définit les points où l'évaluation peut avoir lieu (on notera qu'il s'agit donc d'une sémantique en appel par valeur, avec évaluation gauche-droite) :

$$\begin{array}{rcl} \mathbf{E} & ::= & [] \mid (\mathbf{E} N) \mid (V \mathbf{E}) \mid \textbf{let } x = \mathbf{E} \textbf{ in } M \mid \textbf{let rec } x = \mathbf{E} \textbf{ in } M \\ & | & \{\mathbf{E}, l = M\} \mid \{R, l = \mathbf{E}\} \mid (\mathbf{E}.l) \mid (\mathbf{E} \setminus l) \end{array}$$

On notera *capt*$(\mathbf{E})$ l'ensemble des variables capturées par $\mathbf{E}$, autrement dit celles liées dans $\mathbf{E}$ par un **let rec** qui a [] dans sa portée.

Il est alors possible de définir une relation de réduction *locale* $M \to M'$ par les règles suivantes :

$$\begin{array}{rcll}
(\lambda x M V) & \to & \{^V/_x\}(M) & \\
\textbf{let } x = V \textbf{ in } M & \to & \{^V/_x\}(M) & \\
\textbf{let rec } x = V \textbf{ in } M & \to & \{^{\textbf{let rec } x=V \textbf{ in } V}/_x\}(M) & \\
(\{R, l = V\}.l) & \to & V & \\
(\{R, l = V\}.l') & \to & R.l' & l \neq l' \\
(\{R, l = V\} \setminus l) & \to & R & \\
(\{R, l = V\} \setminus l') & \to & \{(R \setminus l'), l = V\} & l \neq l' \\
M \to M' & \Longrightarrow & \mathbf{E}[M] \to \mathbf{E}[M'] &
\end{array}$$

On notera qu'on effectue ici syntaxiquement les substitutions, aussi bien pour la β-réduction que pour les liaisons locales introduites par les **let**. Dans une implémentation réelle (et donc dans la machine abstraite pour MLOBJ présentée au Chapitre 5), on préférera une sémantique par environnements et clôtures.

Enfin, afin de traiter le caractère impératif du langage, nous enrichissons le langage par une mémoire S, qui est une application des adresses mémoire vers les valeurs :

$$S ::= \varepsilon \mid u := V; S$$

La valeur $S(u)$ d'une adresse pour une mémoire donnée et la fonction $\{u := V\}S$ de mise à jour sont définies de la façon usuelle. On appelle *configuration* et on note $[S \mid M]$ la paire formée d'une expression M et d'une mémoire S. La relation de réduction *globale* entre configurations est alors donnée par les règles suivantes :

$$\begin{array}{rcll}
M \to M' & \Longrightarrow & [S \mid M] \to [S \mid M'] & \\
[S \mid \mathbf{E}[(\textbf{ref } V)]] & \to & [u := V; S \mid \mathbf{E}[u]] & u \text{ frais, } fv(V) \cap capt(\mathbf{E}) = \emptyset \\
[S \mid \mathbf{E}[(\,\textbf{!}u)]] & \to & [S \mid \mathbf{E}[V]] & S(u) = V, fv(V) \cap capt(\mathbf{E}) = \emptyset \\
[S \mid \mathbf{E}[((\textbf{set } u)V)]] & \to & [\{u := V\}S \mid \mathbf{E}[()]] & fv(V) \cap capt(\mathbf{E}) = \emptyset
\end{array}$$

1.2 Le système de types

Nous commençons par décrire la syntaxe des types.

Degrés Dans la suite, les types seront décorés par des degrés. Leur valeur peut être 0 (la variable ou l'argument n'est pas protégé), ou 1 (la variable ou l'argument est protégé), ou encore par une variable de degré $p, q \ldots$. On pose $0 \leqslant 1$ (cette relation d'ordre sera étendue dans la suite). Nous aurons également à considérer des conjonctions de degrés, notées $\alpha, \ldots$.

$$\begin{array}{rcl}
a, b & ::= & 0 \mid 1 \mid p \\
\alpha, \beta & ::= & a \mid \alpha \wedge \beta
\end{array}$$

Types fonctionnels Les types fonctionnels sont similaires à ceux d'un langage à la ML, c'est-à-dire d'un noyau fonctionnel auquel on aurait ajouté des références et des enregistrements dans notre cas. **unit** désigne le type de retour d'une procédure ; t est une variable de type ; $\theta^a \to \tau$ est le type d'une fonction dont l'argument est θ, qui renvoie τ et qui protège (ou non) son argument suivant les indications fournies par le degré a ; τ **ref** est le type d'une référence de type τ ; et ρ désigne le type d'un enregistrement.

$$\tau, \theta \ ::= \ \textbf{unit} \mid t \mid (\theta^a \to \tau) \mid \tau \ \textbf{ref} \mid \rho$$

En anticipant sur la suite, nous pouvons donner quelques exemples de types avec degrés :

- $I = \lambda x\ x$ a pour type $\tau^0 \to \tau$ pour tout type τ, car l'argument x n'est pas protégé dans le corps de la fonction ;
- $K = \lambda x \lambda y\ x$ a pour type $\tau^p \to \sigma^q \to \tau$, car le premier argument x est protégé dans $\lambda y\ x$ et le deuxième argument y n'est pas utilisé ;
- A l'inverse, $F = \lambda x \lambda y\ y$ a pour type $\tau^p \to \sigma^0 \to \sigma$, car y n'est cette fois-ci pas protégé (mais x l'est toujours) ;
- La fonction $\lambda f(\textbf{let rec}\ x = f\ x\ \textbf{in}\ x)$ calcule le point fixe de la fonction f qui lui est passée en paramètre. Son type principal est $(\tau^1 \to \tau)^0 \to \tau$. On constate que f doit impérativement être une fonction protectrice : en effet, si ce n'était pas le cas, il serait impossible de calculer le point fixe $x = f\ x$. Et, bien évidemment, la fonction f elle-même n'est pas protégée.

Types enregistrements Un type enregistrement est la construction imbriquée de champs étiquetés par l accompagnés par le type de leur contenu. Le type de tête peut être soit le type d'un enregistrement vide $\{\}$ (auquel cas le type est clos), soit une variable de rangée t (auquel cas le type est ouvert).

$$\rho\ ::=\ t \mid \{\} \mid \{\rho, l : \tau\}$$

Comme pour les termes, on écrira $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ pour $\{\ldots\{\{\}, l_1 : \tau_1\}\ldots, l_n : \tau_n\}$, et $\{t, l_1 : \tau_1, \ldots, l_n : \tau_n\}$ pour $\{\ldots\{t, l_1 : \tau_1\}\ldots, l_n : \tau_n\}$. On identifiera également les types obtenus par réarrangement de l'ordre des champs, autrement dit les types sont considérés modulo la relation d'équivalence donnée par :

$$\{\{\rho, l : \tau\}, l' : \tau'\} = \{\{\rho, l' : \tau'\}, l : \tau\}$$

Schémas de types Afin d'avoir du polymorphisme dans notre langage, nous suivons la technique utilisée pour le système Robin-Milner. Un schéma de type est obtenu par abstraction de certaines variables de type et de rangée, qui sont alors dites *polymorphes* (par opposition, les autres sont dites *monomorphes*) [2]. De plus, nous allons décorer ces variables par une liste finie de labels L, qui représente la liste des champs non autorisés s'il s'agit d'une variable de rangée (pour une variable de type, L sera toujours $\emptyset$). Intuitivement, si $t :: L$ désigne une variable de rangée, nous n'aurons pas le droit de substituer à t un type enregistrement contenant un champ dont le label est dans L.

$$\begin{array}{rcl} \sigma & ::= & \tau \mid (\forall Q.\sigma) \\ Q & ::= & t :: L \mid t :: L, Q \end{array}$$

Nous identifierons également les schémas de type obtenus par α-conversion des variables ainsi liées.

Types bien formés Dans ce qui précède, rien n'empêche un type enregistrement de contenir plusieurs champs de même label comme dans $\{\{\rho, l\ :\ \tau\}, l\ :\ \tau'\}$. Afin d'exclure ces cas, nous donnons ci-dessous les règles d'un prédicat $C \vdash \tau :: L$ (et $C \vdash \sigma :: L$) qui assure que le type τ est bien formé par rapport à un ensemble d'hypothèses C, et que si τ est un type enregistrement il ne contient pas de champ dont le label est dans L.

$$\frac{}{t :: L, C \vdash t :: L'}\ L' \subseteq L \qquad \frac{}{C \vdash \textbf{unit} :: \emptyset} \qquad \frac{C \vdash \theta :: \emptyset \quad C \vdash \tau :: \emptyset}{C \vdash (\theta^a \to \tau) :: \emptyset} \qquad \frac{C \vdash \tau :: \emptyset}{C \vdash \tau\ \textbf{ref} :: \emptyset}$$

[2] On pourra noter que pour le systèmes de types original, il n'est possible de rendre polymorphes que les variables de types ou de rangée, mais pas les variables de degrés. Cependant, il est possible d'arriver au même effet avec une contrainte $p \leqslant \alpha$ dans l'environnement, grâce au caractère *statique* du typage. En revanche, dans l'interpréteur MlObj , étant donné que l'inférence de types se fait de façon progressive au fur et à mesure des définitions, nous inclurons aussi les variables de degrés dans Q. Ceci posera quelques difficultés pour décider quelles variables peuvent être rendues polymorphes...

$$\frac{}{C \vdash \{\} :: L} \qquad \frac{C \vdash \rho :: L \cup \{l\} \quad C \vdash \tau :: \emptyset}{C \vdash \{\rho, l : \tau\} :: L}\ l \notin L \qquad \frac{Q, C \vdash \sigma :: \emptyset}{C \vdash (\forall Q.\sigma) :: \emptyset}\ dom(Q) \cap dom(C) = \emptyset$$

On remarquera que si $C \vdash \tau :: L$ où τ n'est pas un type enregistrement, alors $L = \emptyset$ (idem pour $C \vdash \sigma :: L$).

Contraintes L'ensemble des contraintes C ne contient pas seulement des hypothèses sur les variables de types et leurs champs non autorisés. Il contient également des inégalités de la forme $p \leqslant \alpha$, où p est une variable de degré et α une expression de degré. Le système de règles suivant permet alors de dériver des inégalités de la forme $\alpha \leqslant \beta$ à partir de ces contraintes.

$$\frac{}{C \vdash 0 \leqslant \alpha} \qquad \frac{}{C \vdash \alpha \leqslant 1} \qquad \frac{}{C \vdash \alpha \leqslant \alpha} \qquad \frac{C \vdash \alpha \leqslant \beta \quad C \vdash \beta \leqslant \alpha'}{C \vdash \alpha \leqslant \alpha'}$$

$$\frac{}{p \leqslant \alpha, C \vdash p \leqslant \alpha} \qquad \frac{}{C \vdash \alpha \wedge \beta \leqslant \alpha} \qquad \frac{}{C \vdash \alpha \wedge \beta \leqslant \beta} \qquad \frac{C \vdash \alpha \leqslant \beta \quad C \vdash \alpha \leqslant \beta'}{C \vdash \alpha \leqslant \beta \wedge \beta'}$$

C désigne donc un ensemble de contraintes de la forme $t :: L$ ou $p \leqslant \alpha$, et on écrira $t :: L, C$ et $p \leqslant \alpha, C$ pour l'opération d'extension (à noter que $Q = t_1 :: L_1, \ldots, t_n :: L_n$ est aussi un ensemble de contraintes ; on écrira donc Q, C).

Environnements bien formés Un *environnement* Γ est un ensemble fini de liaisons de la forme $x : \sigma$ ou $u : \tau$, liant des variables à leur schéma de types et des adresses mémoire à leur type. Le petit système à trois règles suivant infère des jugements de la forme $C \vdash \Gamma$ qui vérifie que les (schémas de) types contenus dans Γ sont bien formés vis-à-vis de C. [3]

$$\frac{}{C \vdash \emptyset} \qquad \frac{C \vdash \sigma :: \emptyset \quad C \vdash \Gamma}{C \vdash x : \sigma, \Gamma} \qquad \frac{C \vdash \tau :: \emptyset \quad C \vdash \Gamma}{C \vdash u : \tau, \Gamma}$$

Typage des termes Un jugement de typage est de la forme $C; \Gamma^\gamma \vdash M : \tau$, où C est un ensemble de contraintes, M un terme, τ un type et Γ^γ un *contexte de typage*. Ce dernier est constitué d'un environnement Γ et d'un ensemble fini γ d'hypothèses de la forme $x : \alpha$, liant des variables à des expressions de degrés. Cette annotation sur x indique si la variable se trouve en position dangereuse ou si elle est protégée. De plus, le domaine de Γ restreint aux seules variables doit être le même que celui de γ. On pourra donc écrire Γ^γ sous la forme :

$$u_1 : \tau_1, \ldots, u_k : \tau_k, x_1 : \sigma_1^{\alpha_1}, \ldots, x_n : \sigma_n^{\alpha_n}$$

et on utilisera les notations usuelles $u : \tau, \Gamma^\gamma$ et $x : \sigma^\alpha, \Gamma^\gamma$ pour l'extension d'un contexte de typage.

Pour une expression de degré α et un terme M, la notation α_M désignera la fonction

$$\alpha_M(x) = \begin{cases} \alpha & \text{si } x \in fv(M) \\ 1 & \text{sinon} \end{cases}$$

On écrira également $\delta \wedge \gamma$ pour la fonction définie point à point par $(\delta \wedge \gamma)(x) = \delta(x) \wedge \gamma(x)$; et on notera 0 et 1 pour les fonctions constantes qui retournent les degrés 0 et 1.

[3] Dans [Bou04], il manque le cas de u.

Enfin, on définit la relation de sous-typage suivante pour un ensemble de variables X :

$$C \vdash \delta \leqslant \gamma \text{ sur } X \iff C \vdash \delta(x) \leqslant \gamma(x) \text{ pour tout } x \in X$$

Cette définition nous permet de donner la première règle du système de types, celle du sous-typage sur les degrés : le degré des variables peut toujours être "dégradé" sans danger.

$$\frac{C; \Gamma^{\gamma} \vdash M : \tau \quad C \vdash \delta \leqslant \gamma \text{ sur } fv(M)}{C; \Gamma^{\delta} \vdash M : \tau}$$

Pour le typage d'un axiome sur x, il nous reste à donner la définition d'une instance d'un schéma de types. On note S une substitution des variables de type, de rangée et de degré, de domaine fini. Soit $\sigma = \forall Q.\tau$ un schéma de type. On dit que τ' est une instance de σ et on note $C \vdash \sigma \succeq \tau'$ s'il existe une substitution S telle que $\tau' = S(\tau)$, $dom(S) \subseteq Q$ et $C \vdash S(Q)$ (cette condition est nécessaire afin de s'assurer que S ne donne pas naissance à des types mal formés).

L'axiome pour une variable x est alors donné par la règle ci-dessous ; dans le contexte de typage, on voit que x n'est pas protégé et a le degré 0 ; en revanche, toutes les autres variables le sont et ont le degré 1.

$$\frac{C \vdash x : \sigma, \Gamma \quad C \vdash \sigma \succeq \tau}{C; x : \sigma^0, \Gamma^1 \vdash x : \tau}$$

Pour une abstraction $\lambda x M$, le degré de x dans le contexte de typage est repris comme annotation dans le type de la fonction. On notera qu'il ne peut pas s'agir d'une conjonction ; mais ceci n'est en rien restrictif car il est toujours possible d'introduire une nouvelle variable p et de rajouter une contrainte $p \leqslant \alpha$. Par ailleurs, puisqu'une fonction est une valeur et que son évaluation ne demande pas l'évaluation du corps de la fonction, toutes les autres variables qui apparaissent dans M reçoivent le degré 1.

$$\frac{C; x : \theta^a, \Gamma \vdash M : \tau}{C; \Gamma^1 \vdash \lambda x M : (\theta^a \to \tau)} \qquad \frac{C; \Gamma^{\gamma} \vdash M : \theta^a \to \tau \quad C; \Gamma^{\gamma} \vdash N : \theta}{C; \Gamma^{0_M \wedge \delta} \vdash (MN) : \tau} \quad (1)$$

(1) où δ désigne :

$$\delta(x) = \begin{cases} a & \text{si } N = x \\ (a_N \wedge \gamma)(x) & \text{sinon} \end{cases}$$

A l'inverse, pour une application, toutes les variables apparaissant dans la partie fonction sont potentiellement dangereuses (par exemple, x est dangereux dans $(\lambda y\ (x\ y))\ V$). Elles sont donc toutes mises à 0 par Γ^{0_M}. Les autres variables ont un degré au mieux égal à a (si elles apparaissent dans l'argument) ou à celui qu'elles ont dans N (d'où le $a_N \wedge \gamma$). Lorsque l'argument N est seulement une variable, il est possible d'être un peu plus précis en lui donnant seulement le degré a (sinon son degré serait 0, puisque le degré de x dans x est 0). En effet, puisque nous devinons que M est de la forme $\lambda y M'$, on peut affirmer que le degré de x dans (Mx) est le même que celui de y dans M'. Cette amélioration est indispensable afin de pouvoir typer des termes du style **let rec** $x = G\ x$ **in** M.

Examinons maintenant le cas de **let** $x = M$ **in** N. En ce qui concerne le polymorphisme, il n'est introduit que si M est un terme pur (autrement dit, si M ne peut pas produire d'effet de bord, c'est-à-dire d'affectation dans la mémoire). Les variables ont pour degré celui qu'elles ont dans M et N, ou encore le degré α de x lorsqu'elles apparaissent dans N. Dans le cas où N est une expression pure, on peut montrer que cette dernière contrainte peut être relâchée, et se contenter de γ. Pour le **let rec**, la règle de typage est strictement la même, sauf que la variable récursive x voit son degré forcé à 1 dans le typage de M (et son type θ doit bien évidemment être le même que celui trouvé pour M). Ceci nous permet d'assurer que la récursion est bien évaluable. On pourra

noter que, lorsque M est une fonction, cette condition est satisfaite puisque toutes les variables ont un degré 1 dans un tel cas. Il est donc toujours possible de typer **let rec** $f = \lambda x M$ **in** N comme en ML, au moins en ce qui concerne les degrés.

$$\frac{Q, C; \Gamma^\gamma \vdash M : \theta \quad C; x : (\forall Q.\theta)^\alpha, \Gamma^\gamma \vdash N : \tau}{C; \Gamma^\delta \vdash \textbf{let } x = M \textbf{ in } N : \tau} \ (2)$$

$$\frac{Q, C; x : \theta^1, \Gamma^\gamma \vdash M : \theta \quad C; x : (\forall Q.\theta)^\alpha, \Gamma^\gamma \vdash N : \tau}{C; \Gamma^\delta \vdash \textbf{let rec } x = M \textbf{ in } N : \tau} \ (2)$$

où (2) désigne : $t \in dom(Q) \Longrightarrow t \notin dom(C)$ et Q est vide si M n'est pas pur, et

$$\delta = \begin{cases} \alpha_N \wedge \gamma & \text{si } N \text{ n'est pas pur} \\ \gamma & \text{sinon} \end{cases}$$

Passons maintenant aux constructions relatives aux références et aux adresses mémoire. Dans tous les cas, on retourne simplement le type de la primitive correspondante, en vérifiant que le type retourné et l'environnement de typage sont bien formés. En ce qui concerne les degrés, toutes les variables ont le degré 1 car elles n'apparaissent pas dans le terme, et les degrés des arguments dans les fonctions sont mis à 0 car l'on considère que ces fonctions utilisent leurs arguments [4].

$$\frac{C \vdash u : \tau, \Gamma}{C; u : \tau, \Gamma^1 \vdash u : \tau \textbf{ ref}} \qquad \frac{C \vdash \Gamma \quad C \vdash \tau :: \emptyset}{C; \Gamma^1 \vdash \textbf{ref} : \tau^0 \to \tau \textbf{ ref}} \qquad \frac{C \vdash \Gamma \quad C \vdash \tau :: \emptyset}{C; \Gamma^1 \vdash \textbf{ !} : (\tau \textbf{ ref})^0 \to \tau}$$

$$\frac{C \vdash \Gamma \quad C \vdash \tau :: \emptyset}{C; \Gamma^1 \vdash \textbf{set} : (\tau \textbf{ ref})^0 \to \tau^0 \to \textbf{unit}} \qquad \frac{C \vdash \Gamma}{C; \Gamma^1 \vdash () : \textbf{unit}}$$

Le typage des enregistrements n'appelle que peu de commentaires, car il se déduit facilement des constructions correspondantes. On notera que pour $\{M, l = N\}$, on vérifie que le type de M ne contient pas de champ de label l par $C \vdash \rho :: \{l\}$. A l'inverse, pour $(M.l)$ et $(M \setminus l)$, la prémisse assure que le type de M contient bien un champ de label l. En ce qui concerne les degrés, ils sont ici simplement propagés.

$$\frac{C \vdash \Gamma}{C; \Gamma^1 \vdash \{\} : \{\}} \qquad \frac{C; \Gamma^\gamma \vdash M : \rho \quad C; \Gamma^\gamma \vdash N : \tau \quad C \vdash \rho :: \{l\}}{C; \Gamma^\gamma \vdash \{M, l = N\} : \{\rho, l : \tau\}}$$

$$\frac{C; \Gamma^\gamma \vdash M : \{\rho, l : \tau\}}{C; \Gamma^\gamma \vdash (M.l) : \tau} \qquad \frac{C; \Gamma^\gamma \vdash M : \{\rho, l : \tau\}}{C; \Gamma^\gamma \vdash (M \setminus l) : \rho}$$

Typage des configurations Enfin, pour être complet, nous donnons les règles permettant de typer les configurations. Tout d'abord, les deux règles suivantes donnent le typage pour la mémoire, dont les jugements sont de la forme $C; \Gamma \vdash S$ (il n'y a plus besoin des degrés ici). Leur seul rôle est de s'assurer que le type des valeurs stockées dans la mémoire est bien le même que celui répertorié dans l'environnement.

$$\frac{C \vdash \Gamma}{C; \Gamma \vdash \varepsilon} \qquad \frac{C; u : \tau, \Gamma^\gamma \vdash V : \tau \quad C; u : \tau, \Gamma \vdash S}{C; u : \tau, \Gamma \vdash u := V; S}$$

Enfin, une configuration est typable si ses deux composantes sont typables pour les mêmes contraintes et le même environnement :

$$\frac{C; \Gamma \vdash S \quad C; \Gamma^\gamma \vdash M : \tau}{C; \Gamma \vdash [S \mid M] : \tau}$$

[4] ce qui est discutable dans certains cas, voir la note dans le chapitre sur l'inférence...

1.3 Résultats

Cette section reprend les principaux résultats énoncés et démontrés dans [Bou04].

Le premier d'entre eux concerne l'invariance du typage par réduction pour les termes et les configurations (propriété dite de "subject reduction").

Proposition 1.2
- *Si* $C;\Gamma^\gamma \vdash M : \tau$ *et* $M \to^* N$, *alors* $C;\Gamma^\gamma \vdash N : \tau$.
- *Si* $C;\Gamma \vdash [S \mid M] : \tau$ *où* $u \in dom(\Gamma) \Longrightarrow u \in dom(S)$, *et* $[S \mid M] \to^* [S' \mid N]$, *alors il existe* Δ *tel que* $C;\Delta \vdash [S \mid N] : \tau$.

Le deuxième résultat concerne la sûreté du typage : l'exécution d'un terme bien typable ne peut pas provoquer d'erreur. Préalablement, il nous faut préciser cette notion d'erreur par la définition des termes *erronés*, qui fait elle-même appel à la définition suivante :

Définition 1.3 *Un terme* M *est une* expression de tête *si* $M = \mathbf{H}[x]$ *avec* $x \notin capt(\mathbf{H})$, *où les contextes* $\mathbf{H}$ *sont donnés par la grammaire :*

$$\mathbf{H} ::= \mathbf{E}[([]\ V)] \mid \mathbf{E}[(\ \textbf{\textit{!}}\ [])] \mid \mathbf{E}[(\textbf{\textit{set}}\ [])] \mid \mathbf{E}[([].l)] \mid \mathbf{E}[([] \setminus l)]$$

Définition 1.4 *Un terme* M *est* erroné *s'il contient une sous-expression d'une des formes suivantes :*
- $(V\ N)$, *où* V *est soit une adresse mémoire, soit* $()$, *soit un enregistrement ;*
- $(\textbf{\textit{let rec}}\ x = \mathbf{H}[x]\ \textbf{\textit{in}}\ M)$ *avec* $x \notin capt(\mathbf{H})$ *;*
- $(\textbf{\textit{let rec}}\ x = \mathbf{E}[N]\ \textbf{\textit{in}}\ M)$ *où* N *est soit* $(\textbf{\textit{ref}}\ V)$, *soit* $((\textbf{\textit{set}}\ u)\ V)$ *avec* $x \in fv(V)$ *;*
- $(\ \textbf{\textit{!}}\ V)$ *ou* $(\textbf{\textit{set}}\ V)$ *où* V *n'est ni une variable ni une adresse mémoire ;*
- $\{V, l = N\}$ *où* V *n'est pas un enregistrement ;*
- $(V.l)$ *ou* $(V \setminus l)$, *où* V *n'est ni une variable ni un enregistrement non vide.*

On peut alors énoncer le résultat de sûreté :

Théorème 1.5 *Soit* $[S \mid M]$ *une configuration close typable (i.e. il existe* C, Γ *et* τ *tels que* $C;\Gamma \vdash [S \mid M] : \tau$*) et dont l'évaluation termine sur* $[S' \mid N]$. *Alors,* N *est une valeur non erronée et elle a le type* τ.

Un algorithme d'inférence est également présenté dans [Bou04] ; nous ne le détaillerons pas ici puisque son implémentation est décrite en détail dans le chapitre suivant. Néanmoins, on peut mentionner que ses preuves de correction ont été établies et que cet algorithme permet de construire le typage principal de tout terme typable (cf. [Bou04] pour les définitions et l'énoncé précis des résultats).

1.4 Extensions du langage

Le langage présenté dans ce chapitre est certes complet d'un point de vue théorique, mais reste encore trop simple pour pouvoir l'utiliser comme un mini-langage de programmation. Afin d'y remédier, nous allons lui ajouter quelques possibilités, valables dans toute la suite.

Liaisons multiples Nous étendons les constructions **let** et **let rec** en admettant plusieurs définitions simultanées et mutuellement récursives dans le cas de **let rec**.

$$\begin{array}{rcl} M & ::= & \ldots \\ & | & \textbf{let } x_1 = M_1 \textbf{ and } \cdots \textbf{ and } x_n = M_n \textbf{ in } N \\ & | & \textbf{let rec } x_1 = M_1 \textbf{ and } \cdots \textbf{ and } x_n = M_n \textbf{ in } N \end{array}$$

Précisons la sémantique de cette dernière forme. Un **let rec** ne se réduit que lorsque toutes les liaisons M_i ont été évaluées en des valeurs V_i. Si l'on note :

$$LR(P) = \textbf{let rec } x_1 = V_1 \textbf{ and } \cdots \textbf{ and } x_n = V_n \textbf{ in } P$$

le terme obtenu avec le corps P, alors la règle de réduction du **let rec** s'écrit :

$$LR(M) \;\rightarrow\; \{^{LR(V_1)}/_{x_1}, \ldots, ^{LR(V_n)}/_{x_n}\}(M)$$

En ce qui concerne les contextes d'évaluation, afin de conserver une sémantique déterministe pour le langage, nous choisissons d'évaluer les liaisons de gauche à droite, avec :

$$\mathbf{E} \;::=\; \cdots \;\mid\; \textbf{let rec } x_1 = V_1 \textbf{ and } \cdots \textbf{ and } x_i = \mathbf{E} \textbf{ and } \cdots \textbf{ and } x_n = M_n \textbf{ in } M$$

Le mot-clé **and** n'est donc pas commutatif pour cette sémantique.

Enfin, nous donnons la généralisation de la règle de typage pour un **let rec** avec liaisons multiples ; celle-ci se déduit sans difficulté à partir de la version avec liaison unique :

$$\frac{\forall i \;\; Q_i, C; x_i : \theta_i^1, \Gamma^\gamma \vdash M_i : \theta_i \qquad C; x_1 : (\forall Q_1.\theta_1)^{\alpha_1}, \ldots, x_n : (\forall Q_n.\theta_n)^{\alpha_n}, \Gamma^\gamma \vdash N : \tau}{C; \Gamma^\delta \vdash \textbf{let rec } x_1 = M_1 \textbf{ and } \cdots \textbf{ and } x_n = M_n \textbf{ in } N : \tau} \quad (3)$$

où (3) désigne : $t \in dom(Q_i) \Longrightarrow t \notin dom(C)$ et Q_i est vide si M_i n'est pas pur, et

$$\delta = \begin{cases} (\alpha_1)_N \wedge \cdots \wedge (\alpha_n)_N \wedge \gamma & \text{si } N \text{ n'est pas pur} \\ \gamma & \text{sinon} \end{cases}$$

Conditionnelle Nous ajoutons également une instruction **if** :

$$\begin{array}{rcl} M & ::= & \ldots \\ & | & \textbf{if } M \textbf{ then } N_1 \textbf{ else } N_2 \end{array}$$

Sa sémantique est celle utilisée dans les langages de programmation, à savoir l'évaluation de M détermine quelle branche sera évaluée (et donc pas celle d'un appel par valeur).

Entiers, booléens, listes Nous ajoutons enfin des types de base, comme les entiers et les booléens, et des objets composés, à savoir les listes.

$$\begin{array}{rcll} M & ::= & \ldots & \\ & | & n \in \mathbb{N} & \\ & | & \textbf{true} & \\ & | & \textbf{false} & \\ & | & [] & \text{liste vide} \\ & | & M :: N & \text{cons} \end{array}$$

avec leur sémantique évidente.

Pour être complet, nous ajoutons donc également les types suivants à la syntaxe :

$$\begin{array}{rcl} \tau & ::= & \ldots \\ & | & \textbf{int} \\ & | & \textbf{bool} \\ & | & \tau\ \textbf{list} \end{array}$$

Chapitre 2

Algorithmes d'unification et d'inférence

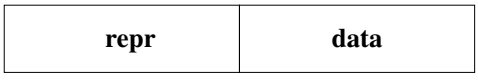
Ce chapitre décrit un algorithme d'inférence de types pour le système avec degrés vu au chapitre précédent. Un tel algorithme est déjà détaillé dans [Bou04], mais il est de nature "mathématique", à savoir qu'il repose sur des résolutions de contraintes et de l'unification à partir de substitutions les plus génériques possibles (*most general unifiers*). Son but était essentiellement de coller au formalisme mathématique, afin de pouvoir prouver la principalité du type inféré. Mais, même s'il est possible de le réutiliser tel quel, il s'avère peu efficace en pratique et conduit à des manipulations inutiles sur de grosses structures de données.

Notre but ici est de donner une version implémentable et pratique de cet algorithme, à savoir capable de passer à l'échelle sur un vrai langage de programmation. Ce que nous allons décrire est le système d'inférence utilisé réellement dans l'interpréteur MLOBJ décrit dans les chapitres suivants. Les idées générales proviennent de la Section 6.7 de [ASU89], elle-même inspirée de [Rob65], où est décrit un algorithme d'unification pour un mini-langage à la ML. Elles ont été librement adaptées et étendues au présent langage.

2.1 Cadre général

L'idée maîtresse est de remplacer le schéma inefficace "création de la substitution la plus générique, puis application de cette substitution au reste du monde" par de l'unification directement sur les termes de types. Ainsi, à la fin de l'opération, les deux termes à unifier représentent le même type ET les substitutions nécessaires ont eu lieu à tous les autres endroits où ces termes (ou une de leurs sous-occurrences) sont utilisés.

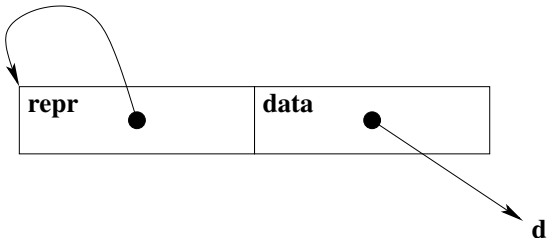
Pour aboutir à ce résultat, la structure centrale que nous allons manipuler pour décrire les types est celle d'une cellule à deux champs :



Le champ `data` contient la structure syntaxique du type et dépend de la sorte de cellule considérée. Par exemple, il s'agira d'un simple *tag* pour le type *int*, de trois pointeurs vers d'autres cellules pour le type fonctionnel $\theta^a \to \tau$, etc... Nous détaillerons les cas qui méritent notre attention dans la suite.

Le champ `repr` est un pointeur vers une cellule, censée être un "représentant" pour cette cellule, i.e. le représentant d'une certaine classe d'équivalence. S'il pointe vers la cellule elle-même, cela signifie que la cellule est sa propre représentante et que les données du champ `data` sont pertinentes. S'il pointe vers une autre cellule, c'est là qu'il faut chercher les vraies données et donc le champ `data` est caduc. Unifier deux types reviendra donc à essayer d'unifier les représentants (et donc les classes d'équivalence) tout au long de l'arbre syntaxique des types (en fait, nous allons plus généralement unifier des graphes, et donc admettre n'importe quel type récursif[5]).

Création d'une nouvelle cellule Soit à créer une nouvelle cellule contenant les données `d`. Il suffit d'allouer et d'initialiser la structure suivante :



Recherche du représentant Pour trouver le représentant d'une cellule u, il suffit de suivre les champs `repr` :

$$
\begin{array}{rcl}
REPR(u) & \triangleq & \texttt{si } u \texttt{ et } u.repr \texttt{ sont le même pointeur} \\
& & \texttt{alors renvoyer } u \\
& & \texttt{sinon } \; v \leftarrow REPR(u.repr) \\
& & \qquad u.repr \leftarrow v \\
& & \qquad \texttt{renvoyer } v
\end{array}
$$

On notera que dans le cas où u n'est pas son propre représentant, on met à jour son champ `repr` afin de raccourcir au maximum la chaîne d'indirections, et donc de pouvoir accéder directement au représentant la fois suivante.

Unification (idée générale) Pour unifier deux cellules u et v, il suffit d'en choisir une arbitrairement et de modifier son champ `repr` : $u.repr \leftarrow v$. Puis, on parcourt récursivement l'arbre syntaxique contenu dans le champ `data` afin d'unifier également les sous-types. Ceci dépend du type de la cellule considérée et sera vu au cas par cas dans la suite. A noter qu'on peut alors mettre le champ `data` de u à vide, afin de ne pas garder de pointeurs inutiles et de pouvoir récupérer la mémoire par le ramasse-miettes.

2.2 Unification de degrés

Cellule degré Il existe quatre types de cellules pour les degrés : deux sont constants et correspondent aux degrés 0 et 1, le troisième contient deux pointeurs vers d'autres cellules et correspond

[5] En guise de comparaison, l'algorithme d'inférence utilisé dans OCaml admet aussi les type récursifs, mais n'admet dans un deuxième temps que ceux pour lesquels la récursion traverse un objet ou un constructeur de type. Ainsi, les types récursifs dans la partie purement fonctionnelle ne sont pas admis, à juste titre car ils sont en grande majorité le fruit d'erreurs de programmation (dans un cadre d'utilisation classique). Nous aurions pu utiliser la même restriction pour MlObj, mais nous préférons privilégier ici la liberté d'expérimentation, à charge pour le programmeur d'être vigilant. En l'état, ceci équivaut à utiliser l'option `-rectypes` d'OCaml ; une des conséquences est que tout terme purement fonctionnel est typable dans MlObj.

au type degré $\alpha \wedge \beta$, le dernier correspond à une variable de degré p et mérite une attention particulière. Pour une telle cellule, nous allons conserver également dans le champ `data` deux listes de pointeurs, l'une pour les variables de degrés q telles que $q \leqslant p$ et l'autre pour les variables q telles que $p \leqslant q$. Si u désigne la cellule en question, nous appellerons $PRED(u)$ et $SUCC(u)$ respectivement ces deux listes. Un des invariants du système sera de conserver la propriété de cohérence :

$$u \in SUCC(v) \Leftrightarrow v \in PRED(u)$$

Ainsi, l'ensemble des variables de degrés pourra être vu comme un graphe orienté, dans lequel chaque noeud connaît directement l'ensemble de ses successeurs et de ses prédécesseurs.

Unification avec 1 Soit à unifier une cellule u avec le degré constant 1. Cette opération se définit ainsi (on suppose qu'il existe une cellule canonique *one* représentant 1) :

$MGU\_DEGRE\_UN(u) \triangleq$

```
u ← REPR(u)
si u représente le degré 1
alors u.repr ← one
sinon si u représente le degré 0
alors erreur
sinon si u représente u1 ∧ u2
alors   MGU_DEGRE_UN(u1)
        MGU_DEGRE_UN(u2)
sinon   /* u représente une variable de degré */
        pour tout v ∈ PRED(u)
            SUCC(v) ← SUCC(v) \ {u}
        u.repr ← one
        pour tout v ∈ SUCC(u)
            MGU_DEGRE_UN(v)
```

Si u représente 1, pas de problème. Si c'est 0, on a une erreur de typage car il est impossible de satisfaire la contrainte $0 = 1$. Si u représente $\alpha \wedge \beta$, on s'appelle récursivement car $\alpha \wedge \beta = 1$ si et seulement si $\alpha = 1$ et $\beta = 1$. Si u est une variable de degré p, on l'unifie avec 1, on supprime tous les prédécesseurs q car $q \leqslant 1$ est toujours vrai, et on unifie tous les successeurs avec 1 car $1 \leqslant q \Rightarrow q = 1$.

Unification avec 0 Définie de façon symétrique au cas précédent, seulement pour les cas où u représente un degré a (i.e. pas une conjonction $\alpha \wedge \beta$).

Unification de deux degrés Soient deux degrés a et b (i.e. pas des conjonctions), représentées par les cellules u et v. L'unification se fait facilement par l'algorithme suivant à partir des deux précédents, et en prenant garde à conserver l'invariant de cohérence dans le cas où on unifie deux

variables :

$$
\begin{array}{lcl}
MGU\_DEGRE(u,v) & \triangleq & u \leftarrow REPR(u) \\
& & v \leftarrow REPR(v) \\
& & \texttt{si } u \texttt{ et } v \texttt{ sont la même cellule} \\
& & \texttt{alors rien} \\
& & \texttt{sinon si } u \texttt{ représente le degré } 0 \\
& & \texttt{alors } MGU\_DEGRE\_ZERO(v) \\
& & \texttt{sinon si } u \texttt{ représente le degré } 1 \\
& & \texttt{alors } MGU\_DEGRE\_UN(v) \\
& & \texttt{sinon si } v \texttt{ représente le degré } 0 \\
& & \texttt{alors } MGU\_DEGRE\_ZERO(u) \\
& & \texttt{sinon si } v \texttt{ représente le degré } 1 \\
& & \texttt{alors } MGU\_DEGRE\_UN(u) \\
& & \texttt{sinon } \texttt{/* } u \texttt{ et } v \texttt{ représentent des variables */} \\
& & \quad u.repr \leftarrow v \\
& & \quad \texttt{pour tout } w \in PRED(u) \\
& & \quad\quad SUCC(w) \leftarrow (SUCC(w) \setminus \{u\}) \cup \{v\} \\
& & \quad \texttt{pour tout } w \in SUCC(u) \\
& & \quad\quad PRED(w) \leftarrow (PRED(w) \setminus \{u\}) \cup \{v\} \\
& & \quad PRED(v) \leftarrow (PRED(u) \cup PRED(v)) \setminus \{u,v\} \\
& & \quad SUCC(v) \leftarrow (SUCC(u) \cup SUCC(v)) \setminus \{u,v\}
\end{array}
$$

Ajout d'une contrainte $a \leqslant b$ Soient deux cellules u et v telles que u et v représentent des degrés a et b. L'algorithme suivant ajoute la contrainte correspondant à $a \leqslant b$ (le cas d'erreur correspond à la contrainte insatisfiable $1 \leqslant 0$) :

$$
\begin{array}{lcl}
AJOUT\_INEQ(u,v) & \triangleq & u \leftarrow REPR(u) \\
& & v \leftarrow REPR(v) \\
& & \texttt{si } u \texttt{ représente } 1 \texttt{ et } v \texttt{ représente } 0 \\
& & \texttt{alors erreur} \\
& & \texttt{sinon si } u \texttt{ représente } 0 \texttt{ ou } v \texttt{ représente } 1 \\
& & \texttt{alors rien} \\
& & \texttt{sinon si } u \texttt{ représente une variable et } v \texttt{ représente } 0 \\
& & \texttt{alors } MGU\_DEGRE\_ZERO(u) \\
& & \texttt{sinon si } u \texttt{ représente } 1 \texttt{ et } v \texttt{ représente une variable} \\
& & \texttt{alors } MGU\_DEGRE\_UN(v) \\
& & \texttt{sinon } \texttt{/* } u \texttt{ et } v \texttt{ représentent des variables */} \\
& & \quad \texttt{si } u \texttt{ et } v \texttt{ sont la même cellule} \\
& & \quad \texttt{alors rien} \\
& & \quad \texttt{sinon } SUCC(u) \leftarrow SUCC(u) \cup \{v\} \\
& & \quad\quad\quad\quad\ PRED(v) \leftarrow PRED(v) \cup \{u\}
\end{array}
$$

Ajout d'une contrainte $a \leqslant \alpha$ On étend cet algorithme au cas plus général où v est quelconque et représente α. Il nous reste donc seulement à ajouter le cas où v représente $v_1 \wedge v_2$ par :

$$
\begin{array}{lcl}
AJOUT\_INEQ(u,v) & \triangleq & AJOUT\_INEQ(u,v_1) \\
& & AJOUT\_INEQ(u,v_2)
\end{array}
$$

car $a \leqslant \alpha \wedge \beta \Leftrightarrow a \leqslant \alpha$ et $a \leqslant \beta$ [6].

2.3 Unification d'enregistrements

Les cellules correspondant aux enregistrements suivent la structure des types enregistrements, à savoir le type enregistrement vide $\{\}$, l'extension de champ $\{\rho, l : \tau\}$ et la variable de rangée t. Les deux premiers n'appellent pas de commentaire particulier. Le troisième est détaillé ci-dessous. Un invariant garanti par le système est qu'il n'y a jamais deux champs de même label dans un même enregistrement.

Variable de rangée Une cellule correspondant à une variable de rangée contient comme données la liste des labels qu'il n'est pas possible d'utiliser à la place de cette variable. Si t désigne la variable en question et L la liste correspondante, ceci équivaut à avoir la contrainte $t :: L$ dans le système de types. L'algorithme fait en sorte de calculer la liste L minimale.

Unification Soient deux cellules u et v correspondant à des enregistrements. L'algorithme suivant $MGU\_RECORD(u, v)$ cherche à les unifier.

Comme dans tous les autres cas, on commence par trouver les représentants : $u \leftarrow REPR(u)$ et $v \leftarrow REPR(v)$. Si, par chance, u et v sont le même pointeur, on peut s'arrêter là. Sinon, étant donné que les labels n'ont pas d'ordre particulier dans les enregistrements, il ne sera pas possible de faire de l'unification au fur et à mesure en suivant l'arbre syntaxique comme pour les degrés et les types. Il va falloir faire la liste des champs dans u et v, afin de trouver ceux qui sont ou non en commun, et trouver les éventuelles variables de rangée dans u et v (autrement dit si les types sont clos ou non). Trois cas peuvent se produire :

- Les deux types enregistrements sont clos. Dans ce cas, soit $\{l_1 : u_1, \ldots, l_n : u_n\}$ et $\{m_1 : v_1, \ldots, m_p : v_p\}$ ces deux types (les u_i et v_j désignent les cellules représentant les types des champs). Pour que ces deux types soient unifiables, il faut nécessairement que $n = p$ et $\{l_1, \ldots, l_n\} = \{m_1, \ldots, m_p\}$, autrement dit que les deux listes de labels soient identiques. Sinon, l'inférence de types échoue et on renvoie une erreur de typage. Dans le cas positif, il ne reste plus qu'à unifier les types des champs : $MGU\_TYPE(u_i, v_j)$ pour tous les (i, j) tels que $l_i = m_j$.
- L'un des types est clos, l'autre possède une variable de rangée. Notons $\{l_1 : u_1, \ldots, l_n : u_n\}$ et $\{w,\ m_1 : v_1, \ldots, m_p : v_p\}$ ces deux types, où w est la cellule correspondant à la variable de rangée. Notons également L la liste de ses labels non autorisés. Pour que les deux enregistrements soient unifiables, il faut au moins que $\{m_1, \ldots, m_p\} \subseteq \{l_1, \ldots, l_n\}$. Notons $\{l_{i_1}, \ldots, l_{i_k}\} = \{l_1, \ldots, l_n\} \setminus \{m_1, \ldots, m_p\}$ les k labels qui apparaissent uniquement dans le premier type. Pour qu'il y ait unification, il faut encore que $l_{i_j} \notin L$ pour tout j. Dans tous les autres cas, on a une erreur de typage. Sinon, on peut instancier la cellule w avec les champs manquants : $w.repr \leftarrow \{l_{i_1} : u_{i_1}, \ldots, l_{i_k} : u_{i_k}\}$ (la structure en cellules nécessaire est recréée). Enfin, il reste à unifier les champs en commun : $MGU\_TYPE(u_i, v_j)$ pour tous les (i, j) tels que $l_i = m_j$.

[6] Une analyse plus fine des usages de cette fonction révélerait qu'elle est en fait toujours appelée avec une variable de degré comme premier argument u ; il serait donc possible de la simplifier. Une conséquence surprenante est que l'erreur de typage "$1 \leqslant 0$" ne sera donc jamais levée dans MLOBJ... Nous préférons cependant garder la version la plus générale possible, au cas où l'ajout de nouvelles constructions au langage et de nouvelles règles de typage le nécessiterait.

– Les deux types possèdent une variable de rangée. Notons $\{w_1,\ l_1 : u_1, \ldots, l_k : u_k, l_{k+1} : u_{k+1}, \ldots, l_n : u_n\}$ et $\{w_2,\ l_1 : v_1, \ldots, l_k : v_k, m_{k+1} : v_{k+1}, \ldots, m_p : v_p\}$ ces deux types où $l_1, \ldots, l_k$ sont les labels en commun, et $l_{k+1}, \ldots, l_n$ et $m_{k+1}, \ldots, m_p$ les labels différents. Notons également L_1 et L_2 les deux listes de labels non autorisés pour w_1 et w_2. Pour que les deux types enregistrements soient unifiables, il faut que $l_i \notin L_2$ pour $k+1 \leq i \leq n$ et $m_i \notin L_1$ pour $k+1 \leq i \leq p$. Sinon, on a une erreur de typage. Soit w_3 une nouvelle cellule de type variable de rangée dont la liste des labels non autorisés est $L_1 \cup L_2$. On effectue alors les instanciations : $w_1 \leftarrow \{w_3,\ m_{k+1} : v_{k+1}, \ldots, m_p : v_p\}$ et $w_2 \leftarrow \{w_3,\ l_{k+1} : u_{k+1}, \ldots, l_n : u_n\}$. Enfin, il reste à essayer d'unifier les champs en commun : $MGU\_TYPE(u_i, v_i)$ pour $1 \leq i \leq k$.

2.4 Unification de types

Les cellules correspondant aux types suivent de façon naturelle la syntaxe des types eux-mêmes. Au contraire des variables de degré ou de rangée, il n'y a pas d'information supplémentaire concernant les variables de type.

Unification Soient u et v deux cellules correspondant à des types. L'algorithme $MGU\_TYPE(u, v)$ suivant cherche à les unifier.

Comme dans tous les autres cas, on commence par trouver les représentants : $u \leftarrow REPR(u)$ et $v \leftarrow REPR(v)$. Si, par chance, u et v sont le même pointeur, on peut s'arrêter là. Sinon, on considère les types représentés par ces cellules :

– S'il s'agit de deux types simples *identiques* (*int*, *bool*, *unit*, ...), l'unification est immédiate : $u.repr \leftarrow v$.
– S'il s'agit de deux types fonctionnels $u_1 \ {}^{u_2}\!\rightarrow u_3$ et $v_1 \ {}^{v_2}\!\rightarrow v_3$ (les u_i et v_j sont les cellules correspondant aux sous-types et aux sous-degrés), on unifie u et v par $u.repr \leftarrow v$, puis on tente l'unification des composantes : $MGU\_TYPE(u_1, v_1)$, $MGU\_DEGRE(u_2, v_2)$ et $MGU\_TYPE(u_3, v_3)$ (on rappelle que d'après la syntaxe des types, les degrés u_2 et v_2 ne peuvent pas être des conjonctions ; on peut donc utiliser MGU_DEGRE).
– S'il s'agit de deux types enregistrements u' et v', on unifie u et v par $u.repr \leftarrow v$, puis on tente l'unification des champs par $MGU\_RECORD(u', v')$.
– S'il s'agit de deux types construits *identiques*, par exemple $u'\ ref$ et $v'\ ref$ (ou $u'\ list$ et $v'\ list$, etc...), on unifie u et v par $u.repr \leftarrow v$, puis on unifie les sous-types par $MGU\_TYPE(u', v')$.
– Si u est une variable de type, on l'instancie avec $u.repr \leftarrow v$.
– Si v est une variable de type, on l'instancie avec $v.repr \leftarrow u$.
– Dans tous les autres cas, u et v ne sont pas unifiables et on renvoie une erreur de typage.

On notera enfin que les algorithmes d'unification présentés jusqu'ici fonctionnent correctement (i.e. terminent) même sur des types récursifs, c'est-à-dire sur des graphes cycliques. En effet, puisqu'on unifie toujours les représentants avant de passer aux sous-types, si jamais on retombe sur ces mêmes cellules lors d'une récursion, ces algorithmes constateront au deuxième passage que les représentants sont les mêmes et ils stopperont sans faire de descente récursive.

2.5 Généralisation et instanciation

Commençons par rappeler quelques définitions standards :

- Un *schéma de type* $\sigma = \forall Q.\tau$ est la donnée d'un type τ et d'un ensemble de variables de type Q. Les variables libres de τ apparaissant dans Q sont dites *polymorphes*, les autres sont dites *monomorphes*. Pour l'inférence de types, le type τ sera une cellule représentant τ ; et les variables de Q des cellules représentant des variables de type, de rangée ou de degré.
- Un *environnement de typage* E consiste en une suite de liaisons de la forme $x : \sigma$ entre des variables et des schémas de type. On notera $E, x : \sigma$ l'ajout d'une nouvelle liaison pour x (qui masque une éventuelle liaison pour x dans E si elle existe), et $E(x)$ le schéma de type correspondant à x dans E.

Le problème de la généralisation consiste à déterminer quelles variables d'un type τ peuvent être généralisées sans danger dans un environnement E donné. Si Q désigne ces variables, on pourra alors stocker le schéma de type $\forall Q.\tau$ dans l'environnement.

A l'inverse, si un identifieur a pour schéma de type $\forall Q.\tau$ dans l'environnement courant, l'instanciation permet de lui donner le type τ', qui est τ dans lequel toutes les variables polymorphes de Q ont été remplacées par des variables fraîches.

Depuis le système de types d'Hindley-Milner pour les langages à la ML, la technique classique consiste à introduire la généralisation au niveau du **let (rec)**, et l'instanciation au niveau des variables. On sait aussi que ceci pose un problème pour les langages avec références, et la technique utilisée habituellement de nos jours, est de ne généraliser le type de M dans **let** $x = M \ldots$ que si on peut assurer *syntaxiquement* que son évaluation n'a pas d'effet de bord.

Qu'en est-il pour notre langage avec enregistrements, et comment cela va-t-il affecter l'inférence de types ? Tout d'abord, nous allons suivre la tradition en introduisant la généralisation de types au niveau des opérateurs **let**. La règle syntaxique de décision sur M est adaptée au langage : on généralise uniquement si M est une expression *pure*, dont la syntaxe a déjà été donnée (Définition 1.1).

Etant donnée la structure cyclique des types, l'opération d'instanciation est délicate à écrire, mais elle ne pose pas de problème théorique particulier. Il reste la généralisation, autrement dit déterminer quelles variables doivent être rendues polymorphes. Pour les variables de type, nous allons utiliser la technique habituelle. Les variables de rangée suivront un traitement similaire. En revanche, pour les variables de degré, il va falloir être vigilant en raison des contraintes, qui peuvent être soit entre variables polymorphes, soit entre variables monomorphes, soit entre une variable polymorphe et une variable monomorphe. Nous allons montrer sur des exemples que cette dernière configuration doit être évitée, et que certaines des contraintes peuvent être éliminées lors de la généralisation.

Exemple 2.1 *Supposons que f soit une fonction entièrement polymorphe de type $\forall\{t_1, a, t_2\}.\ t_1\ ^a \rightarrow t_2$ dans l'environnement de typage global. Soit à typer le terme*

$$\textbf{\textit{let}}\ g = \lambda x\ f\ x\ \textbf{\textit{in}}\ \ldots$$

L'occurrence de f reçoit une instance du schéma de type, à savoir $t'_1\ ^{a'} \rightarrow t'_2$. Le sous-terme $\lambda x\ f\ x$ reçoit alors le type $t'_1\ ^b \rightarrow t'_2$, où b est une variable de degré fraîche, avec la contrainte $b \leqslant a'$. Puisque t'_1 et t'_2 n'apparaissent pas dans l'environnement de typage, elles sont généralisables. La même remarque s'applique pour b. De plus, puisque a' n'apparaît pas non plus dans l'environnement de typage, elle ne sera jamais utilisée ailleurs (une autre instance de f produira une instance fraîche a'' de a). Une instance b' de b produirait donc à chaque fois une contrainte inutile $b' \leqslant a'$, toujours satisfiable puisque a' ne sera jamais substituée. On en tire comme conclusion que lors de la généralisation, il est possible de supprimer les variables de degrés qui n'apparaissent ni dans l'environnement de typage ni dans le type à généraliser, ainsi que les contraintes sur ces variables. Ainsi, dans notre exemple, a' peut être supprimée, et g reçoit le schéma de type $\forall\{t'_1, b, t'_2\}.\ t'_1\ ^b \rightarrow t'_2$, avec aucune contrainte sur b.

Examinons maintenant le typage du terme suivant :

$$\textbf{\textit{let}}\ r = \ \textbf{\textit{ref}}\ f\ \textbf{\textit{in}}\ \textbf{\textit{let}}\ g = \lambda x\ (\textbf{\textit{!}}\ r)\ x\ \textbf{\textit{in}}\ \ldots$$

L'occurrence de f reçoit une instance du schéma de type, à savoir ${t'_1}^{\ a'} \to t'_2$. Le sous-terme ***ref*** *f n'étant pas pur, son type n'est pas généralisé, et r reçoit donc le schéma de type $\forall\emptyset.({t'_1}^{\ a'} \to t'_2)\ ref$. La suite du typage conduit à donner à $\lambda x\ (\textbf{\textit{!}}\ r)\ x$ le type $({t'_1}^{\ b} \to t'_2)$ où b est une variable de degré fraîche avec la contrainte $b \leqslant a'$. Dans ce type, t'_1 et t'_2 ne peuvent être généralisés, car ils apparaissent dans l'environnement de typage (dans le type de r). Ce n'est pas le cas de b, mais peut-elle être généralisée pour autant ? Supposons qu'elle le soit. Si dans la suite du code, une affectation sur r met une fonction non protectrice à la place de f, cela nous amènera à substituer a' par 0. Nous aurions alors pour g un schéma de type $\forall\{b\}\ldots.$ avec la contrainte systématique $b = 0$ pour toutes les instances passées ou futures de b, autrement dit la variable b n'est plus du tout polymorphe. Ceci nous paraît contraire à l'idée qu'une variable polymorphe ne doit pas être perturbée ou modifiée par des évènements postérieurs à sa création. Nous nous interdirons donc de généraliser b dans l'exemple ci-dessus. La règle pratique sera de consulter la composante connexe de b dans le graphe des contraintes. Si cette composante contient des variables non généralisables (i.e. apparaissant dans l'environnement de typage), a' dans notre exemple, la variable n'est pas généralisée.*

Nous allons formaliser précisément l'opération de généralisation. Mais avant cela, il sera utile de définir précisément les variables libres d'un type.

Variables libres Soit un type τ. On notera $ftv(\tau)$ l'ensemble des variables de type, de rangée et de degré libres dans τ. La définition de $ftv(\tau)$ se fait de façon standard par induction sur τ. Le seul cas particulier de traitement sera celui des variables de degrés ; nous prendrons la définition suivante :

$$ftv(a) = \texttt{la composante connexe pour} \leqslant \texttt{de}\ a\ \texttt{dans le graphe des contraintes}$$

Nous étendons cette définition aux schémas de type, avec :

$$ftv(\forall Q.\tau) = ftv(\tau) \setminus Q$$

Enfin, nous l'étendons naturellement aux environnements de typage par :

$$ftv(E) = \bigcup_{(x_i:\sigma_i)\in E} ftv(\sigma_i)$$

Par ailleurs, nous définissons également $fdeg(\tau)$ comme l'ensemble des variables de degrés libres apparaissant dans τ, cette fois-ci de façon usuelle, i.e. sans inclure la composante connexe.

D'un point de vue algorithmique, ces fonctions effectuent un parcours récursif des cellules représentant le type τ et de ses sous-cellules. Etant donné que les types peuvent être récursifs, il faut en plus conserver à jour une liste des cellules déjà visitées, afin de ne pas boucler. On écrira $ftv(u)$ et $fdeg(u)$ respectivement si u est la cellule qui représente τ.

Généralisation Soient u une cellule représentant un type τ, E un environnement de typage et M un terme. La fonction suivante $GEN(u, E, M)$ renvoie un schéma correspondant à la généralisation de τ.

$GEN(u, E, M) \triangleq$ `si` M `n'est pas pur`
`alors retourner` $\forall\emptyset.\ u$
`sinon` $Q_1 \leftarrow ftv(u) \setminus ftv(E)$
$Q'_1 \leftarrow$ `les variables de degrés de` Q_1
$Q_2 \leftarrow Q'_1 \setminus fdeg(u)$
`supprimer les variables de` Q_2 `et les contraintes associées`
`retourner` $\forall Q_1 \setminus Q_2\ .u$

Dans le cas où M est pur, seules les variables de types et de rangée apparaissant dans le type τ mais pas dans E sont généralisées. Concernant les variables de degrés, seules celles qui n'apparaissent pas dans la composante connexe d'une variable de l'environnement peuvent être généralisées. De plus, préalablement, on supprime celles d'entre elles qui n'apparaissent pas directement dans τ, car elles ne servent plus à rien.

Instanciation Enfin, il reste à décrire l'opération d'instanciation $INST(\sigma)$ pour un schéma de type $\sigma = \forall Q.u$ donné. Nous ne détaillerons pas l'algorithme complet, mais donnerons seulement ses étapes principales :

- Pour chaque variable de Q, on crée une nouvelle cellule-variable correspondante. Pour les variables de type, il n'y a rien de plus à faire. Pour les variables de rangée, on reprend la même liste de labels non autorisés. Pour les variables de degrés, on copie les contraintes en les traduisant (i.e. $a \leqslant b$ devient $a' \leqslant b'$ si a' est l'instance de a et b' celle de b).
- On parcourt ensuite récursivement le type u. Les variables polymorphes sont remplacées par leurs instances fraîchement créées, les autres, les anciennes variables monomorphes, sont conservées.
- A chaque étape du parcours récursif, une nouvelle cellule est créée, dont les champs sont remplis par récursion.
- Au final, on obtient un graphe de cellules qui est la copie du graphe d'origine, sauf en ce qui concerne les variables : les variables monomorphes sont partagées entre les deux graphes et les variables polymorphes de la copie sont des instances de celles de l'original.
- Afin de gérer les types récursifs sans boucler indéfiniment, on conserve dans une liste d'association pour chaque cellule v déjà visitée du graphe d'origine un pointeur vers la cellule w qui la remplace dans la copie. Si on repasse par v, on reprend w sans réaliser de nouvelle copie.

2.6 Algorithme d'inférence

Nous avons maintenant en main tous les outils pour réaliser l'inférence de types proprement dite. Celle-ci prendra la forme d'une fonction $INFER(E; M)$ définie par cas sur le terme M, où E est un environnement de typage liant toutes les variables libres de M. Le résultat de cette fonction est un couple (u, f) où u est une cellule de type (le type le plus générique inféré pour M), et f une fonction qui associe un degré à chaque variable. A ce sujet, nous rappelons les abréviations suivantes : si f et g sont deux telles fonctions, on note $f \wedge g$ la fonction $\lambda x.f(x) \wedge g(x)$; $\mathbf{0}$ et $\mathbf{1}$ sont les fonctions constantes qui renvoient toujours 0 et 1 ; et, si d est un degré, d_M désigne la fonction

définie par :

$$d_M(x) = \begin{cases} d & \text{si } x \in fv(M) \\ 1 & \text{sinon} \end{cases}$$

Identifieur L'inférence de types pour un identifieur est très simple : il suffit de créer une nouvelle instance du schéma de type pour x dans E ; de plus, toutes les variables sauf x sont protégées, on renvoie donc $\mathbf{0}_x$.

$$INFER(E; x) \triangleq \texttt{renvoyer } (INST(E(x)), \mathbf{0}_x)$$

Fonction On commence par étendre l'environnement avec une nouvelle variable de type fraîche monomorphe u, dans lequel on infère le type du corps de la fonction. Soit (v, f) le résultat obtenu. Le type retourné est alors $u \xrightarrow{w} v$ (ou plus précisément une nouvelle cellule qui représente ce type fonctionnel), où w est une variable de degré fraîche pour laquelle on a ajouté la contrainte $w \leqslant f(x)$ dans le système (c'est le seul endroit de l'inférence où peuvent être créées des contraintes entre variables de degrés). Enfin, toutes les variables étant protégées dans une fonction, on renvoie également $\mathbf{1}$.

$$\begin{array}{rcl} INFER(E; \lambda x M) & \triangleq & \texttt{soit } u \texttt{ une cellule-variable de type fraîche} \\ & & (v, f) \leftarrow INFER(E, x : \forall\emptyset.u; M) \\ & & \texttt{soit } w \texttt{ une cellule-variable de degré fraîche} \\ & & AJOUT\_INEQ(w, f(x)) \\ & & \texttt{retourner } (u \xrightarrow{w} v, \mathbf{1}) \end{array}$$

Application Pour une application, il faut unifier le type trouvé pour la partie fonction avec un type fonctionnel frais dont l'argument est le type inféré pour la partie argument de l'application. Le type retourné est alors le membre droit de ce type fonctionnel. Jusqu'ici, tout est classique et suit un schéma d'inférence usuel pour un langage fonctionnel. La différence réside dans les degrés dont l'expression finale est une reformulation de celle déjà expliquée au chapitre précédent pour le système de types [7].

$$\begin{array}{rcl} INFER(E; MN) & \triangleq & (u_1, f_1) \leftarrow INFER(E; M) \\ & & (u_2, f_2) \leftarrow INFER(E; N) \\ & & \texttt{soit } v \texttt{ une cellule-variable de type fraîche} \\ & & \texttt{soit } w \texttt{ une cellule-variable de degré fraîche} \\ & & MGU\_TYPE(u_1, u_2 \xrightarrow{w} v) \\ & & \texttt{retourner } (v, \mathbf{0}_M \wedge w_N \wedge \delta) \\ & & \qquad \texttt{avec } \delta(x) = \begin{cases} 1 & \text{si } N = x \\ f_2(x) & \text{sinon} \end{cases} \end{array}$$

Liaisons locales L'écriture de l'inférence est rendue plus ardue en raison des liaisons multiples simultanées. Pour chaque liaison $x_i = M_i$, on infère le type de M_i, puis on le généralise si possible. Les schémas de types obtenus servent à étendre l'environnement de typage courant, dans lequel on infère alors un type pour le corps du **let**. En ce qui concerne les degrés, on "additionne" les

[7] Dans [Bou04], l'algorithme d'inférence ne correspond pas au système de types si $x \notin fv(N)$. Par exemple, pour $\lambda x.((\lambda y.y)\ 3)$, on infère $t^0 \to int$ avec l'algorithme d'inférence, alors qu'on peut le typer $t^1 \to int$ grâce au système de types (il s'agit d'une coquille qu'on peut aisément corriger.)

contraintes produites par toutes les sous-inférences, auxquelles il faut encore ajouter celles de δ si le corps du **let** n'est pas pur (ceci a également été expliqué au chapitre précédent).

$$
\begin{array}{l}
INFER(E; \textbf{let } x_1 = M_1 \textbf{ and } \ldots \textbf{ and } x_n = M_n \textbf{ in } N) \triangleq \\
\quad \texttt{pour } i \texttt{ de } 1 \texttt{ à } n \\
\quad\quad (u_i, f_i) \leftarrow INFER(E; M_i) \\
\quad \texttt{pour } i \texttt{ de } 1 \texttt{ à } n \\
\quad\quad \sigma_i \leftarrow GEN(u_i, E, M_i) \\
\quad (v, g) \leftarrow INFER(E, x_1 : \sigma_1, \ldots, x_n : \sigma_n; N) \\
\quad \texttt{retourner } (v, \bigwedge_{1 \le i \le n} f_i \ \wedge g \wedge \delta) \quad \texttt{avec } \delta = \begin{cases} \mathbf{1} & \text{si } N \text{ est pur} \\ \bigwedge_{1 \le i \le n} g(x_i)_{M_i} & \text{sinon} \end{cases}
\end{array}
$$

Liaisons locales récursives L'inférence est similaire au cas précédent. Cependant, il faut commencer par créer des variables de types fraîches pour chaque liaison, car s'agissant de liaisons récursives, les variables doivent apparaître dans l'environnement global. Dans un deuxième temps, et avant la phase de généralisation, on unifie pour chaque liaison le type inféré avec cette variable fraîche. Ensuite, l'inférence du corps du **let rec** est possible comme précédemment. Enfin, il reste à vérifier que toutes les variables récursives sont bien protégées dans chacune des liaisons en les unifiant à 1 (vérifier que chaque variable est seulement protégée dans sa propre liaison ne suffit pas, comme on peut le constater sur l'exemple : **let rec** $x = y$ **and** $y = x$ **in** $\ldots$ [8]).

$$
\begin{array}{l}
INFER(E; \textbf{let rec } x_1 = M_1 \textbf{ and } \ldots \textbf{ and } x_n = M_n \textbf{ in } N) \triangleq \\
\quad \texttt{soient } u_i \ n \texttt{ cellules-variables de type fraîches} \\
\quad \texttt{pour } i \texttt{ de } 1 \texttt{ à } n \\
\quad\quad (v_i, f_i) \leftarrow INFER(E, x_1 : \forall\emptyset.u_1, \ldots, x_n : \forall\emptyset.u_n; M_i) \\
\quad \texttt{pour } i \texttt{ de } 1 \texttt{ à } n \\
\quad\quad MGU\_TYPE(u_i, v_i) \\
\quad \texttt{pour } i \texttt{ de } 1 \texttt{ à } n \\
\quad\quad \sigma_i \leftarrow GEN(v_i, E, M_i) \\
\quad (w, g) \leftarrow INFER(E, x_1 : \sigma_1, \ldots, x_n : \sigma_n; N) \\
\quad \texttt{pour } i \texttt{ de } 1 \texttt{ à } n \\
\quad\quad \texttt{pour } j \texttt{ de } 1 \texttt{ à } n \\
\quad\quad\quad MGU\_DEGRE\_UN(f_i(x_j)) \\
\quad \texttt{retourner } (w, \bigwedge_{1 \le i \le n} f_i \ \wedge g \wedge \delta) \ \texttt{(avec } \delta \texttt{ définie comme pour let)}
\end{array}
$$

Constantes L'inférence de types pour des constantes ne pose bien évidemment pas de problème particulier.

$$
\begin{array}{rcl}
INFER(E; n) & \triangleq & \texttt{retourner } (int, \mathbf{1}) \\
INFER(E; s) & \triangleq & \texttt{retourner } (string, \mathbf{1})
\end{array}
$$

[8] Si on décidait de l'ordre d'évaluation des définitions, on pourrait sans doute faire un peu mieux.

Conditionnelle L'inférence pour un **if** se résume à vérifier que le type du test est bien un booléen et que les branches **then** et **else** ont le même type.

$$\begin{array}{rcl} INFER(E; \textbf{if } B \textbf{ then } M \textbf{ else } N) & \triangleq & (u_1, f_1) \leftarrow INFER(E; B) \\ & & MGU\_TYPE(u_1, bool) \\ & & (u_2, f_2) \leftarrow INFER(E; M) \\ & & (u_3, f_3) \leftarrow INFER(E; N) \\ & & MGU\_TYPE(u_2, u_3) \\ & & \texttt{retourner } (u_3, f_1 \wedge f_2 \wedge f_3) \end{array}$$

Enregistrements Le typage d'un enregistrement vide équivaut à celui d'une constante.

$$INFER(E; \{\}) \triangleq \texttt{retourner } (\{\}, \mathbf{1})$$

Pour inférer le type de $\{M, l = N\}$, il faut d'abord vérifier que le type inféré pour M est celui d'un enregistrement qui ne contient pas de champ l (ceci se fait simplement en l'unifiant avec un enregistrement contenant uniquement une variable de rangée qui n'accepte pas l). Ensuite, on peut inférer un type pour N, et construire le type étendu (rappelons que par "$\{v, l : u_2\}$", on désigne en fait implicitement une nouvelle cellule représentant ce type).

$$\begin{array}{rcl} INFER(E; \{M, l = N\}) & \triangleq & (u_1, f_1) \leftarrow INFER(E; M) \\ & & \texttt{soit } v \texttt{ une cellule-variable de rangée fraîche, avec } L = \{l\} \\ & & \texttt{soit } v' \texttt{ la cellule-enregistrement contenant seulement } v \\ & & MGU\_TYPE(u_1, v') \\ & & (u_2, f_2) \leftarrow INFER(E; N) \\ & & \texttt{retourner } (\{v, l : u_2\}, f_1 \wedge f_2) \end{array}$$

Pour les opérations de sélection et de restriction, le début est le même, à savoir vérifier que le type de l'expression M est un enregistrement qui contient un champ l. Puis, on renvoie alors le type du champ lui-même ou le type de l'enregistrement sans le champ, suivant le cas.

$$\begin{array}{rcl} INFER(E; M.l) & \triangleq & (u, f) \leftarrow INFER(E; M) \\ & & \texttt{soit } v \texttt{ une cellule-variable de rangée fraîche, avec } L = \{l\} \\ & & \texttt{soit } w \texttt{ une cellule-variable de type fraîche} \\ & & MGU\_TYPE(u, \{v, l : w\}) \\ & & \texttt{retourner } (w, f) \\ INFER(E; M \setminus l) & \triangleq & (u, f) \leftarrow INFER(E; M) \\ & & \texttt{soit } v \texttt{ une cellule-variable de rangée fraîche, avec } L = \{l\} \\ & & \texttt{soit } w \texttt{ une cellule-variable de type fraîche} \\ & & MGU\_TYPE(u, \{v, l : w\}) \\ & & \texttt{soit } v' \texttt{ la cellule-enregistrement contenant seulement } v \\ & & \texttt{retourner } (v', f) \end{array}$$

Références et listes Il n'est pas besoin d'inclure ces cas dans l'algorithme d'inférence, car dans MLOBJ toutes les fonctions de création, d'accès et de modification sur ces structures sont définies comme des primitives pré-chargées dès le lancement. Elles ont donc un schéma de type dans l'environnement de typage initial, et seront prises en charge par la règle qui gère les identifieurs. Par exemple, l'environnement initial contient la liaison suivante pour l'opérateur cons **(::)** sur les listes :

$$\textbf{(::)} : \forall \{u, v\}.\ u\ ^{v} \rightarrow u\ list\ ^{0} \rightarrow u\ list$$

et ainsi de suite... La liste complète des fonctions prédéfinies avec leur type est donnée en Annexe A.6 [9].

[9] On peut y constater que les types donnés à **ref** et **(:=)** dans MLOBJ sont un peu plus permissifs que dans [Bou04]. A notre connaissance, ceci ne remet pas en cause la sûreté du système de types.

Chapitre 3

MlObj

Ce chapitre donne des exemples réels d'interaction avec l'interpréteur MlObj. A quelques différences près, la syntaxe utilisée se rapproche beaucoup de celle d'Objective Caml. Ce chapitre se présente comme un tutoriel, introduisant les principales possibilités offertes par l'interpréteur sur des exemples. Le lecteur est renvoyé au mini-manuel de référence en Annexe A pour une présentation complète de MlObj, sa syntaxe, ses fonctions prédéfinies, etc...

Dans la suite, les zones encadrées indiquent des interactions avec l'interprète. Les commandes entrées à la suite du prompt `>` sont en caractères droits. Les réponses de l'interprète sont en italiques[10]. Lors de l'évaluation d'une expression, cette réponse prend la forme "*`- : type = valeur`*" où *`valeur`* est le résultat de l'évaluation de l'expression et *`type`* son (schéma de) type inféré par le système (ce type peut éventuellement se prolonger sur les lignes suivantes s'il est récursif ou s'il contient des contraintes sur les degrés, comme on va le voir sur des exemples plus loin). Lors de l'évaluation d'une déclaration globale `let (rec) ...`, la réponse prend la forme "*`val nom : type = valeur`*" où *`nom`* est le nom de la variable stockée dans l'environnement global.

3.1 Exemples fonctionnels simples

Commençons par une définition simple : la fonction identité. On notera au passage la syntaxe utilisée pour les types, ici le type de retour correspond au $\forall\alpha.\ \alpha^0 \rightarrow \alpha$ inféré par le système de types (les variables de type polymorphes sont introduites par une apostrophe `'` suivie d'une lettre minuscule, les degrés sont introduits par un accent circonflexe `^`).

```
> let id = fun x -> x;;
val id : 'a ^0-> 'a = <fun>
```

Deuxième exemple simple : la fonction factorielle (on notera que le test d'égalité s'écrit `==` et non `=` comme en Caml).

```
> let rec fact n = if n == 0 then 1 else n * fact (n - 1);;
val fact : int ^0-> int = <fun>
> fact 5;;
- : int = 120
```

[10] Merci à Jens Klöcker pour son script `caml-tex` et son package `caml.sty`, que j'ai pu adapter à mes besoins, afin d'automatiser l'évaluation et la mise en page.

Un peu de manipulation de références : cette partie est strictement identique au code utilisé en OBJECTIVE CAML.

```
> let x = ref 0;;
val x : int ref = ref 0
> x := 1;;
- : unit = ()
> !x;;
- : int = 1
```

Définissons récursivement l'itérateur `map` sur les listes, à partir des fonctions prédéfinies `null`, `hd` et `tl` :

```
> let map f =
   let rec loop l = if null l then [] else f (hd l) :: loop (tl l)
   in loop;;
val map : ('a ^b-> 'c) ^d-> 'a list ^0-> 'c list = <fun>

> map (fun x -> x * x) [1, 2, 3];;
- : int list = [1, 4, 9]
```

A seule fin de vérifier que l'interpréteur génère bien des types récursifs quelconques, voici le typage pour $\Delta = \lambda x\ xx$. Les types de la forme `Xn` sont des abréviations (pas forcément récursives), dont le contenu est donné après le mot-clé `where`.

```
> let delta x = x x;;
val delta : X1 ^0-> 'a
where X1 = X1 ^b-> 'a
 = <fun>
```

3.2 Degrés et contraintes

Afin de mettre en évidence les contraintes sur les degrés, voici les fonctions application et composition de fonctions :

```
> fun f x -> f x;;
- : ('a ^b-> 'c) ^d-> 'a ^e-> 'c
where ^e<=^b  = <fun>
> fun f g x -> f (g x);;
- : ('a ^b-> 'c) ^d-> ('e ^f-> 'a) ^g-> 'e ^h-> 'c
where ^h<=^b ^h<=^f  = <fun>
```

On notera que l'inférence de types produit le type $(a^b \to c)^d \to a^e \to c$ dans le premier cas avec la contrainte supplémentaire $e \leqslant b$, contrainte indiquée par l'interpréteur par le mot-clé `where`.

Les instructions suivantes illustrent la mise en oeuvre des techniques exposées à l'Exemple 2.1. La fonction polymorphe `ignore` nous servira d'exemple (le caractère _ désigne un identifieur muet dont on ne se sert pas dans la suite). Dans le premier cas, `fun x -> ignore x`, le degré `^b` est bien

rendu polymorphe, et la contrainte `^b` $\leqslant$ `^c` où `^c` est produite par l'instanciation de `ignore` a bien été supprimée.

```
> let ignore _ = ();;
val ignore : 'a ^b-> unit = <fun>
> fun x -> ignore x;;
- : 'a ^b-> unit = <fun>
```

Dans le deuxième cas, on crée une référence `r` sur `ignore`, dont toutes les variables sont évidemment monomorphes (comme en OCaml, une variable monomorphe est précédée par un underscore _). Puis, on évalue `fun x -> !r x`. Comme prévu, le degré `^_b` n'est cette fois-ci pas généralisé, et on conserve la contrainte `^_b` $\leqslant$ `^_c` associée. Une consultation de `r` nous permet de constater que la même contrainte a été rajoutée (dans les deux réponses de l'interpréteur, le `^_b` de l'une correspond au `^_c` de l'autre).

```
> let r = ref ignore;;
val r : ('_a ^_b-> unit) ref = ref <fun>
> fun x -> !r x;;
- : '_a ^_b-> unit
where ^_b<=^_c  = <fun>
> r;;
- : ('_a ^_b-> unit) ref
where ^_c<=^_b  = ref <fun>
```

3.3 Enregistrements

Les deux exemples suivants montrent respectivement la syntaxe utilisée pour un enregistrement clos composé de trois champs (deux entiers et une chaîne de caractères), et pour une fonction qui prend un enregistrement et l'étend avec un nouveau champ `l`.

```
> let e = { posx = 0, posy = 1, couleur = "bleu" };;
val e : { couleur : string, posx : int, posy : int } = { couleur =
"bleu", posx = 0, posy = 1 }
> let etend = fun enr -> { enr, l = 0 };;
val etend : 'A ^0-> { 'A, l : int }
where 'A :: {l}  = <fun>
```

Dans le deuxième cas, on notera qu'une variable de rangée (ici polymorphe) est introduite par une apostrophe ' suivie d'une lettre majuscule, et que la liste de ses champs non autorisés est donnée à la suite du mot-clé `where`.

Les deux exemples suivants montrent respectivement la sélection et la restriction de champs :

```
> e.couleur;;
- : string = "bleu"
> (etend e) \ posy;;
- : { couleur : string, l : int, posx : int } = { couleur = "bleu", l =
0, posx = 0 }
```

Passons maintenant à des exemples plus concrets. La fonction `point` suivante définit une "classe" pour les points, contenant un champ position modifiable et une méthode pour bouger le point. Elle est paramétrée par une position initiale `x`.

```
> let point x self = { pos = ref x,
                       move y = self.pos := !self.pos + y };;
val point : 'a ^b-> { 'A, pos : int ref } ^c-> { move : int ^0-> unit,
pos : 'a ref }
where 'A :: {pos}  = <fun>
```

Afin de créer des instances de cette classe, nous avons besoin de la fonction `fix`, qui calcule le point fixe d'une fonction :

```
> let fix = fun f -> let rec x = f x in x;;
val fix : ('a ^1-> 'a) ^0-> 'a = <fun>
```

On notera, conformément à nos attentes, le degré `^1` dans le type. Ainsi, par unification, ceci nous assure que `fix` ne pourra être appliquée qu'à des fonctions qui protègent leur argument.

Il est maintenant possible de créer un nouveau point `p` et de le bouger :

```
> let p = fix (point 4);;
val p : { move : int ^0-> unit, pos : int ref } = { move = <fun>, pos =
ref 4 }
> p.move 2;;
- : unit = ()
> p;;
- : { move : int ^0-> unit, pos : int ref } = { move = <fun>, pos = ref
6 }
```

Les commandes suivantes créent une nouvelle classe qui hérite de `point` et qui lui ajoute une méthode pour remettre la position à 0, et l'instancient :

```
> let clearable_point x self = { point x self, clear _ = self.pos := 0 };;
val clearable_point : 'a ^b-> { 'A, pos : int ref } ^c-> { clear : 'd
^e-> unit, move : int ^0-> unit, pos : 'a ref }
where 'A :: {pos}  = <fun>
> let q = fix (clearable_point 5);;
val q : { clear : '_a ^_b-> unit, move : int ^0-> unit, pos : int ref }
= { clear = <fun>, move = <fun>, pos = ref 5 }
> q.clear ();;
- : unit = ()
> q;;
- : { clear : unit ^_a-> unit, move : int ^0-> unit, pos : int ref } = {
clear = <fun>, move = <fun>, pos = ref 0 }
```

Etant donné que dans notre langage, les "classes" n'existent pas en tant que telles, mais sont en fait des valeurs fonctionnelles standards, il est possible de définir directement une fonction qui

prend une super-classe et qui renvoie la classe héritière avec un champ `color` en plus. Par exemple, on peut l'appliquer aussi bien à la classe `point` qu'à sa sous-classe `clearable_point` :

```
> let color c super = fun x self -> { super x self, color = ref c };;
val color : 'a ^b-> ('c ^d-> 'e ^f-> 'A) ^g-> 'c ^h-> 'e ^i-> { 'A,
color : 'a ref }
where 'A :: {color} ^i<=^f  = <fun>
> fix ((color 256 point) 0);;
- : { color : int ref, move : int ^0-> unit, pos : int ref } = { color =
ref 256, move = <fun>, pos = ref 0 }
> fix ((color 256 clearable_point) 0);;
- : { clear : '_a ^_b-> unit, color : int ref, move : int ^0-> unit, pos
: int ref } = { clear = <fun>, color = ref 256, move = <fun>, pos = ref
0 }
```

En revanche, on peut vérifier qu'il n'est pas possible d'ajouter deux fois le champ `color`, le typage nous le signale par un échec :

```
> color 123 (color 256 point);;
Unable to unify 'A
where 'A :: {color}
and { color : int ref, move : int ^0-> unit, pos : 'a ref }
```

Revenons un moment sur la classe `point`. Le champ position `pos` est explicitement déclaré comme une référence. Est-il possible de s'en passer ? La nouvelle classe `point` ci-dessous suit cette idée. En contrepartie, puisqu'il n'est plus possible de modifier directement la *valeur* du champ, la méthode `move` renvoie `self` dans lequel on remplace le champ `pos` par sa nouvelle valeur.

```
> let point x self = { pos = x,
                       move d = { self, pos <- self.pos + d }};;
val point : 'a ^b-> { 'A, pos : int } ^c-> { move : int ^0-> { 'A, pos :
int }, pos : 'a }
where 'A :: {pos}  = <fun>
> (((fix (point 0)).move 3).move 2).pos;;
- : int = 2
```

Nous invitons le lecteur à comprendre pourquoi le résultat de l'évaluation est 2 et non pas 5 comme on le voudrait. Au moment du point fixe par `fix`, la valeur `self` dans `self.pos` est liée à l'enregistrement créé. Ceci a pour conséquence que lors du deuxième appel `move`, la valeur `self.pos` fait toujours référence à l'enregistrement initial, et non pas à celui retourné par le premier appel de `move`. Autrement dit, il faut prendre garde qu'avec ce style de programmation, le paramètre `self` ne désigne pas l'objet courant sur lequel on appelle la méthode comme c'est le cas dans les langages à objets usuels, mais l'objet au moment de sa création (et il serait donc préférable d'utiliser un autre nom que `self`...).

Nous reviendrons sur ce point au chapitre suivant, en apportant une solution dans le cadre plus général des mixins.

3.4 Simulation de paires

On pourra noter que notre langage ne propose ni les paires ni les tuples de façon générale. Cependant, il est facile de les encoder à l'aide des enregistrements. Les définitions suivantes donnent les fonctions nécessaires, à savoir la création de paires et la consultation de ses composantes.

```
> let pair x y = { fst = x, snd = y };;
val pair : 'a ^b-> 'c ^0-> { fst : 'a, snd : 'c } = <fun>
> let fst p = p.fst;;
val fst : { 'A, fst : 'a } ^0-> 'a
where 'A :: {fst}  = <fun>
> let snd p = p.snd;;
val snd : { 'A, snd : 'a } ^0-> 'a
where 'A :: {snd}  = <fun>
> fst (pair 1 2);;
- : int = 1
```

On pourra objecter à la vue de leur type que ces fonctions `fst` et `snd` peuvent prendre en argument bien plus que des véritables paires : en fait n'importe quel enregistrement qui contient le champ `fst` ou `snd`. Par exemple :

```
> snd { x = 0, snd = "boo", f x = x + 1 };;
- : string = "boo"
```

Si on désire limiter l'usage de `fst` et `snd` aux paires créées par `pair`, il est possible de limiter le type de leur argument en l'indiquant explicitement :

```
> let fst (p : { fst:'a, snd:'b }) = p.fst;;
val fst : { fst : 'a, snd : 'b } ^0-> 'a = <fun>
> let snd (p : { fst:'a, snd:'b }) = p.snd;;
val snd : { fst : 'a, snd : 'b } ^0-> 'b = <fun>
```

Les deux fonctions ont alors le type souhaité, et l'exemple ci-dessus n'est plus possible [11]. De façon générale, on peut apporter des contraintes sur le type d'un identifieur ou d'une expression entière par `(x : type)` ou `(expr : type)`. Au moment de l'inférence, le type le plus générique inféré est alors unifié avec `type`.

[11] Une autre façon de simuler les paires serait de construire une classe avec comme méthodes les projections.

Chapitre 4

Mixins et exemples avancés

Dans ce Chapitre, nous introduisons une couche syntaxique supplémentaire à notre langage, formellement puis sur des exemples, afin de démontrer sa puissance expressive, aussi bien opérationnellement que du point de vue du typage. Cet ajout implémente un langage pour la programmation orientée objet à base de *mixins*. Grossièrement, les mixins peuvent être vus comme des définitions de classes paramétrées par leur super-classe ; ils furent introduits pour la première fois dans des langages orientés objets basés sur Lisp dans les années 80 (on pourra également trouver quelques similarités avec les classes paramétrées d'Eiffel [Mey86] ou les "classes virtuelles" de Beta [MP89]). Dans [BC90], les mixins sont présentés comme la "brique de base" pour l'héritage, point de vue que nous ne pouvons que partager ; des travaux plus récents s'y sont également à nouveau intéressé [AZ98, FKF98, BPSM99a].

4.1 Présentation générale

Nous commençons par donner quelques éléments de terminologie.

Un *générateur* est une fonction $Gen = \lambda\ self.\{\ldots\}$ qui prend en paramètre $self$ et retourne un enregistrement (ce que nous appelions "classes" dans le chapitre précédent étaient donc des générateurs paramétrés par des valeurs initiales). Informellement, un *objet* est alors le point fixe d'un générateur (voir [CP89]) : (`fix` *Gen*), autrement dit un enregistrement récursif (comme c'est le cas dans [Sny86, Car88, CHC94, Wan94]). Pour qu'un générateur puisse générer un objet par point fixe, il faut que le type de $self$ soit unifiable avec le type de retour du générateur, et en outre que ce générateur soit une fonction protectrice pour $self$.

Un *mixin* est une fonction qui transforme un générateur en un autre générateur. Elle est donc de la forme $\lambda g \lambda s\ \{\ldots\}$. En règle générale, l'action d'un mixin sera de modifier le générateur donné en paramètre en lui ajoutant, retranchant ou modifiant des champs.

Un *champ* de mixin est un champ au sens habituel des enregistrements, contenant une valeur soit non modifiable, soit modifiable (c'est-à-dire une référence sur cette valeur).

Une *méthode* est également un champ au sens des enregistrements. Afin de respecter le système de types, il faudra que ce soit toujours une fonction. Pour cette raison, une méthode prendra toujours un paramètre unit `()` comme premier argument (une telle fonction est appelée un "thunk"), afin de geler l'évaluation de la méthode lors du calcul de point fixe. L'appel de méthode devra en conséquence passer `()` en premier argument pour dégeler la fonction. De plus, deux paramètres particuliers sont accessibles dans le corps des méthodes : `super` et `self` (ce ne sont pas des mots-clés, mais bien des identifieurs soumis à l'α-conversion). Lors de l'évaluation de la méthode, `self` sera lié à l'objet courant en tant qu'instance de la classe courante, et `super` à l'objet courant en

tant qu'instance de la super-classe (ceci est nécessaire dans le cas où l'on redéfinit une méthode qui a besoin d'appeler son ancienne version, voir les exemples plus loin).

Une *classe* est une fonction paramétrée par des paramètres d'instance (typiquement des valeurs initiales), et retournant un mixin. Elle a donc la forme générale

$$\lambda x_1 \dots x_n \ \lambda g \ \lambda s \ \{\dots \text{champs} \dots \text{méthodes} \dots\}$$

Pour construire une *instance* d'une classe C, il faut lui fournir, outre les paramètres d'instance, un générateur de départ g. Dans le cas standard, il s'agira du générateur vide $\lambda s \ \{\}$ (mais pour d'autres usages, on peut imaginer d'autres générateurs de base ; ceci permet par exemple de définir des champs ou des méthodes qui apparaissent systématiquement dans les objets créés). La forme générale d'une instance de C est donc :

$$\texttt{fix} \ (C \ N_1 \cdots N_n \ (\lambda s \ \{\}))$$

où les N_i sont les paramètres d'instance.

Pour cette raison, nous définirons la fonction `new` suivante :

$$\texttt{new} = \lambda m \ \ \texttt{fix} \ (m \ (\lambda s \ \{\}))$$

et on écrira

$$\texttt{new} \ (C \ N_1 \cdots N_n)$$

pour la création d'une instance.

4.2 Sucre syntaxique pour les mixins

Nous allons mettre en oeuvre le modèle des mixins par l'intermédiaire de constructions syntaxiques spécifiques, qui sont automatiquement transformées dans le langage de base des enregistrements.

Un mixin est délimité par les mots-clés `mixin` et `end`. Son contenu est une suite d'instructions introduites chacune par un mot-clé spécifique. Il n'y a pas de délimiteur (les mots-clés suffisent). Les instructions sont parcourues dans l'ordre où elles sont données, et modifient le mixin au fur et à mesure. Intuitivement, si g est le générateur donné en paramètre au mixin (au mot-clé `mixin`), ces instructions font des modifications sur g et retournent un générateur final g' (au niveau du `end`), qui est la valeur de retour du mixin.

Nous allons maintenant donner la fonction de traduction $[\![M]\!]$ pour un mixin M. On prendra garde au fait que, dans les définitions qui suivent, M représente une séquence éventuellement vide d'expressions.

$$[\![\textbf{mixin} \ M \ \textbf{end}]\!] = [\![M]\!]$$

Commençons par la traduction des champs. Les mots-clés `var` et `cst` introduisent des champs modifiables et constants respectivement :

$$[\![M \ \textbf{var} \ l = N]\!] = \lambda g \lambda s \ \{[\![M]\!] \ g \ s, \ l = \ \textbf{ref} \ [\![N]\!]\}$$
$$[\![M \ \textbf{cst} \ l = N]\!] = \lambda g \lambda s \ \{[\![M]\!] \ g \ s, \ l = [\![N]\!]\}$$

Ici, puisque M désigne le mixin construit jusqu'ici, $[\![M]\!] \ g \ s$ est l'enregistrement qui lui correspond pour un générateur g et un paramètre self s donnés. Il suffit donc d'étendre cet enregistrement avec

le champ correspondant (par le biais d'une référence pour les champs modifiables) et de rendre le mixin correspondant.

Les méthodes sont introduites par le mot-clé `meth` et existent sous deux formes : définition d'une nouvelle méthode avec $=$ et redéfinition d'une méthode existante avec $\leftarrow$. Le nom de la méthode l est obligatoirement suivi de deux identifieurs x et y (dont le choix est libre), correspondant à `super` et `self` respectivement.

$$[\![M \textbf{ meth } l(y,x) \ = N]\!] = \lambda g \lambda s \ (\textbf{let } z = [\![M]\!] \ g \ s \textbf{ in } \{z, l = \lambda(u : unit) \ (\lambda y \lambda x \ [\![N]\!]) \ z \ s\})$$
$$[\![M \textbf{ meth } l(y,x) \ \leftarrow N]\!] = \lambda g \lambda s \ (\textbf{let } z = [\![M]\!] \ g \ s \textbf{ in } \{z \setminus l, l = \lambda(u : unit) \ (\lambda y \lambda x \ [\![N]\!]) \ z \ s\})$$

Comme précédemment, $z = [\![M]\!] \ g \ s$ désigne l'enregistrement créé jusqu'ici, que l'on étend par un nouveau champ l. Ce champ est un thunk (une fonction dont le paramètre est forcé à *unit*) dont le corps est $(\lambda y \lambda x \ [\![N]\!]) \ z \ s$, autrement dit le corps de la méthode $[\![N]\!]$, dans lequel y et x sont liés à z et s respectivement. Au moment de la création de l'objet, s sera bien l'objet lui-même, et z l'enregistrement construit jusqu'à ce point. Pour une redéfinition de méthode, la traduction est la même, sauf que l'ancien champ l est d'abord enlevé de z avant de lui être rajouté.

La présence du thunk nous oblige à prendre pour l'appel de méthode une syntaxe différente de celle de la sélection de champs, puisqu'il faut lui passer une valeur de type *unit* :

$$[\![M \# l]\!] = ([\![M]\!].l) \ ()$$

Le mot-clé `inherit` introduit l'*héritage* : toutes les transformations induites par le mixin N sont reprises à cet endroit.

$$[\![M \textbf{ inherit } N]\!] = \lambda g \lambda s \ ([\![N]\!] \ ([\![M]\!] \ g) \ s)$$

Plus précisément, le générateur g passé en paramètre du mixin-résultat est d'abord donné à $[\![M]\!]$, qui nous renvoie un générateur g', que l'on passe ensuite à $[\![N]\!]$. On notera que cette traduction se transforme en $[\![N]\!] \circ [\![M]\!]$ par η-expansion. L'héritage se résume donc à de la composition de mixins ! Nous gardons cependant la première traduction car cette deuxième version produit un type trop générique.

Les mots-clés `without` et `rename ... as` permettent respectivement de supprimer et de renommer un champ ou une méthode d'un mixin. Au vu des cas précédents, leur traduction ne pose pas de problème :

$$[\![M \textbf{ without } l]\!] = \lambda g \lambda s \ ([\![M]\!] \ g \ s) \setminus l$$
$$[\![M \textbf{ rename } l \textbf{ as } l']\!] = \lambda g \lambda s \ (\textbf{let } x = [\![M]\!] \ g \ s \textbf{ in } \{x \setminus l, l' = x.l\})$$

Enfin, pour être complet, il reste à définir le mixin "identité" qui sert de point de départ :

$$[\![\ ]\!] = \lambda g \lambda s \ (g \ s)$$

Un mot sur le typage On pourra vérifier sur les règles de traduction ci-dessus qu'un mixin $[\![M]\!]$ protège toujours son deuxième argument s (à condition que ce soit le cas de N dans M **inherit** N) ; ceci pourrait être faux pour les méthodes s'il n'y avait pas le thunk. Il est donc toujours possible de calculer un objet par point fixe, au moins en ce qui concerne les degrés (l'encodage des mixins a été conçu en ce sens).

En revanche, tout mixin n'est pas forcément instanciable : il faut (et il suffit) pour cela que le type du deuxième argument s soit unifiable avec le type de retour du mixin lorsque le générateur donné en premier argument a pour type de retour l'enregistrement vide $\{\}$. La condition nécessaire

et suffisante pour pouvoir appliquer `new` sera donc "lisible" sur le type du mixin (en pratique, la lecture des types générés par les mixins commence à être difficile ; mais ils restent indispensables en cas d'erreur). Nous verrons dans les exemples quels mixins sont instanciables et lesquels ne le sont pas (à juste titre). Ces derniers n'ont alors qu'un rôle de "modifieurs" pour d'autres mixins (un peu à la manière des classes virtuelles dans les langages à objets).

4.3 Exemples dans MlObj

La sur-couche syntaxique et la traduction présentée ci-dessus sont intégrées telles qu'elles dans MlObj. Nous allons maintenant présenter quelques exemples d'utilisation et tenter de donner la mesure de la puissance expressive apportée par ce formalisme.

Avant toute chose, il nous faut définir la fonction `new` qui comme `fix` ne fait pas partie intégrante du langage[12] :

```
> let new m = fix (m (fun s -> {}));;
val new : (('a ^b-> { }) ^c-> 'd ^1-> 'd) ^0-> 'd = <fun>
```

Commençons par notre exemple préféré, exprimé comme un mixin :

```
> let point x =
   mixin
    var pos = x
    meth move(super,self) d = self.pos := !self.pos + d
   end;;
val point : 'a ^b-> (X1 ^c-> 'A) ^d-> X1 ^e-> { 'A, move : unit ^f-> int
^0-> unit, pos : 'a ref }
where X1 = { 'B, pos : int ref }
'B :: {pos} 'A :: {pos,move} ^e<=^c  = <fun>
```

Et un exemple d'utilisation :

```
> let p = new (point 5);;
val p : { move : unit ^_a-> int ^0-> unit, pos : int ref } = { move =
<fun>, pos = ref 5 }
> p#move 2;;
- : unit = ()
```

Le mixin suivant définit une classe contenant une couleur et une méthode pour la modifier :

[12] La raison est que d'autres variantes sont possibles (voir par exemple `new_init` plus loin), et que MlObj étant un langage expérimental, nous préférons laisser un maximum de libertés.

```
> let coloring c =
   mixin
    var color = c
    meth paint(super,self) c' = self.color := c'
   end;;
val coloring : 'a ^b-> (X1 ^c-> 'A) ^d-> X1 ^e-> { 'A, color : 'a ref,
paint : unit ^f-> 'g ^h-> unit }
where X1 = { 'B, color : 'g ref }
'B :: {color} 'A :: {color,paint} ^e<=^c  = <fun>
```

A partir de ces deux mixins, il est possible de construire une classe “point colorée” par héritage multiple :

```
> let colorPoint x c =
   mixin
    inherit point x
    inherit coloring c
   end;;
val colorPoint : 'a ^b-> 'c ^d-> (X1 ^e-> 'A) ^f-> X1 ^g-> { 'A, color :
'c ref, move : unit ^h-> int ^0-> unit, paint : unit ^i-> 'j ^k-> unit,
pos : 'a ref }
where X1 = { 'B, color : 'j ref, pos : int ref }
'B :: {color,pos} 'A :: {color,paint,pos,move} ^g<=^e  = <fun>
```

A noter que les deux instructions `inherit` pourraient ici être interverties sans changer le résultat.

```
> let cp = new (colorPoint 42 255);;
val cp : { color : int ref, move : unit ^_a-> int ^0-> unit, paint :
unit ^_b-> int ^_c-> unit, pos : int ref } = { color = ref 255, move =
<fun>, paint = <fun>, pos = ref 42 }
> cp#move 3;;
- : unit = ()
> cp#paint 128;;
- : unit = ()
```

Le mixin `reset` suivant définit uniquement une méthode qui réinitialise le champ `pos`.

```
> let reset =
   mixin
    meth reset(super,self) = self.pos := 0
   end;;
val reset : (X1 ^a-> 'A) ^b-> X1 ^c-> { 'A, reset : unit ^d-> unit }
where X1 = { 'B, pos : int ref }
'B :: {pos} 'A :: {reset} ^c<=^a  = <fun>
```

A noter que ce mixin est bien typable, alors qu'il ne contient pas de champ `pos` défini... En revanche, comme nous l'avons expliqué plus haut et comme on peut le constater sur son type, il n'est pas instanciable. La seule utilisation que nous pourrons en faire afin d'arriver à une classe

instanciable sera donc d'en hériter dans un autre mixin qui aura un champ `pos` défini par ailleurs. Néanmoins, l'intérêt est que nous pourrons le réutiliser dans des situations différentes sans avoir à récrire le code de la méthode `reset`. Par exemple, les deux classes suivantes respectent les conditions nécessaires et suffisantes décrites plus haut, et sont donc instanciables :

```
> let resetPoint x =
   mixin
    inherit point x
    inherit reset
   end;;
val resetPoint : 'a ^b-> (X1 ^c-> 'A) ^d-> X1 ^e-> { 'A, move : unit
^f-> int ^0-> unit, pos : 'a ref, reset : unit ^g-> unit }
where X1 = { 'B, pos : int ref }
'B :: {pos} 'A :: {reset,pos,move} ^e<=^c  = <fun>

> let resetColorPoint x c =
   mixin
    inherit colorPoint x c
    inherit reset
   end;;
val resetColorPoint : 'a ^b-> 'c ^d-> (X1 ^e-> 'A) ^f-> X1 ^g-> { 'A,
color : 'c ref, move : unit ^h-> int ^0-> unit, paint : unit ^i-> 'j
^k-> unit, pos : 'a ref, reset : unit ^l-> unit }
where X1 = { 'B, color : 'j ref, pos : int ref }
'B :: {color,pos} 'A :: {reset,color,paint,pos,move} ^g<=^e  = <fun>
```

On peut aller encore plus loin et définir un mixin qui fait appel à des méthodes définies dans une hypothétique super-classe :

```
> let wrapper =
   mixin
    meth reset(super,self) d <- super#reset; super#paint d
   end;;
val wrapper : ('a ^b-> { 'A, paint : X1, reset : unit ^c-> unit }) ^d->
'a ^e-> { 'A, paint : X1, reset : unit ^f-> 'g ^0-> 'h }
where X1 = unit ^i-> 'g ^j-> 'h
'A :: {paint,reset} ^e<=^b  = <fun>
```

Grâce au typage, ce mixin ne pourra être repris dans une classe instanciable que s'il est dans l'environnement adéquat, à savoir avec une super-classe qui contient des méthodes `reset` et `paint`. Par exemple, puisque `resetColorPoint` vérifie cette condition, on pourra instancier la classe suivante :

```
> let resetColorPoint2 x c =
   mixin
    inherit resetColorPoint x c
    inherit wrapper
   end;;
val resetColorPoint2 : 'a ^b-> 'c ^d-> (X1 ^e-> 'A) ^f-> X1 ^g-> { 'A,
color : 'c ref, move : unit ^h-> int ^0-> unit, paint : unit ^i-> 'j
^k-> unit, pos : 'a ref, reset : unit ^l-> 'j ^0-> unit }
where X1 = { 'B, color : 'j ref, pos : int ref }
'B :: {color,pos} 'A :: {reset,color,paint,pos,move} ^g<=^e  = <fun>
```

Nous allons maintenant montrer comment il est possible d'exécuter automatiquement une procédure au moment de l'instanciation. La technique est simple : il suffit de rajouter une méthode `init` aux classes pour lesquelles on souhaite ce comportement et de les instancier par la fonction `new_init` ci-dessous. Cette fonction crée l'objet de façon standard par la méthode `new`, appelle sa méthode `init`, puis supprime cette méthode avant de renvoyer l'objet (ce qui évite de la réexécuter par erreur).

```
> let new_init m =
   let o = new m in
   o#init; o\init;;
val new_init : (('a ^b-> { }) ^c-> X1 ^1-> X1) ^0-> 'A
where X1 = { 'A, init : unit ^d-> unit }
'A :: {init}  = <fun>
```

En guise d'exemple, nous donnons une variante de la classe `point` qui imprime un message à l'écran lors de la création d'un nouveau point :

```
> let printable_point x =
   mixin
    inherit point x
    meth print(super,self) = print_int !self.pos
    meth init(super,self) = print_string "new point at "; self#print;
                            print_newline()
   end;;
val printable_point : 'a ^b-> (X1 ^c-> 'A) ^d-> X1 ^e-> { 'A, init :
unit ^f-> unit, move : unit ^g-> int ^0-> unit, pos : 'a ref, print :
unit ^h-> unit }
where X1 = { 'B, pos : int ref, print : unit ^i-> unit }
'B :: {print,pos} 'A :: {pos,move,print,init} ^e<=^c  = <fun>
> let p = new_init (printable_point 17);;
new point at 17
val p : { move : unit ^_a-> int ^0-> unit, pos : int ref, print : unit
^_b-> unit } = { move = <fun>, pos = ref 17, print = <fun> }
```

On vérifie que l'objet ne contient plus de méthode `init` au final. Si on souhaite utiliser des initialiseurs pour des classes qui héritent l'une de l'autre, il faut prendre garde à ce que lors de la

redéfinition des méthodes `init`, on commence par appeler la méthode `init` de la super-classe :

```
meth init(super,self) <- super#init ; ...
```

On peut même imaginer de généraliser cette technique à toutes les classes et faire en sorte que tous les mixins possèdent une méthode `init`. Il suffit pour cela d'utiliser la fonction suivante à la place de `new`. Cette fonction fournit comme générateur de base à tous les mixins un générateur qui renvoie un enregistrement avec une fonction `init` qui ne fait rien :

```
> let new_init m =
   let o = fix (m (fun s -> { init _ = () })) in
   o#init; o\init;;
val new_init : (('a ^b-> { init : 'c ^d-> unit }) ^e-> X1 ^1-> X1) ^0->
'A
where X1 = { 'A, init : unit ^f-> unit }
'A :: {init}  = <fun>
```

Les mixins associés au types récursifs permettent d'imaginer encore de nombreuses combinaisons, par exemple des mixins dont une méthode retourneraient `self` ou même un autre mixin. Tout cela est rendu possible par le fait que les mixins ont le statut de simple valeurs.

```
> let m =
   mixin
    meth another(super,self) = mixin
                                meth myself(super,self) = self
                               end
   end;;
val m : ('a ^b-> 'A) ^c-> 'a ^d-> { 'A, another : unit ^e-> ('f ^g-> 'B)
^h-> 'f ^i-> { 'B, myself : unit ^j-> 'f } }
where 'B :: {myself} 'A :: {another} ^d<=^b ^i<=^g  = <fun>
> (new ((new m)#another))#myself;;
- : X1
where X1 = { myself : unit ^_a-> X1 }
 = { myself = <fun> }
```

A l'aide d'une classe récursive, il est également possible de définir une méthode pour cloner un objet. Dans l'exemple suivant, la méthode `clone` retourne une instance fraîche en gardant la même position pour le point. Cette classe contient aussi une méthode `colorless` qui supprime toutes les informations relatives à la couleur (afin de les cacher par exemple).

```
> let rec clonablePoint x c =
   mixin
    inherit colorPoint x c
    meth clone(z,s) = new (clonablePoint !s.pos !s.color)
    meth colorless(z,s) = s \ color \ paint
   end;;
val clonablePoint : int ^a-> 'b ^c-> (X1 ^1-> { }) ^d-> X1 ^1-> X1
where X4 = unit ^e-> int ^0-> unit
X3 = unit ^f-> { clone : X2, colorless : X3, move : X4, pos : int ref }
X2 = unit ^g-> X1
X1 = { clone : X2, color : 'b ref, colorless : X3, move : X4, paint :
unit ^h-> 'b ^i-> unit, pos : int ref }
 = <fun>
```

Comme prévu, une modification sur le point de départ n'affecte pas son clone (et inversement) :

```
> let p = new (clonablePoint 10 256) in
   let q = p#clone in
    p#move 5; q#colorless;;
- : X1
where X4 = unit ^_a-> int ^0-> unit
X3 = unit ^_b-> X1
X2 = unit ^_c-> { clone : X2, color : int ref, colorless : X3, move :
X4, paint : unit ^_d-> int ^_e-> unit, pos : int ref }
X1 = { clone : X2, colorless : X3, move : X4, pos : int ref }
 = { clone = <fun>, colorless = <fun>, move = <fun>, pos = ref 10 }
```

Il est également possible de programmer des interactions du type "sujet/observateur". Dans l'exemple suivant, une fenêtre de la classe `window` est déclarée en tant que sujet d'un observateur `o`. Lorsqu'elle est déplacée, elle notifie l'observateur en appelant sa méthode `moved` et en se passant elle-même en paramètre. Ici, le comportement de l'observateur (en l'occurrence un gestionnaire de fenêtres) sera d'appeler une méthode de la fenêtre pour qu'elle se redessine.

```
> let subject o =
   mixin
    meth notify(z,s) y = y o s
   end;;
val subject : 'a ^b-> ('c ^d-> 'A) ^e-> 'c ^f-> { 'A, notify : unit ^g->
('a ^h-> 'c ^i-> 'j) ^0-> 'j }
where 'A :: {notify} ^f<=^d  = <fun>

> let window x o =
   mixin
    inherit point x
    inherit subject o
    meth move(z,s) d <- z#move d; s#notify (fun o -> o#moved)
    meth draw(z,s) = print_string "Je me redessine en x = ";
                     print_int !s.pos; print_newline ()
   end;;
val window : 'a ^b-> 'c ^d-> (X1 ^e-> 'A) ^f-> X1 ^g-> { 'A, draw : unit
^h-> unit, move : unit ^i-> int ^0-> 'j, notify : unit ^k-> ('c ^l-> {
'B, notify : unit ^m-> ({ 'C, moved : unit ^n-> 'o } ^0-> 'o) ^p-> 'j,
pos : int ref } ^q-> 'r) ^0-> 'r, pos : 'a ref }
where X1 = { 'B, notify : unit ^m-> ({ 'C, moved : unit ^n-> 'o } ^0->
'o) ^p-> 'j, pos : int ref }
'C :: {moved} 'B :: {notify,pos} 'A :: {notify,pos,move,draw} ^g<=^e  =
<fun>
```

```
> let manager =
   mixin
    meth moved(z,s) w = w#draw
   end;;
val manager : ('a ^b-> 'A) ^c-> 'a ^d-> { 'A, moved : unit ^e-> { 'B,
draw : unit ^f-> 'g } ^0-> 'g }
where 'B :: {draw} 'A :: {moved} ^d<=^b  = <fun>

> let m = new manager in
   let w = new (window 42 m) in
    w#move 10;;
Je me redessine en x = 52
- : unit = ()
```

En guise de dernier exemple, nous allons traiter la question des méthodes binaires. La classe suivante définit des points qui possèdent une méthode permettant de les comparer entre eux.

```
> let pointEq x =
   mixin
    inherit point x
    meth eq(super,self) p = !self.pos == !p.pos
   end;;
val pointEq : 'a ^b-> (X1 ^c-> 'A) ^d-> X1 ^e-> { 'A, eq : unit ^f-> {
'B, pos : int ref } ^0-> bool, move : unit ^g-> int ^0-> unit, pos : 'a
ref }
where X1 = { 'C, pos : int ref }
'C :: {pos} 'B :: {pos} 'A :: {pos,move,eq} ^e<=^c  = <fun>
> let p = new (pointEq 0);;
val p : { eq : unit ^_a-> { '_A, pos : int ref } ^0-> bool, move : unit
^_b-> int ^0-> unit, pos : int ref }
where '_A :: {pos}  = { eq = <fun>, move = <fun>, pos = ref 0 }
> let q = new (pointEq 1);;
val q : { eq : unit ^_a-> { '_A, pos : int ref } ^0-> bool, move : unit
^_b-> int ^0-> unit, pos : int ref }
where '_A :: {pos}  = { eq = <fun>, move = <fun>, pos = ref 1 }
> p#eq q;;
- : bool = false
```

Jusqu'ici tout se passe bien. On peut cependant constater que le type de `p` (et `q`) a été précisé suite à l'appel de la méthode `eq`. Ceci est dû au fait que `p` n'étant pas polymorphe, toutes ses variables restent monomorphes. Et l'appel à `eq` en instancie certaines.

```
> p;;
- : { eq : unit ^_a-> { eq : unit ^_b-> { '_A, pos : int ref } ^0->
bool, move : unit ^_c-> int ^0-> unit, pos : int ref } ^0-> bool, move :
unit ^_d-> int ^0-> unit, pos : int ref }
where '_A :: {pos}  = { eq = <fun>, move = <fun>, pos = ref 0 }
```

Si on appelle `p` sur lui-même, on va encore préciser certaines des variables dans son type.

```
> p#eq p;;
- : bool = true
> p;;
- : { eq : unit ^_a-> { eq : unit ^_a-> { eq : X1, move : X2, pos : int
ref } ^0-> bool, move : unit ^_b-> int ^0-> unit, pos : int ref } ^0->
bool, move : unit ^_b-> int ^0-> unit, pos : int ref }
where X2 = unit ^_b-> int ^0-> unit
X1 = unit ^_a-> { eq : X1, move : X2, pos : int ref } ^0-> bool
 = { eq = <fun>, move = <fun>, pos = ref 0 }
```

En revanche, alors que la classe `colorPoint` hérite de la classe `point` et contient aussi un champ `pos`, il n'est pas possible de l'utiliser comme argument de la méthode `eq` :

```
> p#eq (new (colorPoint 1 256));;
Unable to unify { eq : unit ^a-> { eq : X1, move : X2, pos : int ref }
^0-> bool, move : unit ^b-> int ^0-> unit, pos : int ref }
where X2 = unit ^b-> int ^0-> unit
X1 = unit ^a-> { eq : X1, move : X2, pos : int ref } ^0-> bool

and { color : int ref, move : unit ^c-> int ^0-> unit, paint : unit ^d->
int ^e-> unit, pos : int ref }
```

Par contre, si on recrée un nouveau point, l'instruction précédente devient possible. Mais par contraste, il n'est plus possible de comparer le point à lui-même...

```
> let p = new (pointEq 0);;
val p : { eq : unit ^_a-> { '_A, pos : int ref } ^0-> bool, move : unit
^_b-> int ^0-> unit, pos : int ref }
where '_A :: {pos}  = { eq = <fun>, move = <fun>, pos = ref 0 }
> p#eq (new (colorPoint 1 256));;
- : bool = false
> p#eq p;;
Unable to unify { color : int ref, move : unit ^a-> int ^0-> unit, paint
: unit ^b-> int ^c-> unit, pos : int ref }
and { eq : unit ^d-> { color : int ref, move : unit ^a-> int ^0-> unit,
paint : unit ^b-> int ^c-> unit, pos : int ref } ^0-> bool, move : unit
^e-> int ^0-> unit, pos : int ref }
```

En conclusion, les méthodes binaires sont utilisables, mais elles interagissent mal avec l'héritage [13]. Plus précisément, en MlObj, pour une méthode binaire d'un objet `o` donné, il est nécessaire de n'utiliser en argument que des instances issues d'une même classe (qui n'est pas forcément celle de `o`). Il est probable que l'utilisation de méthodes polymorphes dans les objets permettraient de résoudre cette question (entre autres), mais leur mise en oeuvre dans le système de types avec degrés n'est pas évidente... Une solution intermédiaire serait d'utiliser des types avec intersection...

[13] Pour une étude complète des problèmes posés par les méthodes binaires dans les langages de programmation orientés objets et quelques techniques permettant d'y remédier, on pourra se reporter avec intérêt à [BCC+95].

Chapitre 5

Une machine abstraite pour la récursion

Dans ce chapitre, nous proposons une machine abstraite pour le cœur de notre langage, c'est-à-dire pour un λ-calcul avec appel par valeur, étendu par un point fixe également en appel par valeur, et nous montrons que cette machine respecte la sémantique du langage. Ces travaux ont été publiés dans [BZ02].

5.1 Préliminaires

Afin d'éviter toute confusion, nous rappelons la syntaxe du cœur de langage sur lequel nous allons travailler :

$$\begin{array}{rcll} M, N & ::= & V \mid (MN) \mid \mu x M & \text{expressions} \\ V & ::= & x \mid \lambda x M & \text{valeurs} \end{array}$$

Ce langage ne contient plus d'enregistrements, car comme nous l'avons vu au Chapitre 1, les difficultés liées à l'évaluation du point fixe (et au typage par les degrés) résident dans le cœur fonctionnel, et non dans les constructions objets.

Dans cette partie, nous utilisons un constructeur de point fixe $\mu x M$ plutôt qu'un opérateur **let rec**. Néanmoins, ceci ne change pas grand chose au niveau de la théorie car ils sont interchangeables : $\mu x M$ est équivalent à **let rec** $x = M$ **in** x, et **let rec** $x = M$ **in** N est équivalent à $((\lambda x N)\ (\mu x M))$. La règle de typage correspondante s'adapte facilement de celle du Chapitre 1.

La sémantique de ce langage est donnée par la relation de réduction déterministe suivante :

$$\begin{array}{rcl} ((\lambda x M)\ V) & \to & \{^{V}/_{x}\}(M) \\ \mu x V & \to & \{^{\mu x V}/_{x}\}(V) \\ M \to M' & \Longrightarrow & \mathbf{E}[M] \to \mathbf{E}[M'] \end{array}$$

où les contextes d'évaluation $\mathbf{E}$ sont donnés par la grammaire :

$$\mathbf{E} ::= [] \mid (\mathbf{E}N) \mid (V\mathbf{E}) \mid \mu x \mathbf{E}$$

Bien entendu, la machine ne fonctionnera correctement que pour des termes dont l'évaluation est sans danger, autrement dit pour des termes typables. Le seul résultat dont nous aurons en fait besoin est le lemme suivant, qui a été démontré dans [Bou04].

Lemme 5.1 *Si $\mu x M$ est typable et $M \to^* \mathbf{E}[x]$, où x n'est pas liée dans $\mathbf{E}$, alors $\mathbf{E} = \mathbf{E}'[(\lambda y N[])]$, et $\lambda y N$ protège son argument.*

5.2 La machine abstraite

Notre machine est semblable aux machines à pile classiques utilisées pour l'évaluation de λ-expressions.

$$\mathcal{M} ::= (S, \sigma, M, \xi)$$

En tant que telle, elle se compose :
- d'une *pile de contrôle* S, pour conserver le contexte d'évaluation courant ;
- d'un *environnement* σ, qui contient les valeurs des variables libres apparaissant dans le code ;
- d'un *code* M, autrement dit la sous-expression en cours d'évaluation ;
- d'une *mémoire* ξ, qui contient les valeurs des expressions évaluées récursivement.

Dans les définitions, on dénotera par $A \in \mathcal{A}$ les *réponses* possibles de la machine, encore appelées les *valeurs* (à ne pas confondre avec V ci-dessus). Pour notre langage réduit, cet ensemble se réduit aux fonctions :

$$A ::= \lambda x M$$

La pile se compose de "*frames*" (ou *blocs d'activation*), des contextes d'évaluation élémentaires (nous suivons la présentation de [Pit02]). Il existe deux types de frames pour l'évaluation des applications, et un pour celle des récursions. Comme c'est souvent le cas, les substitutions de variables par des termes ne seront pas faites explicitement ; à la place, la machine gère des *clôtures* [Lan64], des paires formées d'un environnement et d'une expression, notées σM([14]).

$$\begin{array}{lcl} S & ::= & \varepsilon \mid S :: \mathbf{F} \\ \mathbf{F} & ::= & ([]\, \sigma N) \mid (\sigma A\, []) \mid \mu \ell\, [] \end{array}$$

En ce qui concerne la récursion, notre machine utilise l'idée de point fixe impératif de Landin : pour évaluer $\mu x M$, on alloue une nouvelle adresse ℓ avec une valeur "vide" (noté $\xi \uplus \{\ell \mapsto \bullet\}$), puis on évalue M dans un environnement qui lie x à ℓ, et on assigne à ℓ le résultat de cette évaluation (noté $\xi[\ell := \sigma A]$). Dans le cas où M est déjà une valeur, comme il n'y a rien à évaluer, il est possible de gagner en efficacité et de faire appel à une stratégie plus directe, qui lie x à M de façon circulaire dans l'environnement (voir le paragraphe suivant).

$$\xi ::= \varnothing \mid \xi \uplus \{\ell \mapsto \bullet\} \mid \xi[\ell := \sigma A]$$

Un environnement est une suite de liaisons portant sur des variables libres du code.

$$\sigma, \rho ::= \varepsilon \mid \sigma :: \{x \mapsto \rho A\} \mid \sigma :: \{x \mapsto A\} \mid \sigma :: \{x \mapsto \ell\}$$

Il s'agit soit d'une liaison classique $\sigma :: \{x \mapsto \rho A\}$ liant x à une clôture, soit d'une liaison vers une adresse mémoire $\sigma :: \{x \mapsto \ell\}$, soit d'une *liaison récursive* $\sigma :: \{x \mapsto A\}$ où la valeur A a ses variables liées dans σ (sauf en ce qui concerne x). Nous aurions pu utiliser une syntaxe plus explicite, du style

$$\sigma :: \{\text{rec } x \mapsto \rho A\}$$

mais il s'avère qu'un invariant de notre machine est que lorsqu'une telle configuration apparaît, on a toujours $\rho = \sigma$. Nous pouvons donc nous passer de la composante ρ, et par conséquent de l'indication explicite qu'il s'agit d'une liaison récursive, car aucune confusion n'est plus possible.

Passons maintenant à la description de la sémantique opérationnelle de la machine, sous la forme d'une relation de transition $\mathcal{M} \to \mathcal{M}'$. Le premier groupe de règles sert à l'évaluation

[14] Nous pourrions aller plus loin et utiliser ici des indices de De Bruijn en place des variables.

d'une application (MN) de façon standard : la première étape consiste à stocker l'argument N et l'environnement courant sur la pile et à continuer avec l'évaluation de M. Lorsque celle-ci est terminée, on stocke le résultat en attente et on dépile l'évaluation de N laissée en attente. Enfin, lorsque fonction et argument sont entièrement évalués, il est possible de procéder à la β-réduction, en rajoutant une liaison dans l'environnement, avant de continuer avec l'évaluation du corps de la fonction.

$$
\begin{array}{lrcl}
(\mathsf{ApplStart}) & (S, \sigma, (MN), \xi) & \to & (S :: ([]\,\sigma N), \sigma, M, \xi) \\
(\mathsf{ApplSwap}) & (S :: ([]\,\rho N), \sigma, A, \xi) & \to & (S :: (\sigma A\,[]), \rho, N, \xi) \\
(\mathsf{ApplClose}) & (S :: (\sigma \lambda x M\,[]), \rho, A, \xi) & \to & (S, \sigma :: \{x \mapsto \rho A\}, M, \xi)
\end{array}
$$

Le deuxième groupe de règles concerne la récursion. Deux cas peuvent se produire. Dans le cas d'une valeur récursive, nous créons directement une liaison récursive dans l'environnement (et nous n'avons donc pas besoin de la mémoire). Sinon, nous utilisons la technique de point fixe de Landin évoquée plus haut en allouant une nouvelle adresse mémoire ℓ vide. Une fois la valeur trouvée, nous mettons à jour cette case mémoire, et nous continuons avec cette même valeur.

$$
\begin{array}{lrcl}
(\mathsf{RecStd}) & (S, \sigma, \mu x A, \xi) & \to & (S, \sigma :: \{x \mapsto A\}, A, \xi) \\
(\mathsf{RecStart}) & (S, \sigma, \mu x M, \xi) & \to & (S :: \mu\ell\,[], \sigma :: \{x \mapsto \ell\}, M, \xi \uplus \{\ell \mapsto \bullet\}) \\
 & & & \text{pour } M \neq A \text{ et } \ell \text{ frais} \\
(\mathsf{RecClose}) & (S :: \mu\ell\,[], \sigma, A, \xi) & \to & (S, \sigma, A, \xi[\ell := \sigma A])
\end{array}
$$

Le troisième groupe de règles concerne la recherche de liaisons dans l'environnement. La première règle dépile simplement les liaisons non pertinentes ($\sigma :: \{y \mapsto Z\}$ désigne n'importe lequel des trois types de liaisons). La deuxième règle concerne le cas standard : on enchaîne avec la clôture trouvée. La troisième règle concerne les liaisons récursives : on notera que le nouvel environnement est bien $\sigma :: \{x \mapsto A\}$ et non σ comme dans le cas précédent, car x reste liée dans A. Enfin, dans le cas d'une adresse vers une case mémoire, on va chercher la valeur dans celle-ci.

$$
\begin{array}{lrcll}
(\mathsf{VarOther}) & (S, \sigma :: \{y \mapsto Z\}, x, \xi) & \to & (S, \sigma, x, \xi) & x \neq y \\
(\mathsf{VarStd}) & (S, \sigma :: \{x \mapsto \rho A\}, x, \xi) & \to & (S, \rho, A, \xi) & \\
(\mathsf{VarRec}) & (S, \sigma :: \{x \mapsto A\}, x, \xi) & \to & (S, \sigma :: \{x \mapsto A\}, A, \xi) & \\
(\mathsf{VarLoc}) & (S, \sigma :: \{x \mapsto \ell\}, x, \xi) & \to & (S, \rho, A, \xi) & \xi(\ell) = \rho A
\end{array}
$$

Enfin, il nous reste le cas $(S, \rho :: \{x \mapsto \ell\}, x, \xi)$ avec $\xi(\ell) = \bullet$. Nous avons déjà évoqué la propriété (Lemme 5.1) que pour un terme typable, x devait forcément être l'argument d'une fonction (protectrice) en attente, et qu'il suffit donc de "passer" le pointeur vers ℓ en guise d'argument, sans qu'il soit besoin de fournir une "vraie" valeur. D'où la règle "magique" suivante :

$$
\begin{array}{lrcl}
(\mathsf{Magic}) & (S :: (\sigma \lambda y M\,[]), \rho :: \{x \mapsto \ell\}, x, \xi) & \to & (S, \sigma :: \{y \mapsto \ell\}, M, \xi) \\
 & & & \text{si } \xi(\ell) = \bullet
\end{array}
$$

On notera qu'on n'a *pas* la règle $(S, \sigma :: \{x \mapsto \ell\}, x, \xi) \to (S, \sigma, \bullet, \xi)$ lorsque $\xi(\ell) = \bullet$.

Dans la suite de ce chapitre, nous allons prouver que ces 11 règles sont suffisantes pour couvrir tous les cas (pour des termes typables) et que la machine est correcte vis-à-vis de la sémantique du langage.

Exemple 5.2 *Soit à évaluer le terme $M = \mu z(Kz)$ avec $K = \lambda x \lambda y x$ (on pourra vérifier que M est bien typable). Son évaluation à partir d'un état initial de la machine est le suivant :*

$$\begin{array}{rcll}
(\varepsilon, \varepsilon, M, \varnothing) & \to & (\varepsilon :: \mu\ell\,[], \sigma, (Kz), \xi) & \text{avec } \ell \text{ frais, } \sigma = \varepsilon :: \{z \mapsto \ell\} \text{ et } \xi = \varnothing \uplus \{\ell \mapsto \bullet\} \\
& \to & (\varepsilon :: \mu\ell\,[] :: ([]\,\sigma z), \sigma, K, \xi) & \\
& \to & (\varepsilon :: \mu\ell\,[] :: (\sigma K\,[]), \sigma, z, \xi) & \\
& \to & (\varepsilon :: \mu\ell\,[], \rho, \lambda y x, \xi) & \text{avec } \rho = \sigma :: \{x \mapsto \ell\} \text{ (règle (Magic))} \\
& \to & (\varepsilon, \rho, \lambda y x, \xi[\ell := \rho\lambda y x]) &
\end{array}$$

et nous allons voir dans la suite que cette configuration est finale et que son "interprétation" (i.e. le résultat du calcul donné par la machine) est bien $\lambda y(\mu z \lambda y z)$.

5.3 Preuve de correction

5.3.1 Interprétation

La preuve de correction de la machine abstraite est basée principalement sur la capacité à reconstruire un terme du langage à partir d'un état donné de la machine. Ceci dans le but bien évidemment de montrer une correspondance opérationnelle entre ces deux entités.

Nous commençons par définir de façon récursive l'interprétation d'un environnement et d'une mémoire. Le résultat est une substitution :

$$\begin{array}{ll}
[\![\varepsilon]\!] = id & [\![\varnothing]\!] = id \\
[\![\sigma :: \{x \mapsto \rho A\}]\!] = [\![\sigma]\!] \uplus \{^{[\![\rho]\!](A)}/_x\} & [\![\xi \uplus \{\ell \mapsto \bullet\}]\!] = [\![\xi]\!] \\
[\![\sigma :: \{x \mapsto A\}]\!] = [\![\sigma]\!] \circ \{^{\mu x A}/_x\} & [\![\xi[\ell := \rho A]]\!] = \{^{\mu x\{^x/_\ell\} \circ [\![\xi]\!] \circ [\![\rho]\!](A)}/_\ell\} \circ [\![\xi]\!] \\
[\![\sigma :: \{x \mapsto \ell\}]\!] = [\![\sigma]\!] \uplus \{^\ell/_x\} &
\end{array}$$

Dans cette définition, l'opération d'extension d'une substitution $\theta \uplus \{^Z/_x\}$ est définie comme suit :

$$\begin{array}{rcl}
(\theta \uplus \{^Z/_x\})(x) & = & Z \\
(\theta \uplus \{^Z/_x\})(y) & = & \theta(y) \quad \text{pour } y \neq x
\end{array}$$

L'opération de composition $\theta \circ \theta'$ est elle définie avec son sens mathématique usuel.

Puis, nous définissons l'interprétation d'une pile (paramétrée par une mémoire ξ) ; il s'agit d'un contexte d'évaluation pour un terme M :

$$\begin{array}{l}
[\![\varepsilon]\!]_\xi[M] = M \\
[\![S :: F]\!]_\xi[M] = [\![S]\!]_\xi[[\![F]\!]_\xi[M]] \\
[\![([]\,\sigma N)]\!]_\xi[M] = (M \;\; [\![\xi]\!] \circ [\![\sigma]\!](N)) \\
[\![(\sigma A\,[])]\!]_\xi[M] = ([\![\xi]\!] \circ [\![\sigma]\!](A) \;\; M) \\
[\![\mu\ell\,[]]\!]_\xi[M] = \mu x\{^x/_\ell\}M \qquad \text{avec } x \notin fv(M) \quad \text{(en fait, } M \text{ sera toujours clos...)}
\end{array}$$

Enfin, l'interprétation d'un état de la machine est donné par :

$$[\![(S, \sigma, M, \xi)]\!] = [\![S]\!]_\xi[[\![\xi]\!] \circ [\![\sigma]\!](M)]$$

Les deux définitions suivantes identifient respectivement les états de départ et d'arrêt de la machine :

Définition 5.3 *Une* configuration initiale *est un état de la machine de la forme* $\mathcal{L}(M) = (\varepsilon, \varepsilon, M, \varnothing)$ *où* M *est un terme clos.*

Définition 5.4 *Une* configuration finale *est un état de la machine de la forme* $(\varepsilon, \sigma, A, \xi)$.

5.3.2 Lemmes préliminaires

Les lemmes suivants énoncent quelques propriétés concernant les substitutions qui seront utiles pour les théorèmes à venir. Ils sont faciles à démontrer et en conséquence leur démonstration est omise.

Lemme 5.5 $\lambda x M = \lambda y\{^y/_x\}(M)$ *et* $\mu x M = \mu y\{^y/_x\}(M)$ *si* $y \notin fv(M)$.

Lemme 5.6 *Pour toute substitution* θ *telle que* $y \notin dom(\theta) \cup im(\theta)$, *on a :* $\theta(\lambda y M) = \lambda y \theta(M)$ *et* $\theta(\mu y M) = \mu y \theta(M)$.

Lemme 5.7 *Pour toute substitution* θ *telle que* $y \notin dom(\theta) \cup im(\theta)$, *on a :* $\{^{\theta(Z)}/_y\} \circ \theta = \theta \circ \{^Z/_y\}$.

Lemme 5.8 *Pour toute substitution* θ *telle que* $y \notin dom(\theta)$, *on a :* $\theta \circ \{^y/_x\} = \theta \uplus \{^y/_x\}$.

Lemme 5.9 *Pour toute substitution* θ *telle que* $y \notin dom(\theta)$ *et* $\ell \notin dom(\theta) \cup im(\theta)$, *on a :* $\{^y/_\ell\} \circ \theta = \theta \circ \{^y/_\ell\}$.

Lemme 5.10 *Pour toute substitution* θ *telle que* $y \notin dom(\theta) \cup im(\theta) \cup fv(M)$, *on a :* $\{^Z/_y\} \circ \theta \circ \{^y/_x\}(M) = (\theta \uplus \{^Z/_x\})(M)$.

Lemme 5.11 $[\![\xi \uplus \{\ell \mapsto \bullet\}]\!] = [\![\xi]\!]$ *et* $[\![S]\!]_{\xi \uplus \{\ell \mapsto \bullet\}} = [\![S]\!]_\xi$.

Les deux lemmes suivants donnent des propriétés concernant les états de la machine, soit des propriétés d'invariance, soit des propriétés vérifiées pour les états obtenus à partir de configurations initiales.

Lemme 5.12 *Les propriétés suivantes sur les états de la machine sont des invariants pour la relation de transition (on note* $\mathcal{M} = (S, \sigma, M, \xi)$ *un état générique) :*

- $fv(M) \subseteq dom(\sigma)$
- *pour chaque frame* $([]\,\rho M)$ *et* $(\rho A\,[])$ *dans* S, *on a :* $fv(M) \subseteq dom(\rho)$ *et* $fv(A) \subseteq dom(\rho)$ *respectivement*
- *pour chaque substitution* $\sigma' :: \{x \mapsto \rho A\}$ *dans* σ, *on a :* $fv(A) \subseteq dom(\rho)$
- *pour chaque substitution* $\rho :: \{x \mapsto A\}$ *dans* σ, *on a :* $fv(A) \setminus \{x\} \subseteq dom(\rho)$
- *pour chaque affectation* $\xi'[\ell := \rho A]$ *dans* ξ, *on a :* $fv(A) \subseteq dom(\rho)$
- $[\![\mathcal{M}]\!]$ *est un terme clos.*

Lemme 5.13 *Dans ce lemme, on supposera que la machine a été lancée à partir d'une configuration initiale.*

- *Si on atteint un état* $(S :: \mu\ell\,[], \sigma, M, \xi)$, *alors* ℓ *n'apparaît pas dans* S. *De plus,* $[\![S]\!]_{\xi[\ell:=\rho A]} = [\![S]\!]_\xi$.

- *Soit U une métavariable pour l'opération $\uplus\{\ell \mapsto \bullet\}$ ou $[\ell := \rho A]$. Si $\xi = \varnothing U_1 \dots U_n$, alors il existe au plus une opération de chaque type pour chaque ℓ_i. De plus, si $U_i = [\ell_i := \rho_i A_i]$, alors $\xi(\ell_i) = \rho_i A_i$.*
- *Réciproquement, si $\xi(\ell) = \rho A$, alors $\xi = \xi'[\ell := \rho A]U_1 \dots U_n$ avec $\ell \notin dom(U_i)$ et $\xi'(\ell) = \bullet$. Donc, $[\![\xi]\!](\ell) = [\![U_n]\!] \circ \dots \circ [\![U_1]\!](\mu x\{^x/_\ell\} \circ [\![\xi']\!] \circ [\![\rho]\!](A))$.*
- *Si $\xi(\ell) = \bullet$, alors $[\![\xi]\!](\ell) = \ell$.*

5.3.3 Correspondance opérationnelle

Le premier résultat montre que pour une transition de la machine, les interprétations correspondantes sont soit identiques, soit reliées par une réduction, et que la distinction entre ces deux cas se fait simplement par la règle de réduction utilisée.

Proposition 5.14 *On suppose que la machine a été lancée à partir d'une configuration initiale. Pour chaque transition $\mathcal{M} \to \mathcal{M}'$ entre états de la machine, on a soit $[\![\mathcal{M}]\!] \to [\![\mathcal{M}']\!]$, soit $[\![\mathcal{M}]\!] = [\![\mathcal{M}']\!]$ (dans ce cas, on dit que la transition $\mathcal{M} \to \mathcal{M}'$ est* silencieuse*).*

Preuve: Examinons les 11 cas possibles :

(ApplStart)

$$\begin{aligned}
[\![\mathcal{M}]\!] &= [\![(S, \sigma, (MN), \xi)]\!] \\
&= [\![S]\!]_\xi[[\![\xi]\!] \circ [\![\sigma]\!](MN)] \\
&= [\![S]\!]_\xi[[\![\xi]\!] \circ [\![\sigma]\!](M) \quad [\![\xi]\!] \circ [\![\sigma]\!](N)] \\
&= [\![S :: ([]\,\sigma N)]\!]_\xi[[\![\xi]\!] \circ [\![\sigma]\!](M)] \\
&= [\![(S :: ([]\,\sigma N), \sigma, M, \xi)]\!] \\
&= [\![\mathcal{M}']\!]
\end{aligned}$$

(ApplSwap)

$$\begin{aligned}
[\![\mathcal{M}]\!] &= [\![(S :: ([]\,\rho N), \sigma, A, \xi)]\!] \\
&= [\![S :: ([]\,\rho N)]\!]_\xi[[\![\xi]\!] \circ [\![\sigma]\!](A)] \\
&= [\![S]\!]_\xi[[\![\xi]\!] \circ [\![\sigma]\!](A) \quad [\![\xi]\!] \circ [\![\rho]\!](N)] \\
&= [\![S :: (\sigma A\,[])]\!]_\xi[[\![\xi]\!] \circ [\![\rho]\!](N)] \\
&= [\![(S :: (\sigma A\,[]), \rho, N, \xi)]\!] \\
&= [\![\mathcal{M}']\!]
\end{aligned}$$

(ApplClose) Soit y une variable fraîche, plus précisément telle que $y \notin fv(M) \cup dom(\xi) \cup im(\xi) \cup dom(\sigma) \cup im(\sigma)$. On a alors :

$$\begin{aligned}
[\![\mathcal{M}]\!] &= [\![(S :: (\sigma\lambda x M\,[]), \rho, A, \xi)]\!] \\
&= [\![S :: (\sigma\lambda x M\,[])]\!]_\xi[[\![\xi]\!] \circ [\![\rho]\!](A)] \\
&= [\![S]\!]_\xi[[\![\xi]\!] \circ [\![\sigma]\!](\lambda x M) \quad [\![\xi]\!] \circ [\![\rho]\!](A)] \\
&= [\![S]\!]_\xi[[\![\xi]\!] \circ [\![\sigma]\!](\lambda y\{^y/_x\}(M)) \quad [\![\xi]\!] \circ [\![\rho]\!](A)] && \text{par le Lemme 5.5} \\
&= [\![S]\!]_\xi[\lambda y[\![\xi]\!] \circ [\![\sigma]\!] \circ \{^y/_x\}(M) \quad [\![\xi]\!] \circ [\![\rho]\!](A)] && \text{par le Lemme 5.6} \\
&\to [\![S]\!]_\xi[\{^{[\![\xi]\!] \circ [\![\rho]\!](A)}/_y\} \circ [\![\xi]\!] \circ [\![\sigma]\!] \circ \{^y/_x\}(M)] \\
&= [\![S]\!]_\xi[[\![\xi]\!] \circ \{^{[\![\rho]\!](A)}/_y\} \circ [\![\sigma]\!] \circ \{^y/_x\}(M)] && \text{par le Lemme 5.7} \\
&= [\![S]\!]_\xi[[\![\xi]\!] \circ ([\![\sigma]\!] \uplus \{^{[\![\rho]\!](A)}/_x\})(M)] && \text{par le Lemme 5.10} \\
&= [\![S]\!]_\xi[[\![\xi]\!] \circ [\![\sigma :: \{x \mapsto \rho A\}]\!](M)] \\
&= [\![(S, \sigma :: \{x \mapsto \rho A\}, M, \xi)]\!] \\
&= [\![\mathcal{M}']\!]
\end{aligned}$$

(Magic) On suppose que $\xi(\ell) = \bullet$. Soit z une variable fraîche, plus précisément telle que $z \notin fv(M) \cup dom(\xi) \cup im(\xi) \cup dom(\sigma) \cup im(\sigma)$. On a alors :

$$\begin{array}{rcll}
[\![\mathcal{M}]\!] & = & [\![(S :: (\sigma \lambda y M\ []), \rho :: \{x \mapsto \ell\}, x, \xi)]\!] & \\
& = & [\![S :: (\sigma \lambda y M\ [])]\!]_\xi [[\![\xi]\!] \circ [\![\rho :: \{x \mapsto \ell\}]\!](x)] & \\
& = & [\![S]\!]_\xi [[\![\xi]\!] \circ [\![\sigma]\!](\lambda y M) \quad [\![\xi]\!] \circ ([\![\rho]\!] \uplus \{^\ell/_x\})(x)] & \\
& = & [\![S]\!]_\xi [[\![\xi]\!] \circ [\![\sigma]\!](\lambda z \{^z/_y\}(M)) \quad [\![\xi]\!](\ell)] & \text{par le Lemme 5.5} \\
& = & [\![S]\!]_\xi [\lambda z [\![\xi]\!] \circ [\![\sigma]\!] \circ \{^z/_y\}(M) \quad [\![\xi]\!](\ell)] & \text{par le Lemme 5.6} \\
& \rightarrow & [\![S]\!]_\xi [\{^{[\![\xi]\!](\ell)}/_z\} \circ [\![\xi]\!] \circ [\![\sigma]\!] \circ \{^z/_y\}(M)] & \text{(note : on ne se sert pas de} \\
& & & \text{l'hypothèse } \xi(\ell) = \bullet) \\
& = & [\![S]\!]_\xi [[\![\xi]\!] \circ \{^\ell/_z\} \circ [\![\sigma]\!] \circ \{^z/_y\}(M)] & \text{par le Lemme 5.7} \\
& = & [\![S]\!]_\xi [[\![\xi]\!] \circ ([\![\sigma]\!] \uplus \{^\ell/_y\})(M)] & \text{par le Lemme 5.10} \\
& = & [\![S]\!]_\xi [[\![\xi]\!] \circ [\![\sigma :: \{y \mapsto \ell\}]\!](M)] & \\
& = & [\![(S, \sigma :: \{y \mapsto \ell\}, M, \xi)]\!] & \\
& = & [\![\mathcal{M}']\!] &
\end{array}$$

(RecStd)

$$\begin{array}{rcl}
[\![\mathcal{M}]\!] & = & [\![(S, \sigma, \mu x A, \xi)]\!] \\
& = & [\![S]\!]_\xi [[\![\xi]\!] \circ [\![\sigma]\!](\mu x A)] \\
& \rightarrow & [\![S]\!]_\xi [[\![\xi]\!] \circ [\![\sigma]\!] \circ \{^{\mu x A}/_x\}(A)] \\
& = & [\![S]\!]_\xi [[\![\xi]\!] \circ [\![\sigma :: \{x \mapsto A\}]\!](A)] \\
& = & [\![(S, \sigma :: \{x \mapsto A\}, A, \xi)]\!] \\
& = & [\![\mathcal{M}']\!]
\end{array}$$

(RecStart) On rappelle que $M \neq A$ et que $\ell \notin fv(M) \cup dom(\xi) \cup im(\xi) \cup dom(\sigma) \cup im(\sigma)$. De plus, soit y une autre variable fraîche, plus précisément telle que $y \notin fv(M) \cup dom(\xi) \cup im(\xi) \cup dom(\sigma) \cup im(\sigma)$.

$$\begin{array}{rcll}
[\![\mathcal{M}]\!] & = & [\![(S, \sigma, \mu x M, \xi)]\!] & \\
& = & [\![S]\!]_\xi [[\![\xi]\!] \circ [\![\sigma]\!](\mu x M)] & \\
& = & [\![S]\!]_\xi [[\![\xi]\!] \circ [\![\sigma]\!](\mu y \{^y/_\ell\} \circ \{^\ell/_x\}(M))] & \text{par le Lemme 5.5} \\
& = & [\![S]\!]_\xi [\mu y [\![\xi]\!] \circ [\![\sigma]\!] \circ \{^y/_\ell\} \circ \{^\ell/_x\}(M)] & \text{par le Lemme 5.6} \\
& = & [\![S]\!]_\xi [\mu y \{^y/_\ell\} \circ [\![\xi]\!] \circ [\![\sigma]\!] \circ \{^\ell/_x\}(M)] & \text{par le Lemme 5.9} \\
& = & [\![S :: \mu \ell\ []]\!]_\xi [[\![\xi]\!] \circ ([\![\sigma]\!] \uplus \{^\ell/_x\})(M)] & \text{par le Lemme 5.8} \\
& = & [\![S :: \mu \ell\ []]\!]_{\xi \uplus \{\ell \mapsto \bullet\}} [[\![\xi \uplus \{\ell \mapsto \bullet\}]\!] \circ [\![\sigma :: \{x \mapsto \ell\}]\!](M)] & \text{par le Lemme 5.11} \\
& = & [\![(S :: \mu \ell\ [], \sigma :: \{x \mapsto \ell\}, M, \xi \uplus \{\ell \mapsto \bullet\})]\!] & \\
& = & [\![\mathcal{M}']\!] &
\end{array}$$

(RecClose)

$$\begin{array}{rcll}
[\![\mathcal{M}]\!] & = & [\![(S :: \mu \ell\ [], \sigma, A, \xi)]\!] & \\
& = & [\![S :: \mu \ell\ []]\!]_\xi [[\![\xi]\!] \circ [\![\sigma]\!](A)] & \\
& = & [\![S]\!]_\xi [\mu x \{^x/_\ell\} \circ [\![\xi]\!] \circ [\![\sigma]\!](A)] & \\
& \rightarrow & [\![S]\!]_\xi [\{^{\mu x \{^x/_\ell\} \circ [\![\xi]\!] \circ [\![\sigma]\!](A)}/_x\} \circ \{^x/_\ell\} \circ [\![\xi]\!] \circ [\![\sigma]\!](A)] & \\
& = & [\![S]\!]_\xi [\{^{\mu x \{^x/_\ell\} \circ [\![\xi]\!] \circ [\![\sigma]\!](A)}/_\ell\} \circ [\![\xi]\!] \circ [\![\sigma]\!](A))] & \text{puisqu'une variable } x \text{ ne peut} \\
& & & \text{apparaître libre dans } [\![\xi]\!] \circ [\![\sigma]\!](A) \\
& = & [\![S]\!]_\xi [[\![\xi[\ell := \sigma A]]\!] \circ [\![\sigma]\!](A)] & \\
& = & [\![S]\!]_{\xi[\ell := \sigma A]} [[\![\xi[\ell := \sigma A]]\!] \circ [\![\sigma]\!](A)] & \text{par le Lemme 5.13} \\
& = & [\![(S, \sigma, A, \xi[\ell := \sigma A])]\!] & \\
& = & [\![\mathcal{M}']\!] &
\end{array}$$

(VarOther) On suppose que $x \neq y$. Suivant la valeur de Z, $[\![\sigma :: \{y \mapsto Z\}]\!]$ peut être de la forme $[\![\sigma]\!] \uplus \{^{\cdots}/_{y}\}$ ou $[\![\sigma]\!] \circ \{^{\cdots}/_{y}\}$. Mais, dans tous les cas, on a : $[\![\sigma :: \{y \mapsto Z\}]\!](x) = [\![\sigma]\!](x)$. Alors,

$$\begin{array}{rcl} [\![\mathcal{M}]\!] & = & [\![(S, \sigma :: \{y \mapsto Z\}, x, \xi)]\!] \\ & = & [\![S]\!]_{\xi}[[\![\xi]\!] \circ [\![\sigma :: \{y \mapsto Z\}]\!](x)] \\ & = & [\![S]\!]_{\xi}[[\![\xi]\!] \circ [\![\sigma]\!](x)] \\ & = & [\![(S, \sigma, x, \xi)]\!] \\ & = & [\![\mathcal{M}']\!] \end{array}$$

(VarStd)

$$\begin{array}{rcl} [\![\mathcal{M}]\!] & = & [\![(S, \sigma :: \{x \mapsto \rho A\}, x, \xi)]\!] \\ & = & [\![S]\!]_{\xi}[[\![\xi]\!] \circ [\![\sigma :: \{x \mapsto \rho A\}]\!](x)] \\ & = & [\![S]\!]_{\xi}[[\![\xi]\!] \circ ([\![\sigma]\!] \uplus \{^{[\![\rho]\!](A)}/_{x}\})(x)] \\ & = & [\![S]\!]_{\xi}[[\![\xi]\!]([\![\rho]\!](A))] \\ & = & [\![(S, \rho, A, \xi)]\!] \\ & = & [\![\mathcal{M}']\!] \end{array}$$

(VarRec)

$$\begin{array}{rcl} [\![\mathcal{M}]\!] & = & [\![(S, \sigma :: \{x \mapsto A\}, x, \xi)]\!] \\ & = & [\![S]\!]_{\xi}[[\![\xi]\!] \circ [\![\sigma :: \{x \mapsto A\}]\!](x)] \\ & = & [\![S]\!]_{\xi}[[\![\xi]\!] \circ [\![\sigma]\!] \circ \{^{\mu x A}/_{x}\}(x)] \\ & = & [\![S]\!]_{\xi}[[\![\xi]\!] \circ [\![\sigma]\!](\mu x A)] \\ & \rightarrow & [\![S]\!]_{\xi}[[\![\xi]\!] \circ [\![\sigma]\!] \circ \{^{\mu x A}/_{x}\}(A)] \\ & = & [\![S]\!]_{\xi}[[\![\xi]\!] \circ [\![\sigma :: \{x \mapsto A\}]\!](A)] \\ & = & [\![(S, \sigma :: \{x \mapsto A\}, A, \xi)]\!] \\ & = & [\![\mathcal{M}']\!] \end{array}$$

(VarLoc) On suppose que $\xi(\ell) = \rho A$. D'après le Lemme 5.13, on sait que $\xi = \xi'[\ell := \rho A]U_1 \dots U_n$ avec $\ell \notin dom(U_i)$ et $\xi'(\ell) = \bullet$.

$$\begin{array}{rcll} [\![\mathcal{M}]\!] & = & [\![(S, \sigma :: \{x \mapsto \ell\}, x, \xi)]\!] & \\ & = & [\![S]\!]_{\xi}[[\![\xi]\!] \circ ([\![\sigma]\!] \uplus \{^{\ell}/_{x}\})(x)] & \\ & = & [\![S]\!]_{\xi}[[\![\xi]\!](\ell)] & \\ & = & [\![S]\!]_{\xi}[[\![U_n]\!] \circ \dots \circ [\![U_1]\!](\mu y\{^{y}/_{\ell}\} \circ [\![\xi']\!] \circ [\![\rho]\!](A))] & \text{par le Lemme 5.13} \\ & \rightarrow & [\![S]\!]_{\xi}[[\![U_n]\!] \circ \dots \circ [\![U_1]\!] \circ \{^{\mu y\{^{y}/_{\ell}\} \circ [\![\xi']\!] \circ [\![\rho]\!](A)}/_{y}\} \circ \{^{y}/_{\ell}\} \circ [\![\xi']\!] \circ [\![\rho]\!](A)] & \\ & = & [\![S]\!]_{\xi}[[\![U_n]\!] \circ \dots \circ [\![U_1]\!] \circ \{^{\mu y\{^{y}/_{\ell}\} \circ [\![\xi']\!] \circ [\![\rho]\!](A)}/_{\ell}\} \circ [\![\xi']\!] \circ [\![\rho]\!](A)] & \text{car } y \notin fv([\![\xi']\!] \circ [\![\rho]\!](A)) \\ & = & [\![S]\!]_{\xi}[[\![U_n]\!] \circ \dots \circ [\![U_1]\!] \circ [\![\xi'[\ell := \rho A]]\!] \circ [\![\rho]\!](A)] & \\ & = & [\![S]\!]_{\xi}[[\![\xi]\!] \circ [\![\rho]\!](A)] & \\ & = & [\![(S, \rho, A, \xi)]\!] & \\ & = & [\![\mathcal{M}']\!] & \end{array}$$

□

Il ne sera pas nécessaire de démontrer directement la propriété réciproque, car nous allons la déduire du déterminisme de la machine et du déterminisme du calcul sous-jacent.

5.3.4 Déterminisme

Proposition 5.15 *On suppose que la machine a été lancée à partir d'une configuration initiale. Soit $\mathcal{M}$ un état de la machine tel que $[\![\mathcal{M}]\!]$ soit typable. Alors, soit $\mathcal{M}$ est une configuration finale et la machine ne peut plus réduire, soit $\mathcal{M}$ peut réduire et cette réduction est unique.*

Preuve: Soit $\mathcal{M} = (S, \sigma, M, \xi)$ un état de la machine tel que $[\![\mathcal{M}]\!]$ soit typable. Nous donnons un algorithme pour déterminer quelle règle de réduction doit être employée, suivant la valeur de $\mathcal{M}$. Et, comme on peut le vérifier dans chaque cas, seule cette règle peut être appliquée (ce que nous ne répéterons pas pour chaque cas).

Considérons tout d'abord M.

Cas $(M'N')$ **:** Appliquer la règle (ApplStart).

Cas $\mu x M'$ **:** On considère M'.

Cas A **:** Appliquer la règle (RecStd).

Sinon : Appliquer la règle (RecStart).

Cas A **:** On considère S.

Cas $S' :: ([]\, \rho N)$ **:** Appliquer la règle (ApplSwap).

Cas $S' :: (\sigma A'\, [])$ **:** Appliquer la règle (ApplClose). [15]

Cas $S' :: \mu\ell\, []$ **:** Appliquer la règle (RecClose).

Cas ε **:** Nous sommes dans une configuration finale, aucune règle ne s'applique, la machine s'arrête.

Cas x **:** On considère σ.

Cas $\sigma' :: \{y \mapsto Z\}$**,** $y \neq x$ **:** Appliquer la règle (VarOther).

Cas $\sigma' :: \{x \mapsto \rho A\}$ **:** Appliquer la règle (VarStd).

Cas $\sigma' :: \{x \mapsto A\}$ **:** Appliquer la règle (VarRec).

Cas $\sigma' :: \{x \mapsto \ell\}$ **:** On considère $\xi(\ell)$.

Cas ρA **:** Appliquer la règle (VarLoc).

Cas $\bullet$ **:** Puisque $\xi(\ell) = \bullet$, on a $[\![\xi]\!]\ell = \ell$ par le Lemme 5.13 . Alors, l'interprétation de l'état courant est : $[\![\mathcal{M}]\!] = [\![S]\!]_\xi[[\![\xi]\!] \circ [\![\sigma]\!](x)] = [\![S]\!]_\xi[\ell]$. Puisque $[\![\mathcal{M}]\!]$ est clos (Lemme 5.12), nécessairement S est de la forme : $S = S' :: \mu\ell\, [] :: F_1 :: \ldots :: F_n$ où aucun F_i ne lie ℓ. Alors, $[\![\mathcal{M}]\!] = [\![S']\!]_\xi[\mu x\{^x/_\ell\}([\![F_1]\!]_\xi \ldots [\![F_n]\!]_\xi[\ell])]$. Comme $[\![\mathcal{M}]\!]$ est typable, $\mu x\{^x/_\ell\}([\![F_1]\!]_\xi \ldots [\![F_n]\!]_\xi[\ell])$ doit également être typable. D'après le Lemme 5.1, et puisque chaque $[\![F_i]\!]_\xi$ est un contexte d'évaluation qui ne lie pas ℓ, nécessairement F_n est de la forme $(\rho A\, [])$, où $[\![\xi]\!] \circ [\![\rho]\!](A)$ est une fonction qui protège son argument. On peut alors appliquer la règle (Magic).

Cas ε **:** Ce cas ne peut pas se produire car on doit avoir $x \in dom(\sigma)$ (Lemme 5.12).

□

5.3.5 Mesure

Nous avons vu que certaines réductions de la machine correspondaient à de vraies réductions du terme sous-jacent, et que d'autres, les transitions silencieuses, le laissaient inchangé. Afin de prouver que la machine "avance" toujours dans son calcul, nous allons maintenant montrer qu'elle ne peut pas rester bloquée sur une suite infinie de transitions silencieuses.

Proposition 5.16 *Il existe une mesure positive sur les états de la machine, strictement décroissante pour les transitions silencieuses.*

[15] Notons que si on étend notre calcul avec des valeurs autres que les fonctions (des entiers par exemple), nous aurions besoin ici de l'hypothèse de typabilité pour nous assurer que A' est bien une fonction. On pourrait aussi limiter la syntaxe des frames avec $(\sigma\lambda x M\, [])$, mais c'est alors dans (ApplSwap) qu'il faudrait faire appel au typage.

Preuve: On définit la mesure suivante sur les états de la machine :

$$
\begin{array}{lll}
|(MN)| = |M| + |N| + 2 & |\varepsilon| = 0 & |\varepsilon| = 1 \\
|\mu x M| = 2|M| + 1 & |S :: F| = |S| + |F| & |\sigma :: \{x \mapsto Z\}| = |\sigma| + 1 \\
|x| = 1 & |([]\,\sigma N)| = |\sigma|(|N| + 1) & \\
|A| = 0 & |(\sigma A\,[])| = 0 & \\
 & |\mu\ell\,[]| = 0 & |(S, \sigma, M, \xi)| = |S| + |\sigma||M|
\end{array}
$$

Notons tout d'abord que $|\sigma|$ est en fait le nombre de liaisons dans σ augmenté de 1. Donc, $|\sigma| > 0$ pour tout σ. En outre, $|\mathcal{M}| \geq 0$ pour tout état $\mathcal{M}$ de la machine.

Il nous faut montrer que si $\mathcal{M} \to \mathcal{M}'$ avec $[\![\mathcal{M}]\!] = [\![\mathcal{M}']\!]$ (c'est-à-dire pour les règles de réduction (ApplStart), (ApplSwap), (RecStart), (VarOther) et (VarStd)), on a $|\mathcal{M}| > |\mathcal{M}'| \geq 0$. Examinons donc ces cinq règles :

(ApplStart)

$$
\begin{array}{rcl}
|(S, \sigma, (MN), \xi)| & = & |S| + |\sigma|(|M| + |N| + 2) \\
 & = & |S| + |\sigma|(|N| + 1) + |\sigma||M| + |\sigma| \\
 & > & |S| + |\sigma|(|N| + 1) + |\sigma||M| \\
 & = & |(S :: ([]\,\sigma N), \sigma, M, \xi)|
\end{array}
$$

(ApplSwap)

$$
\begin{array}{rcl}
|(S :: ([]\,\rho N), \sigma, A, \xi)| & = & |S| + |\rho|(|N| + 1) + 0 \\
 & = & |S| + |\rho||N| + |\rho| \\
 & > & |S| + |\rho||N| \\
 & = & |(S :: (\sigma A\,[]), \rho, N, \xi)|
\end{array}
$$

(RecStart)

$$
\begin{array}{rcll}
|(S, \sigma, \mu x M, \xi)| & = & |S| + |\sigma|(2|M| + 1) & \\
 & = & |S| + |\sigma||M| + |\sigma||M| + |\sigma| & \\
 & > & |S| + |\sigma||M| + |M| & (\text{car } |\sigma| \geq 1) \\
 & = & |S| + (|\sigma| + 1)|M| & \\
 & = & |(S :: \mu\ell\,[], \sigma :: \{x \mapsto \ell\}, M, \xi \uplus \{\ell \mapsto \bullet\})| &
\end{array}
$$

(VarOther)

$$
\begin{array}{rcl}
|(S, \sigma :: \{y \mapsto Z\}, x, \xi)| & = & |S| + (|\sigma| + 1)1 \\
 & > & |S| + |\sigma|1 \\
 & = & |(S, \sigma, x, \xi)|
\end{array}
$$

(VarStd)

$$
\begin{array}{rcl}
|(S, \sigma :: \{x \mapsto \rho A\}, x, \xi)| & = & |S| + (|\sigma| + 1)1 \\
 & > & |S| + 0 \\
 & = & |(S, \rho, A, \xi)|
\end{array}
$$

□

5.3.6 Correction

Définition 5.17 *Soit M un terme clos. La fonction $\mathcal{E}val$ est définie pour M s'il existe une configuration finale $\mathcal{M}$ telle que $\mathcal{L}(M) \to^* \mathcal{M}$. On pose alors : $\mathcal{E}val(M) = [\![\mathcal{M}]\!]$.*

Théorème 5.18 *Soit M un terme clos.*

- *Si $\mathcal{E}val(M)$ est défini, alors $M \to^* \mathcal{E}val(M)$.*
- *Si M est typable et $M \to^* A$, alors $\mathcal{E}val(M)$ est défini et $\mathcal{E}val(M) = A$.*

Preuve:

- Ce résultat est immédiat d'après la Proposition 5.14 : on sait qu'il existe une configuration finale $\mathcal{M}$ telle que $\mathcal{L}(M) \to^* \mathcal{M}$, donc on a : $[\![\mathcal{L}(M)]\!] = M \to^* [\![\mathcal{M}]\!] = \mathcal{E}val(M)$.
- Montrons la propriété suivante : si M est un terme clos et typable qui converge en n étapes, c'est-à-dire
$$M = M_0 \to M_1 \to \cdots \to M_n = A$$
pour une valeur A, et si $\mathcal{M}$ est atteignable depuis $\mathcal{L}(M)$ avec $[\![\mathcal{M}]\!] = M$, alors il existe une configuration finale $\mathcal{M}'$ telle que $\mathcal{M} \to^* \mathcal{M}'$ et $[\![\mathcal{M}']\!] = A$. On procède par induction sur n et sur $|\mathcal{M}|$. Le résultat est immédiat si $n = 0$ (autrement dit si $M = A$) : il suffit de prendre $\mathcal{M} = \mathcal{M}' = \mathcal{L}(M)$. Sinon, M n'est pas une valeur, et donc $\mathcal{M}$ ne peut pas être une configuration finale. Alors, d'après la Proposition 5.15, il existe $\mathcal{M}'$ telle que $\mathcal{M} \to \mathcal{M}'$. Puisque soit $[\![\mathcal{M}']\!] = M_1$ soit $|\mathcal{M}| > |\mathcal{M}'|$ d'après les Propositions 5.14 et 5.16 (et le fait que la réduction est déterministe), on peut conclure par hypothèse d'induction.

□

Afin de voir que l'hypothèse de typabilité est nécessaire, on pourra vérifier par exemple que le terme $\mu x x$, qui n'est pas typable, se réduit en lui-même, tandis que son évaluation par la machine reste bloquée dans la configuration (non finale) :

$$(\varepsilon :: \mu\ell\,[], \varepsilon :: \{x \mapsto \ell\}, x, \varnothing \uplus \{\ell \mapsto \bullet\})$$

où ℓ est frais. Un autre exemple est $\mu x(\mathbf{F}(\mu z x))$, qui se réduit en $\lambda y y$.

5.4 Implantation dans MlObj

Cette machine a été mise en oeuvre dans MLOBJ, en remplacement de la sémantique big-step utilisée dans une première version. Néanmoins, cette opération a nécessité quelques adaptations...

Tout d'abord, MLOBJ est un mini-langage complet avec un ensemble de valeurs beaucoup plus large que simplement les fonctions du λ-calcul. L'ensemble $\mathcal{A}$ est donc à adapter en conséquence. Ensuite, on peut remarquer qu'excepté pour les fonctions, toutes les valeurs manipulées sont closes (i.e. ne contiennent pas de code) et que donc il est inutile de stocker une clôture complète σA dans ces cas. Par conséquent, nous ne stockerons que des valeurs dans la pile de contrôle et dans les environnements, mais en contrepartie nous inclurons les clôtures $\sigma\lambda x M$ dans la catégorie syntaxique des valeurs. Voici donc la grammaire des valeurs ou *réponses* telle qu'elle est définie dans MLOBJ (afin d'éviter toute confusion avec les adresses mémoire ℓ, nous utilisons *label* à la place de l) :

A	::=	$\sigma\ \lambda x M$	clôture
	|	$()$	unit
	|	n	entier
	|	b	booléen
	|	s	chaîne de caractères
	|	r	référence (cellule mémoire mutable)
	|	$\{label_1 = A_1, \ldots, label_n = A_n\}$	enregistrement extensible
	|	$[A_1; \ldots; A_n]$	liste
	|	f	primitive interne de type $\mathcal{A} \to \mathcal{A}$

Concernant le dernier cas, celui des primitives, il s'agit de pouvoir intégrer au système des opérateurs particuliers qui sortent du cadre de la syntaxe de MLOBJ proprement dite. On les utilisera par exemple pour définir les fonctions +, **ref**, **print_int**, etc... (voir Annexe A.6). En interne, n'importe quelle fonction qui prend une réponse et renvoie une réponse peut être intégrée comme primitive ; bien entendu, celles-ci seront chargées uniquement au lancement de l'interpréteur et ne seront accessibles qu'indirectement à l'utilisateur.

Ensuite, l'ensemble des frames doit être étendu pour pouvoir traiter les opérations induites par ces nouvelles valeurs et par les autres constructions du langage : [16]

F	::=	$([]\,\sigma M)$	application gauche
	|	$(A\,[])$	application droite
	|	$\mathbf{let}_\sigma\ x_1 = A_1\ \mathbf{and} \ldots \mathbf{and}\ x_i = []\ \mathbf{and} \ldots \mathbf{and}\ x_n = M_n\ \mathbf{in}\ M$	liaison locale
	|	$\mathbf{let\ rec}_\sigma\ \ell_1 = A_1\ \mathbf{and} \ldots \mathbf{and}\ \ell_i = []\ \mathbf{and} \ldots \mathbf{and}\ \ell_n = M_n\ \mathbf{in}\ M$	liaison locale récursive
	|	$\mathbf{if}_\sigma\ []\ \mathbf{then}\ M\ \mathbf{else}\ N$	conditionnelle
	|	$\{[], label = \sigma M\}$	enregistrement gauche
	|	$\{A, label = []\}$	enregistrement droit
	|	$[] \setminus label$	restriction
	|	$[].label$	sélection

Pour **let**, **let rec** et **if**, l'environnement σ est partagé par tous les termes en attente. Dans **let** et **let rec**, le point i correspond au terme en cours d'évaluation (i.e. les termes M_1 à M_{i-1} ont déjà été évalués en A_1 à A_{i-1} et les termes M_{i+1} à M_n restent à évaluer). Dans **let rec**, les l_i désignent les adresses mémoire où seront rangées les valeurs récursives. Il n'est pas besoin de garder les noms x_i comme pour le **let**, car des liaisons $x_i \mapsto l_i$ ont déjà été stockées dans l'environnement. Enfin, on notera qu'il n'y a pas de frames pour la construction de listes ni pour les références, car toutes les opérations correspondantes sont faites par l'intermédiaire de primitives internes, ce qui simplifie la machine abstraite. Il n'est pas possible de faire de même pour les enregistrements, car les labels n'étant pas des valeurs, il n'est pas possible de les passer en argument de primitives.

Enfin, pour être complet, il resterait à lister les règles de transition pour la machine, mais nous ne les détaillerons pas toutes ici, car il s'agit d'une liste fastidieuse et elles se déduisent facilement des frames ci-dessus et des 11 règles de notre machine de base. Nous présentons néanmoins les plus complexes, celles relatives au **let rec** à liaisons multiples. La règle (RecStart) initie le processus en allouant n adresses mémoires fraîches, les lie dans l'environnement, et enchaîne avec l'évaluation de la première liaison. La règle (RecCont) récupère le résultat de l'évaluation de la i^{e} liaison, et enchaîne avec l'évaluation de la $i+1^{\text{e}}$. Enfin, une fois toutes les liaisons évaluées, la règle (RecClose) permet de stocker les valeurs trouvées en mémoire et d'enchaîner avec le corps du **let rec**.

[16] Il n'est pas indispensable de traiter le **let** par des frames ; on pourrait également envisager d'éliminer les **and** (par un renommage approprié des variables), et dans un second temps de traduire **let** $x = M$ **in** N par $(\lambda x M)N$ au niveau de l'analyseur syntaxique.

(RecStart)
$(S, \sigma, \textbf{let rec } x_1 = M_1 \textbf{ and } \ldots \textbf{ and } x_n = M_n \textbf{ in } N, \xi) \rightarrow$
$(S :: (\textbf{let rec}_\rho\ \ell_1 = [] \textbf{ and } \ell_2 = M_2 \textbf{ and } \ldots \textbf{ and } \ell_n = M_n \textbf{ in } N), \rho, M_1, \xi')$
pour ℓ_i frais et $\rho = \sigma :: \{x_1 \mapsto \ell_1\} :: \cdots :: \{x_n \mapsto \ell_n\}$ et $\xi' = \xi \uplus \{\ell_1 \mapsto \bullet\} \uplus \cdots \uplus \{\ell_n \mapsto \bullet\}$

(RecCont)
$(S :: (\textbf{let rec}_\sigma\ \ell_1 = A_1 \textbf{ and } \ldots \textbf{ and } \ell_i = [] \textbf{ and } \ldots \textbf{ and } \ell_n = M_n \textbf{ in } N), \rho, A, \xi) \rightarrow$
$(S :: (\textbf{let rec}_\sigma\ \ell_1 = A_1 \textbf{ and } \ldots \textbf{ and } \ell_i = A \textbf{ and } \ell_{i+1} = [] \textbf{ and } \ldots \textbf{ and } \ell_n = M_n \textbf{ in } N), \sigma, M_{i+1}, \xi)$

(RecClose)
$(S :: (\textbf{let rec}_\sigma\ \ell_1 = A_1 \textbf{ and } \ldots \textbf{ and } \ell_n = [] \textbf{ in } N), \rho, A, \xi) \rightarrow$
$(S, \sigma, N, \xi[\ell_1 := A_1] \cdots [\ell_{n-1} := A_{n-1}][\ell_n := A])$

Quant aux primitives internes, elles sont tous simplement appliquées à la valeur calculée pour l'argument :

(Prim) $(S :: (f\ []), \sigma, A, \xi) \rightarrow (S, \sigma, f(A), \xi)$

Par ailleurs, du fait de l'ajout de cette deuxième forme de "fonctions", on pourra noter qu'il y a maintenant deux règles "magiques" : une pour l'application standard et une pour les primitives internes.

$(\text{Magic})_1$ $(S :: (\sigma\lambda y M\ []), \rho :: \{x \mapsto \ell\}, x, \xi) \rightarrow (S, \sigma :: \{y \mapsto \ell\}, M, \xi)$ si $\xi(\ell) = \bullet$

$(\text{Magic})_2$ $(S :: (f\ []), \rho :: \{x \mapsto \ell\}, x, \xi) \rightarrow (S, \rho :: \{x \mapsto \ell\}, f(\ell), \xi)$ si $\xi(\ell) = \bullet$

Deuxième partie

Types avec intersection

Chapitre 6

Calcul $\Lambda_{\mathcal{K}}$

Dans ce Chapitre, nous posons les premières bases de l'inférence de types avec intersection. Comme dans la première partie, nous commençons par y définir le langage utilisé dans toute la suite, en l'occurrence une version étendue du λ-calcul par un opérateur noté $[\_, \_]$ qui permet de ne pas "oublier" des sous-termes habituellement effacés dans le cas d'un β_K-redex. Nous y définissons également un système de types avec intersection standard pour ce calcul.

6.1 Motivation et historique

On peut grossièrement classifier les systèmes de types avec intersection en deux catégories : avec ou sans ω. Ceux de la deuxième catégorie, et celui que nous allons donner en particulier, vérifient la propriété que les termes typables sont exactement les termes fortement normalisables. Par exemple, le terme $M = F\ (\Delta\ \Delta)$, où $F = \lambda x \lambda y\ y$ et $\Delta = \lambda x\ (xx)$, n'est pas typable car il possède une réduction divergente.

Dans les chapitres suivants, nous allons définir un algorithme d'inférence qui "suit" les réductions du terme analysé. Or, dans le cas de M, il existe aussi une réduction convergente, à savoir $M \to \lambda y\ y$, qui "oublie" l'argument divergent $\Delta\ \Delta$, par l'intermédiaire d'une β_K-réduction (i.e. une réduction $(\lambda x M)\ N \to M$ avec $x \notin fv(M)$).

Si notre algorithme suit cette réduction en particulier, il ne devra cependant pas oublier ce sous-terme, sous peine de déclarer M typable. A cette fin, il nous faut donc un moyen de ne pas oublier les arguments des β_K-réductions. Cette possibilité existe déjà dans la littérature, sous la forme d'extensions du λ-calcul.

L'idée principale fut introduite par Klop dans [Klo80] ; celui-ci y définit son extension par l'ajout d'un opérateur de la forme $[M, N]$ au λ-calcul. Intuitivement, ce terme se comporte comme M dans n'importe quel contexte, mais N peut cependant donner lieu à des réductions. Pour traduire un terme du λ-calcul, Klop remplace chaque fonction $\lambda x M$ par le terme $\lambda x\ [M, x]$. Ainsi, dans le résultat, on est certain que pour chaque fonction $\lambda x M$, on a $x \in fv(M)$ et qu'on gardera donc au moins une copie de l'argument lors de chaque β-réduction.

La substitution de termes et les règles de réduction du calcul de Klop sont les règles usuelles du λ-calcul, à ceci près qu'on rajoute l'axiome suivant, nécessaire pour mettre en contact les deux membres d'une application :

$$[M, N]\ M' \to [M\ M', N]$$

Par exemple, la suite de réductions du λ-calcul :

$$(\lambda x\ I)\ A\ B\ C \to I\ B\ C \to B\ C$$

est donc remplacée par l'évolution suivante :

$$(\lambda x\ [I, x])\ A\ B\ C \to [I, A]\ B\ C \to [I\ B, A]\ C \to [I\ B\ C, A] \to [B\ C, A]$$

où l'on pourra noter que le terme A est toujours présent et peut donner lieu à des réductions s'il contient des redex. En revanche, il n'a plus aucun moyen d'interagir avec le reste du terme ; en ce sens, l'introduction de $[\_, \_]$ revient à gérer en même temps deux arbres syntaxiques distincts, l'un principal, l'autre secondaire.

Un des inconvénients de ce calcul est l'existence des réductions "administratives" de la forme $[M, N]\ M' \to [M\ M', N]$ qui ne correspondent pas à un véritable pas de calcul. Dans la même veine, d'autres calculs proposent l'équivalent de la règle $[M, N] \to M$, afin de pouvoir supprimer à tout moment les arguments non utilisés et se ramener à un terme du λ-calcul. Dans la suite de ce chapitre, nous définissons un calcul légèrement différent, qui évite ces réductions administratives, tout en gardant les idées principales du calcul de Klop. L'avantage est que la notion de redex reste exactement la même qu'en λ-calcul (modulo $[M, N]\ M' \equiv [M\ M', N]$), ce qui facilite d'autant la relation avec le λ-calcul ainsi que le typage et en particulier l'inférence.

Plus récemment, Kamareddine [Kam00] a repris et étendu les résultats de Nederpelt [KN95, BKN96] ainsi que d'autres en proposant une relation de réduction généralisée β_e pour le λ-calcul, associée à des notations adéquates. Elle y démontre notamment une propriété de report des β_K-réductions : grâce à la réduction généralisée, celles-ci peuvent être évitées et l'on garde ainsi leurs arguments présents comme dans le calcul de Klop. Des résultats de conservation et de préservation de la normalisation forte y sont également démontrés. Si les résultats sont plus généraux, nous préférons cependant garder un calcul simple, avec des notations proches des notations usuelles du λ-calcul, et bénéficiant d'une correspondance directe avec l'inférence de types.

6.2 Syntaxe et réduction

Soit $\mathcal{IV}ar$ un ensemble d'*identifieurs*, dénotés par les lettres $x, y, z, \ldots$.

Définition 6.1 *Pour mémoire, nous rappelons la grammaire du λ-calcul :*

$$M, N \in \Lambda ::= x \mid MN \mid \lambda x M$$

L'ensemble des variables libres est défini de façon usuelle par :

$$fv(x) = \{x\} \qquad fv(MN) = fv(M) \cup fv(N) \qquad fv(\lambda x M) = fv(M) \setminus \{x\}$$

On équipe ensuite ce calcul de sa relation de réduction habituelle :

$$(\lambda x M)\ N \longrightarrow_\beta M\{x \mapsto N\}$$

où $M\{x \mapsto N\}$ désigne la substitution usuelle des occurrences libres de x par N dans le terme M, avec renommage pour éviter toute capture de variables.

Lorsque $x \in fv(M)$ dans la fonction $\lambda x M$, nous sommes certains de garder une copie de l'argument lors d'une β-réduction ; il n'est donc pas nécessaire de transformer systématiquement cette fonction en $\lambda x\ [M, x]$ comme le fait Klop. A la place, nous choisissons d'introduire l'opérateur $[\_, \_]$ au moment de la réduction, uniquement dans le cas où $x \notin fv(M)$.

Définition 6.2 *La grammaire du* $\Lambda_{\mathcal{K}}$-calcul *est donnée par :*

$$M, N \in \Lambda_{\mathcal{K}} ::= x \mid MN \mid \lambda x M \mid [M, N]$$

On notera l'imbrication $[\cdots [M, N_1] \cdots, N_n]$ *sous la forme* $[M, N_1, \ldots, N_n]$*, y compris lorsque* $n = 0$*, auquel cas cette expression désignera* M *seul. L'ensemble des variables libres se définit comme en* λ*-calcul, avec la règle supplémentaire :*

$$fv([M, N]) = fv(M) \cup fv(N)$$

La relation de réduction $\longrightarrow_{\mathcal{K}}$ *est alors définie par les deux axiomes suivants :*

$$[\lambda x M, N_1, \ldots, N_n]\ N \ \longrightarrow_{\mathcal{K}}\ [M\{x \mapsto N\}, N_1, \ldots, N_n] \qquad \text{pour } x \in fv(M)$$

$$[\lambda x M, N_1, \ldots, N_n]\ N \ \longrightarrow_{\mathcal{K}}\ [M, N_1, \ldots, N_n, N] \qquad \text{pour } x \notin fv(M)$$

Exemple 6.3 *Soit* $M = (\lambda x\ F\ (x\ \Delta))\ \Delta$*, où* $F = \lambda x \lambda y\ y$ *et* $\Delta = \lambda x\ (xx)$*. On a par exemple :*

$$\begin{array}{rcl} M & \longrightarrow_{\mathcal{K}} & (\lambda x\ [\lambda y\ y,\ x\ \Delta])\ \Delta \\ & \longrightarrow_{\mathcal{K}} & [\lambda y\ y,\ \Delta\ \Delta] \\ & \longrightarrow_{\mathcal{K}} & [\lambda y\ y,\ \Delta\ \Delta] \\ & \longrightarrow_{\mathcal{K}} & \ldots \end{array}$$

alors qu'en λ*-calcul, on a les deux réductions :*

$$\begin{array}{rcl} M & \longrightarrow_{\beta} & (\lambda x\ \lambda y\ y)\ \Delta \\ & \longrightarrow_{\beta} & \lambda y\ y \end{array} \qquad\qquad \begin{array}{rcl} M & \longrightarrow_{\beta} & F\ (\Delta\ \Delta) \\ & \longrightarrow_{\beta} & F\ (\Delta\ \Delta) \\ & \longrightarrow_{\beta} & \ldots \end{array}$$

6.3 Propriétés de normalisation

Nous rappelons les définitions usuelles concernant la normalisation de termes, puis nous énonçons une propriété essentielle du $\Lambda_{\mathcal{K}}$-calcul.

Définition 6.4

- *Un terme est* normal *s'il ne peut être réduit. On note* $\mathcal{N}_{\Lambda}$ *et* $\mathcal{N}_{\mathcal{K}}$ *l'ensemble des termes normaux pour le* λ*-calcul et le* $\Lambda_{\mathcal{K}}$*-calcul respectivement.*
- *Un terme est dit (faiblement)* normalisant *(ou encore* normalisable*) s'il possède une réduction qui converge. On note* $\mathcal{WN}_{\Lambda}$ *et* $\mathcal{WN}_{\mathcal{K}}$ *l'ensemble des termes faiblement normalisants pour le* λ*-calcul et le* $\Lambda_{\mathcal{K}}$*-calcul respectivement.*
- *Un terme est dit* fortement normalisant *s'il ne possède pas de réduction infinie (i.e. divergente). On note* $\mathcal{SN}_{\Lambda}$ *et* $\mathcal{SN}_{\mathcal{K}}$ *l'ensemble des termes fortement normalisants pour le* λ*-calcul et le* $\Lambda_{\mathcal{K}}$*-calcul respectivement.*

Exemple 6.5 *Le terme* $M = (\lambda x\ F\ (x\ \Delta))\ \Delta$ *est normalisable, mais pas fortement normalisant en* λ*-calcul. En* $\Lambda_{\mathcal{K}}$*-calcul, il diverge toujours.*

Dans le λ-calcul, il est connu qu'il existe des termes normalisables mais non fortement normalisants. En revanche, pour le calcul de Klop, ces deux notions deviennent identiques :

Lemme 6.6
- $\mathcal{WN}_{\mathcal{K}} = \mathcal{SN}_{\mathcal{K}}$
- $\mathcal{SN}_{\Lambda} = \Lambda \cap \mathcal{SN}_{\mathcal{K}}$, *i.e. un λ-terme est fortement β-normalisant si et seulement s'il est fortement $\mathcal{K}$-normalisant.*

Preuve: Voir [Bou03]. □

6.4 Système de types

Soit $\mathcal{TV}ar$ un ensemble infini de *variables de type*, dénotées par les lettres $t, t', t_1, \ldots$.

Nous ne considérerons et nous n'aurons à manipuler que des *types premiers*, c'est-à-dire des types avec intersection pour lesquels la conjonction ne peut apparaître qu'à gauche d'une flèche. Ceux-ci sont décrits par la syntaxe suivante :

$$\overline{\tau}, \overline{\sigma} \in \mathcal{T}_p \ ::= \ t \ \mid \ \overline{\tau_1}, \ldots, \overline{\tau_n} \to \overline{\sigma}$$

Lorsque $n = 0$, on note $\omega \to \overline{\tau}$ un type avec une séquence vide. On identifie également les types obtenus par *commutativité* des arguments $\overline{\tau_i}$ dans une séquence [17].

Un *environnement* Γ est une suite de liaisons de la forme $x : \overline{\tau}$. On note $\Gamma(x)$ la séquence de types $\overline{\tau_1}, \ldots, \overline{\tau_n}$ lorsque les liaisons portant sur x dans Γ ont pour types $\overline{\tau_i}$. On note $\Gamma \setminus x$ l'environnement Γ dont on a supprimé toutes les liaisons portant sur x. Enfin, la *concaténation* d'environnements, notée Γ_1, Γ_2 est définie point à point par : $(\Gamma_1, \Gamma_2)(x) = \Gamma_1(x), \Gamma_2(x)$.

Il ne nous reste plus qu'à donner les règles du système de types pour le $\Lambda_{\mathcal{K}}$-calcul ; celles-ci respectent la structure des termes :

$$\frac{}{x : \overline{\tau} \vdash x : \overline{\tau}}\,(\text{Typ Id})$$

$$\frac{\Gamma \vdash M : \overline{\tau}}{\Gamma \setminus x \vdash \lambda x M : \Gamma(x) \to \overline{\tau}}\,(\text{Typ } \lambda)$$

$$\frac{\Gamma \vdash M : \overline{\tau_1}, \ldots, \overline{\tau_n} \to \overline{\sigma} \qquad \forall i,\ \Gamma_i \vdash N : \overline{\tau_i}}{\Gamma, \Gamma_1, \ldots, \Gamma_n \vdash MN : \overline{\sigma}}\,(\text{Typ Appl Gen}) \qquad (n \geq 1)$$

$$\frac{\Gamma \vdash M : \omega \to \overline{\sigma} \qquad \Gamma_1 \vdash N : \overline{\tau_1}}{\Gamma, \Gamma_1 \vdash MN : \overline{\sigma}}\,(\text{Typ Appl } \omega)$$

$$\frac{\Gamma_1 \vdash M_1 : \overline{\tau} \qquad \Gamma_2 \vdash M_2 : \overline{\sigma}}{\Gamma_1, \Gamma_2 \vdash [M_1, M_2] : \overline{\tau}}\,(\text{Typ Forget})$$

Les règles importantes sont (Typ Appl Gen) et (Typ Appl ω) : dans le premier cas, l'argument N est typé autant de fois qu'il y a de termes dans le type de M. Dans le deuxième cas, lorsque le type de M est de la forme $\omega \to \overline{\sigma}$, on requiert seulement que N soit typable avec un type $\overline{\tau_1}$ qui n'est pas utilisé dans la suite (cependant les liaisons apportées par l'environnement sont maintenues).

Enfin, nous pouvons énoncer la propriété principale vérifiée par ce système de types :

Théorème 6.7 *Soit M un terme du $\Lambda_{\mathcal{K}}$-calcul. M est typable si et seulement si M est fortement normalisable.*

[17] En revanche, il n'y a pas d'*idempotence* ou *contraction* : $t, t \to t'$ est bien différent de $t \to t'$.

Preuve: Voir [Bou03] pour la première implication. Pour la réciproque, on peut suivre les lignes de la preuve donnée dans [AC98] (basée sur une propriété d'expansion du sujet), adaptée au calcul de Klop. En λ-calcul pur, on pourra aussi trouver une preuve complète dans [Kri90], Chapitre IV, Théorème 6 et Corollaire, ou encore dans [Pot80] pour la première preuve publiée. □

Chapitre 7

Algorithme d'inférence

Dans ce chapitre, nous présentons l'algorithme d'inférence de types avec intersection qui construit un typage principal pour les termes normalisables du $\Lambda_{\mathcal{K}}$-calcul. Ce chapitre contient un grand nombre de définitions et des exemples d'application. La correction de l'algorithme ainsi que les preuves qui l'accompagnent seront présentées au chapitre suivant.

Nous commencerons par rappeler la syntaxe des types, et en distinguerons un sous-ensemble particulier, les *types premiers simples*. Puis nous introduirons la forme générale des *équations* que l'algorithme devra résoudre, ainsi que des *squelettes de preuve*, qui sont des arbres de typage dont certains noeuds ne sont pas corrects. Nous présenterons ensuite le mécanisme d'*indexation* permettant de créer des copies de variables de types. Puis, nous présenterons deux formes de *substitutions* (au sens usuel), ainsi que l'opération de *duplication*. Enfin, nous détaillerons les *règles de résolution* des équations proprement dites, ainsi que de nombreux exemples. Il ne nous restera qu'à voir comment construire l'ensemble initial d'équations correspondant à un terme, et nous aurons alors l'*algorithme d'inférence* complet, qui retourne un arbre de typage pour les termes typables. Enfin, nous donnerons quelques explications intuitives basées sur la *polarité*, permettant de comprendre le fonctionnement de l'algorithme, puis nous en définirons une variante à *rang fini*.

7.1 Types premiers simples

Nous rappelons la définition des types premiers introduits au chapitre précédent, qui sont ceux utilisés par le système de types :

$$\overline{\tau} \in \mathcal{T}_p ::= t \mid \overline{\tau_1}, \ldots, \overline{\tau_n} \to \overline{\sigma}$$

Cependant, le solveur de contraintes ne manipule en fait qu'un fragment de cette définition, les *types premiers simples*, pour lesquels seules des variables de type peuvent apparaître en position d'argument :

$$\tau \in \mathcal{T}_s ::= t \mid t_1, \ldots, t_n \to \tau$$

Il existe une injection triviale de $\mathcal{T}_s$ dans $\mathcal{T}_p$. Dans la suite, nous effectuerons parfois implicitement la conversion, et lorsque τ désignera un type premier simple, nous noterons (abusivement) $\overline{\tau}$ sa version non simple obtenue par l'injection.

7.2 Equations

Un *territoire* T est un ensemble *fini* de variables de types $\{t_1, \ldots, t_n\}$. Nous écrirons souvent cet ensemble sans accolades, mais au contraire des séquences, l'ordre des variables n'est pas important.

Voici la forme générale d'une *contrainte* ou *équation* :

$$eq ::= (\tau \to t \ \stackrel{\perp}{=} \ \sigma \ \ [T])$$

Elle se compose de deux types simples $\tau \to t$ et σ qu'intuitivement il va falloir unifier pour résoudre la contrainte. On remarquera que le type de gauche est toujours de la forme $\tau \to t$ où t est une variable de type, appelée le *noeud* de l'équation. Cette propriété sera un invariant pour l'algorithme d'inférence. Comme nous le verrons dans les preuves, il y aura toujours exactement une équation dans le système pour chaque application non encore réduite dans le terme à analyser. Si MN désigne une telle application, σ est en fait le type de M, τ celui de N, t est le type du noeud application, et le *territoire de l'équation* T est l'ensemble des variables de type apparaissant dans le sous-arbre correspondant à N.

Dans la suite, nous utiliserons la notation $\mathcal{E}$ pour désigner un ensemble fini d'équations.

Lors de la phase finale de l'algorithme, nous aurons à considérer une forme spéciale de contraintes, appelées *équations irréductibles*, pour lesquelles la partie gauche peut être un type quelconque $\overline{\tau}$ alors que la partie droite est restreinte aux variables de type (il n'y a plus de territoire, car celui-ci ne sera plus nécessaire lors de la phase finale de l'algorithme) :

$$\overline{eq} ::= (\overline{\tau} \ \stackrel{\perp}{=} \ t)$$

Nous désignerons les ensembles finis d'équations irréductibles par $\overline{\mathcal{E}}$.

7.3 Squelettes de preuve

Un *environnement* Γ est un ensemble (fini) de liaisons $x : \overline{\tau}$. Un même identifieur x peut avoir plusieurs liaisons. Si $\{\overline{\tau_1}, \ldots, \overline{\tau_n}\}$ sont les liaisons de x dans Γ, alors $\Gamma(x)$ est défini comme $\overline{\tau_1}, \ldots, \overline{\tau_n}$ (une séquence à utiliser à gauche d'un type flèche), et $\Gamma \setminus x$ est défini comme Γ dans lequel on a supprimé toutes les liaisons pour x. La concaténation d'environnements se note Γ, Γ' et est définie point à point : $(\Gamma, \Gamma')(x) = \Gamma(x), \Gamma'(x)$.

Les *squelettes de preuve* sont des squelettes d'arbres de typage, dans lesquels toutes les contraintes habituellement vérifiées ne sont pas correctes. Partant d'un squelette initial, ils seront progressivement transformés au fur et à mesure que les contraintes seront résolues, pour finalement aboutir à un arbre de typage valide :

$$\Pi \quad ::= \quad \frac{}{\Gamma \vdash x : \overline{\tau}}^{(\text{ID})} \ \mid \ \frac{\Pi \qquad \Pi_1 \ \ldots \ \Pi_n}{\Gamma \vdash MN : \overline{\tau}}^{(\text{APPL})} \ \ (n \geq 1)$$

$$\mid \ \frac{\Pi}{\Gamma \vdash \lambda x M : \overline{\tau}}^{(\text{FUN})} \ \mid \ \frac{\Pi_1 \qquad \Pi_2}{\Gamma \vdash [M, N] : \overline{\tau}}^{(\text{FORGET})}$$

En réalité, dans toute la suite, nous allons nous limiter à des squelettes de preuve vérifiant certaines conditions de cohérence. Nous allons imposer que les environnements de typage et les termes figurant dans les prédicats de typage d'un squelette de preuve soient cohérents entre eux, à la manière d'un arbre de typage classique. De plus, nous imposons également une cohérence sur les types pour les règles (FUN) et (FORGET). Tous les squelettes de preuve manipulés respecteront

donc la grammaire restreinte suivante, où les fonctions $Env(\Pi)$, $Term(\Pi)$ et $Typ(\Pi)$ désignent respectivement Γ, M et $\overline{\tau}$ lorsque le jugement de typage à la racine de Π est $\Gamma \vdash M : \overline{\tau}$.

$$
\begin{array}{rcl}
\Pi & ::= & \dfrac{}{x : \overline{\tau} \vdash x : \overline{\tau}}\,(\text{ID}) \\
& | & \dfrac{\Pi \qquad \Pi_1 \ \ldots \ \Pi_n}{Env(\Pi), Env(\Pi_1), \ldots, Env(\Pi_n) \vdash Term(\Pi)\ N : \overline{\tau}}\,(\text{APPL}) \\
& & \qquad\qquad \text{avec } n \geq 1 \text{ et } \forall i\ Term(\Pi_i) = N \\
& | & \dfrac{\Pi}{\Gamma \setminus x \vdash \lambda x\ Term(\Pi) : \Gamma(x) \rightarrow Typ(\Pi)}\,(\text{FUN}) \quad \text{avec } \Gamma = Env(\Pi) \\
& | & \dfrac{\Pi_1 \qquad \Pi_2}{Env(\Pi_1), Env(\Pi_2) \vdash [Term(\Pi_1), Term(\Pi_2)] : Typ(\Pi_1)}\,(\text{FORGET})
\end{array}
$$

On remarquera qu'un tel squelette de preuve est un arbre de typage valide, à l'exception des noeuds (APPL) [18] . Le but de l'algorithme de typage sera donc de générer des transformations sur le squelette de preuve de telle sorte que tous les noeuds (APPL) deviennent une instance valide soit de la règle (Typ Appl Gen), soit de la règle (Typ Appl ω).

Définition 7.1

- *Un squelette de preuve* Π *est un* arbre de typage *(sous-entendu* valide*) lorsque* Π *peut être dérivé à partir des règles du système de typages donné à la Section 6.4.*
- *Un prédicat* $\Gamma \vdash M : \overline{\tau}$ *est un* typage *pour* M *s'il existe un arbre de typage pour* M *dont la racine est ce prédicat.*

7.4 Indexation

Pour toute variable de type t et tout entier $n \in \mathbb{N}^+$, nous supposons qu'il existe une variable de type associée $\langle t \rangle^i$:

$$\langle - \rangle^- : \mathcal{TV}ar \times \mathbb{N}^+ \mapsto \mathcal{TV}ar$$

telle que cette opération soit injective pour ses deux arguments. Dans les exemples et lorsqu'il n'y aura pas de risque d'ambiguïté, nous abrégerons cette notation en t^i.

Bien que dans toute la suite, nous gardions simplement ce mécanisme abstrait, voyons comment il peut être défini dans la pratique. La manière la plus simple de construire une telle opération est de partir d'un ensemble infini Ω de variables de type de base et de définir $\mathcal{TV}ar$ comme étant $\Omega \times (\mathbb{N}^+)^*$, autrement dit les variables de type de base indexées par des chaînes d'entiers de $\mathbb{N}^+$. Une variable de type de base t est alors identifiée avec t^ϵ, et $\langle t^s \rangle^i$ est défini comme $t^{s.i}$, pour $s \in (\mathbb{N}^+)^*$.

Cependant, comme nous n'avons pas vraiment besoin de conserver toute l'information contenue dans la chaîne, nous n'utiliserons pas ce mécanisme d'indexation avec un tel niveau de détail. Il est également possible de considérer que les indexations de variables sont construites à la demande, autrement dit chaque fois que nous avons besoin de calculer une indexation $\langle t \rangle^i$ non rencontrée jusqu'alors, nous choisissons une variable fraîche $t' \in \mathcal{TV}ar$ et nous stockons la liaison dans une table pour les usages futurs. Ceci est effectivement correct car l'algorithme ne réclame qu'un nombre fini d'indexations (lorsqu'il converge).

Une fois définie l'opération d'indexation sur les variables de type, il est possible de l'étendre trivialement à toute structure construite à partir de ces variables, notamment les types, les équations,

[18] Kfoury et Wells mélangent d'ailleurs ces deux notions, avec un système de types contenant deux versions pour l'application, l'une pour le typage véritable et l'autre pour leur équivalent des squelettes de preuve.

les squelettes de preuve, ... Si X désigne une telle structure, nous noterons logiquement $\langle X \rangle^i$ cette extension, et nous noterons également :

$$dupl_n(X) = \langle X \rangle^1, \ldots, \langle X \rangle^n \text{ ou } \{\langle X \rangle^1, \ldots, \langle X \rangle^n\}$$

suivant le contexte d'utilisation (si on attend une séquence ou un ensemble). Lorsque X lui-même désigne un ensemble, cette notation signifie :

$$dupl_n(X) = \langle X \rangle^1 \cup \ldots \cup \langle X \rangle^n$$

ou de façon équivalente :

$$dupl_n(X) = \bigcup_{x \in X} dupl_n(x)$$

Enfin, $ftv(X)$ est défini comme l'ensemble de *variables de type libres* apparaissant dans X (défini trivialement par induction sur X).

7.5 Substitutions

Pour chaque sorte de types (τ et $\overline{\tau}$), il existe une notion correspondante de *substitution* :

$$S \in \mathcal{S}_s ::= \{t \mapsto \tau, T\} \mid D(n, T)$$

$$\overline{S} \in \mathcal{S}_p ::= \{t \mapsto \overline{\tau}\} \mid D(n, T)$$

Dans chaque cas, la première forme est appelée une *substitution simple*, tandis que $D(n, T)$ est appelée une *duplication*.

Comme pour $\mathcal{T}_s$ et $\mathcal{T}_p$, il existe une injection triviale de $\mathcal{S}_s$ dans $\mathcal{S}_p$ (en supprimant l'information de territoire pour les substitutions simples), et nous effectuerons implicitement la conversion lorsque nécessaire.

La composition de substitutions se note $S :: S'$ et $\overline{S} :: \overline{S'}$ (S' étant appliquée en premier). Nous écrirons aussi $\{t_i \mapsto X_i\}_{1 \leq i \leq n}$ pour $\{t_1 \mapsto X_1\} :: \ldots :: \{t_n \mapsto X_n\}$.

Le *domaine* d'une substitution est un sous-ensemble de $\mathcal{TV}ar$, et est défini comme $\{t\}$ pour $\{t \mapsto \tau, T\}$, comme T pour $D(n, T)$, et comme $dom(S) \cup dom(S')$ pour $S :: S'$ (et de même pour $\overline{S}$).

Appliquer une substitution S La première sorte de substitution peut être appliquée à des types premiers simples, aux territoires et aux équations. Les définitions dans le cas d'une substitution simple sont les suivantes :

Pour $S = \{t \mapsto \tau, T\}$:

$$\begin{cases} S(t) = \tau \\ S(t') = t' \quad \text{si } t' \neq t \\ S(t_1, \ldots, t_n \to \sigma) = t_1, \ldots, t_n \to S(\sigma) \\ S(T') = \begin{cases} (T' \setminus \{t\}) \cup T & \text{si } t \in T' \\ T' & \text{sinon} \end{cases} \\ S(\tau' \to t' \ \bot \ \sigma' \ \ [T']) = \{S(\tau') \to t' \ \bot \ S(\sigma') \ \ [S(T')]\} \end{cases}$$

Intuitivement, la variable t est remplacée par τ lorsqu'elle apparaît seule dans un type ou à droite d'une flèche, et par T dans les territoires. On prendra garde au fait que t n'est pas remplacée

à gauche des flèches ni en tant que noeud d'une équation [19]. On remarquera aussi que par conséquent l'application d'une telle substitution à un type simple donne bien un type simple, et que l'application à une équation donne un ensemble d'équations (ici un singleton).

Dans le cas d'une duplication, on utilise les définitions suivantes :

Pour $S = D(n, T)$:

$$\begin{cases} S(t) = t \\ S(t_1, \ldots, t_p \to \tau) = D'(t_1), \ldots, D'(t_p) \to S(\tau) \qquad \text{(avec associativité implicite)} \\ \quad \text{où } D'(t) = \begin{cases} dupl_n(t) & \text{si } t \in T \\ t & \text{sinon} \end{cases} \\ S(T') = \{t_i \mapsto t_i, dupl_n(t_i)\}_{t_i \in T}(T') \\ S(\tau' \to t' \ \stackrel{\perp}{=} \ \sigma' \ \ [T']) = \begin{cases} dupl_n(\tau' \to t' \ \stackrel{\perp}{=} \ \sigma' \ \ [T']) & \text{si } t' \in T \\ \{S(\tau') \to t' \ \stackrel{\perp}{=} \ S(\sigma') \ \ [S(T')]\} & \text{sinon} \end{cases} \end{cases}$$

Intuitivement, les variables de T sont dupliquées n fois lorsqu'elle apparaissent à gauche d'une flèche ou dans un territoire. En ce qui concerne les équations, elles sont dupliquées en entier si leur noeud est dans T, sinon on applique la substitution aux sous-composantes. Comme précédemment, on notera que l'application de S à un type simple donne bien un type simple, et un ensemble d'équations si S est appliquée à une équation.

Enfin, pour un ensemble d'équations $\mathcal{E}$, on définit :

$$S(\mathcal{E}) = \bigcup_{eq \in \mathcal{E}} S(eq)$$

Exemple 7.2 *(Ces exemples sont artificiels et ne peuvent pas exister dans une utilisation normale de l'algorithme.)*

Soit $S = \{t_1 \mapsto t_3, \{t_3, t_4\}\}$ *et* $\tau = t_1, t_2 \to t_1$*. L'application de* S *à* τ *donne :*

$$\begin{aligned} S(\tau) &= t_1, t_2 \to S(t_1) \\ &= t_1, t_2 \to t_3 \end{aligned}$$

Soit également l'équation $eq = (\tau \to t_1 \ \stackrel{\perp}{=} \ \tau \ \ [t_1, t_2])$*. L'application de* S *à* eq *donne :*

$$\begin{aligned} S(eq) &= \{S(\tau) \to t_1 \ \stackrel{\perp}{=} \ S(\tau) \ \ [S(\{t_1, t_2\})]\} \\ &= \{(t_1, t_2 \to t_3) \to t_1 \ \stackrel{\perp}{=} \ t_1, t_2 \to t_3 \ \ [t_3, t_4, t_2]\} \end{aligned}$$

Prenons maintenant $S' = D(2, \{t_1\})$*. L'application de* S' *à* τ *donne :*

$$\begin{aligned} S'(\tau) &= dupl_2(t_1), t_2 \to S'(t_1) \\ &= t_1^1, t_1^2, t_2 \to t_1 \end{aligned}$$

Soit le système à deux équations :

$$\mathcal{E} = \left\{ \begin{array}{lcll} \tau \to t_1 & \stackrel{\perp}{=} & \tau & [t_1, t_2], \\ \tau \to t_2 & \stackrel{\perp}{=} & \tau & [t_1, t_2] \end{array} \right\}$$

[19] En réalité, dans notre système, il est garanti que t ne peut pas apparaître dans ces positions (ceci sera justifié plus loin par des arguments de polarité). Deux choix de conception sont ici possibles : donner une définition classique de la substitution en l'appliquant partout, puis justifier a posteriori grâce aux polarités que le résultat a bien la forme attendue ; ou alors donner une définition inhabituelle de la substitution qui se base sur ces propriétés. Nous avons privilégié la deuxième solution car elle simplifie les preuves.

Comme le noeud t_1 de la première équation est dans le domaine de S', celle-ci est dupliquée ; en revanche, la deuxième équation ne l'est pas. L'application de S' à $\mathcal{E}$ donne donc :

$$S'(\mathcal{E}) = \left\{ \begin{array}{ccccc} (t_1^1, t_2^1 \to t_1^1) \to t_1^1 & \doteq & t_1^1, t_2^1 \to t_1^1 & & [t_1^1, t_2^2], \\ (t_1^2, t_2^2 \to t_1^2) \to t_1^2 & \doteq & t_1^2, t_2^2 \to t_1^2 & & [t_1^2, t_2^2], \\ (t_1^1, t_1^2, t_2 \to t_1) \to t_2 & \doteq & t_1^1, t_1^2, t_2 \to t_1 & & [t_1^1, t_1^2, t_2] \end{array} \right\}$$

Appliquer une substitution $\overline{S}$ La deuxième forme de substitutions s'applique aux types premiers, aux environnements, aux squelettes de preuve et aux équations irréductibles. Pour une substitution simple, on pose :

Pour $\overline{S} = \{t \mapsto \overline{\tau}\}$:

$$\left\{ \begin{array}{l} \overline{S}(t) = \overline{\tau} \\ \overline{S}(t') = t' \qquad \text{if } t' \neq t \\ \overline{S}(\overline{\sigma_1}, \ldots, \overline{\sigma_n} \to \overline{\sigma}) = \overline{S}(\overline{\sigma_1}), \ldots, \overline{S}(\overline{\sigma_n}) \to \overline{S}(\overline{\sigma}) \\ \overline{S}(x_1 : \overline{\tau_1}, \ldots, x_n : \overline{\tau_n}) = x_1 : \overline{S}(\overline{\tau_1}), \ldots, x_n : \overline{S}(\overline{\tau_n}) \\ \overline{S}\left(\dfrac{}{\Gamma \vdash x : \overline{\tau}}^{(\text{ID})}\right) = \dfrac{}{\overline{S}(\Gamma) \vdash x : \overline{S}(\overline{\tau})}^{(\text{ID})} \\ \overline{S}\left(\dfrac{\Pi \quad \Pi_1 \ldots \Pi_n}{\Gamma \vdash MN : \overline{\tau}}^{(\text{APPL})}\right) = \dfrac{\overline{S}(\Pi) \quad \overline{S}(\Pi_1) \ldots \overline{S}(\Pi_n)}{\overline{S}(\Gamma) \vdash MN : \overline{S}(\overline{\tau})}^{(\text{APPL})} \\ \overline{S}\left(\dfrac{\Pi}{\Gamma \vdash \lambda x M : \overline{\tau}}^{(\text{FUN})}\right) = \dfrac{\overline{S}(\Pi)}{\overline{S}(\Gamma) \vdash \lambda x M : \overline{S}(\overline{\tau})}^{(\text{FUN})} \\ \overline{S}\left(\dfrac{\Pi_1 \quad \Pi_2}{\Gamma \vdash [M, N] : \overline{\tau}}^{(\text{FORGET})}\right) = \dfrac{\overline{S}(\Pi_1) \quad \overline{S}(\Pi_2)}{\overline{S}(\Gamma) \vdash [M, N] : \overline{S}(\overline{\tau})}^{(\text{FORGET})} \\ \overline{S}(\overline{\tau'} \doteq t') = (\overline{S}(\overline{\tau'}) \doteq t') \end{array} \right.$$

Cette fois-ci, pas de surprise : la substitution de t par $\overline{\tau}$ s'effectue partout où la variable t apparaît (excepté pour les équations irréductibles, où elle ne s'applique que sur le type de gauche).

Pour les duplications, on pose :

Pour $\overline{S} = D(n, T)$:

$$\left\{ \begin{array}{l} \overline{S}(t) = t \\ \overline{S}(\overline{\tau_1}, \ldots, \overline{\tau_p} \to \overline{\tau}) = D'(\overline{\tau_1}), \ldots, D'(\overline{\tau_p}) \to \overline{S}(\overline{\tau}) \qquad \text{(avec associativité implicite)} \\ \qquad \text{où } D'(\overline{\tau}) = \begin{cases} dupl_n(\overline{\tau}) & \text{si } ftv(\overline{\tau}) \subseteq T \\ \overline{S}(\overline{\tau}) & \text{sinon} \end{cases} \\ \overline{S}(x_1 : \overline{\tau_1}, \ldots, x_n : \overline{\tau_n}) = D'(x_1 : \overline{\tau_1}), \ldots, D'(x_n : \overline{\tau_n}) \qquad \text{(avec associativité implicite)} \\ \qquad \text{où } D'(x : \overline{\tau}) = \begin{cases} dupl_n(x : \overline{\tau}) & \text{si } ftv(\overline{\tau}) \subseteq T \\ x : \overline{S}(\overline{\tau}) & \text{sinon} \end{cases} \end{array} \right.$$

$$\begin{cases} \overline{S}\left(\dfrac{}{\Gamma \vdash x : \overline{\tau}}^{(\text{ID})}\right) = \dfrac{}{\overline{S}(\Gamma) \vdash x : \overline{S}(\overline{\tau})}^{(\text{ID})} \\ \overline{S}\left(\dfrac{\Pi \quad \Pi_1 \dots \Pi_p}{\Gamma \vdash MN : \overline{\tau}}^{(\text{APPL})}\right) = \dfrac{\overline{S}(\Pi) \quad D'(\Pi_1) \dots D'(\Pi_p)}{\overline{S}(\Gamma) \vdash MN : \overline{S}(\overline{\tau})}^{(\text{APPL})} \\ \quad \text{où } D'(\Pi) = \begin{cases} dupl_n(\Pi) & \text{si } ftv(\Pi) \subseteq T \\ \overline{S}(\Pi) & \text{sinon} \end{cases} \\ \overline{S}\left(\dfrac{\Pi}{\Gamma \vdash \lambda x M : \overline{\tau}}^{(\text{FUN})}\right) = \dfrac{\overline{S}(\Pi)}{\overline{S}(\Gamma) \vdash \lambda x M : \overline{S}(\overline{\tau})}^{(\text{FUN})} \\ \overline{S}\left(\dfrac{\Pi_1 \quad \Pi_2}{\Gamma \vdash [M, N] : \overline{\tau}}^{(\text{FORGET})}\right) = \dfrac{\overline{S}(\Pi_1) \quad \overline{S}(\Pi_2)}{\overline{S}(\Gamma) \vdash [M, N] : \overline{S}(\overline{\tau})}^{(\text{FORGET})} \\ \overline{S}(\overline{\tau} \doteq t) = \text{indéfini (car non nécessaire)} \end{cases}$$

Ces définitions sont similaires à celles données pour la première forme de duplication : on duplique uniquement à gauche des flèches et dans les environnements. A noter cependant que puisqu'on peut avoir des types quelconques à gauche d'une flèche, la condition de duplication porte maintenant sur l'ensemble des variables libres (du style $ftv(X) \subseteq T$ au lieu de $t \in T$). En ce qui concerne les squelettes de preuves, on notera qu'il n'existe qu'un seul point de duplication potentielle, à savoir le(s) sous-arbre(s) correspondant(s) à l'argument dans une application.

Enfin, pour un ensemble d'équations irréductibles $\overline{\mathcal{E}}$:

$$\overline{S}(\overline{\mathcal{E}}) = \bigcup_{\overline{eq} \in \overline{\mathcal{E}}} \overline{S}(\overline{eq})$$

Exemple 7.3 *Soit* $\overline{S} = \{t_1 \mapsto t_3 \to t_3\}$ *et* $\overline{\tau} = (t_1 \to t_2), t_2 \to t_1$. *L'application de* $\overline{S}$ *à* $\overline{\tau}$ *donne :*

$$\overline{S}(\overline{\tau}) = ((t_3 \to t_3) \to t_2), t_2 \to t_3 \to t_3$$

Prenons maintenant $\overline{S'} = D(2, \{t_1, t_2\})$. *L'application de* $\overline{S'}$ *à* $\overline{\tau}$ *donne :*

$$\overline{S'}(\overline{\tau}) = (t_1^1 \to t_2^1), (t_1^2 \to t_2^2), t_2^1, t_2^2 \to t_1$$

Soit le squelette de preuve suivant :

$$\Pi = \dfrac{\dfrac{}{x : t_1 \vdash x : t_1}^{(ID)} \qquad \dfrac{}{x : t_2 \vdash x : t_2}^{(ID)}}{x : t_1, x : t_2 \vdash x\ x : \overline{\tau}}^{(APPL)}$$

L'application de $\overline{S'}$ *à* Π *donne le résultat suivant :*

$$\overline{S'}(\Pi) = \dfrac{\dfrac{}{x : t_1 \vdash x : t_1}^{(ID)} \qquad \dfrac{}{x : t_2^1 \vdash x : t_2^1}^{(ID)} \qquad \dfrac{}{x : t_2^2 \vdash x : t_2^2}^{(ID)}}{x : t_1, x : t_2^1, x : t_2^2 \vdash x\ x : (t_1^1 \to t_2^1), (t_1^2 \to t_2^2), t_2^1, t_2^2 \to t_1}^{(APPL)}$$

On remarquera la duplication du sous-arbre droit puisque ses variables libres sont dans le domaine de $\overline{S'}$.

Principalité Nous verrons par la suite que l'algorithme retourne des typages les plus généraux possibles ; on dit alors qu'ils sont *principaux*. Pour les systèmes de types simples en λ-calcul, la notion de principalité se résume à effectuer des substitutions simples sur les types. Dans le cadre des types avec intersection, cette notion est plus complexe et définie diversement suivant les auteurs [Jim96, Wel02]. L'ajout des duplications aux substitutions simples nous permet d'en donner une définition concise :

Définition 7.4

- *Un typage* $\Gamma \vdash M : \overline{\tau}$ *est dit* principal *si c'est un typage pour* M *et si, pour tout typage* $\Gamma' \vdash M : \overline{\tau'}$*, il existe une substitution* $\overline{S}$ *telle que* $\Gamma' = \overline{S}(\Gamma)$ *et* $\overline{\tau'} = \overline{S}(\overline{\tau})$.
- *Un arbre de typage* Π *pour un terme* M *est dit* principal *si, pour tout arbre de typage* Π' *de* M*, il existe une substitution* $\overline{S}$ *telle que* $\Pi' = \overline{S}(\Pi)$.

7.6 Règles de résolution des contraintes

Nous présentons maintenant le cœur du système, à savoir les règles qui permettent de décomposer et de résoudre les contraintes.

Un *état* du système se compose d'équations et d'un squelette de preuve : $(\mathcal{E},\ \Pi)$. Une équation de $\mathcal{E}$ dont la partie droite ne se réduit pas à une variable de type est dite *décomposable* (elle est de la forme $\tau \to t \ \stackrel{\perp}{=}\ t_1, \ldots, t_n \to \sigma \ \ [T]$). Si une telle équation existe dans $\mathcal{E}$, le système peut évoluer (si plusieurs équations satisfont la condition, on choisit une réduction de façon non déterministe [20]). La règle de réduction à utiliser et le résultat dépendent de la valeur de n :

Pour $n \geq 1$:

$$\boxed{\begin{array}{c}(\{\tau \to t \ \stackrel{\perp}{=}\ t_1, \ldots, t_n \to \sigma \ \ [T]\} \cup \mathcal{E},\ \Pi) \ \longrightarrow\ (S(\mathcal{E}),\ \overline{S}(\Pi)) \\ \text{avec } S = \{t_i \mapsto \langle\tau\rangle^i, \langle T\rangle^i\}_{1 \leq i \leq n} :: \{t \mapsto \sigma, \emptyset\} :: D(n, T)\end{array}} \quad (R_n)$$

Pour $n = 0$:

$$\boxed{\begin{array}{c}(\{\tau \to t \ \stackrel{\perp}{=}\ \omega \to \sigma \ \ [T]\} \cup \mathcal{E},\ \Pi) \ \longrightarrow\ (S(\mathcal{E}),\ \overline{S}(\Pi)) \\ \text{avec } S = \{t \mapsto \sigma, \emptyset\}\end{array}} \quad (R_0)$$

(On notera que dans les deux cas, la substitution $\overline{S}$ appliquée à Π est une conversion obtenue à partir de S.)

Intuitivement, dans la première règle, on commence par dupliquer toutes les variables et équations correspondant à l'argument de l'application (dont la liste est donnée par le territoire T) ; puis le noeud de l'équation est remplacé par le type-résultat de la fonction ; enfin chacun des types-arguments de cette fonction est remplacé par la i^{e} copie de τ (on verra dans la suite que partant d'un état initial convenablement choisi, ces variables dupliquées seront fraîches vis-à-vis du reste des équations). Le cas de la règle (R_0) est plus simple, puisqu'on n'effectue aucune duplication (cependant, les équations correspondant à l'argument sont conservées dans $\mathcal{E}$).

Lorsque $n = 1$, la règle (R_n) ci-dessus ne génère en réalité qu'un renommage des variables de T (ceci peut être prouvé formellement à partir du cas particulier des définitions pour $D(1, T)$ données plus haut). Par conséquent, comme ce cas se présente fréquemment dans la pratique, nous utiliserons la règle simplifiée suivante pour l'implémentation et les exemples afin de limiter le nombre de variables nouvelles créées :

$$\boxed{\begin{array}{c}(\{\tau \to t \ \stackrel{\perp}{=}\ t_1 \to \sigma \ \ [T]\} \cup \mathcal{E},\ \Pi) \ \longrightarrow\ (S(\mathcal{E}),\ \overline{S}(\Pi)) \\ \text{avec } S = \{t_1 \mapsto \tau, T\} :: \{t \mapsto \sigma, \emptyset\}\end{array}} \quad (R_1)$$

[20] Comme on le verra dans les résultats du chapitre suivant, ce choix n'est pas crucial car, pour un terme normalisable, le résultat final de l'algorithme ne dépend pas de l'ordre de résolution des contraintes. En revanche, cela peut avoir une incidence sur le nombre total d'étapes nécessaires à l'algorithme pour converger.

Cependant, dans les preuves, afin de limiter le nombre de cas à considérer, nous continuerons à n'utiliser que les règles générales (R_n) et (R_0).

Concernant l'évolution globale du système, il se peut que nous puissions appliquer (R_n) et (R_0) indéfiniment ; dans ce cas, on dira que le système *diverge*. Sinon, il *converge* et on finit par atteindre un état dans lequel aucune de ces règles ne s'applique (i.e. lorsque tous les membres droits des équations sont réduits à des variables de type). On est alors en présence d'un système d'équations irréductibles de la forme $(\overline{\mathcal{E}},\ \Pi)$ (on notera la conversion implicite sur les types, et la suppression des territoires, devenus inutiles) et on utilise la règle de réduction suivante, chargée de résoudre les contraintes restantes :

$$\boxed{(\{\overline{\tau}\ \doteq\ t\} \cup \overline{\mathcal{E}},\ \Pi)\ \longrightarrow_f\ (\overline{S}(\overline{\mathcal{E}}),\ \overline{S}(\Pi)) \qquad \text{avec } \overline{S} = \{t \mapsto \overline{\tau}\}} \quad (R_f)$$

Il est facile de vérifier qu'on ne peut appliquer (R_f) qu'un nombre fini de fois car le nombre d'équations restantes décroît de un à chaque étape. On s'arrête donc lorsqu'on atteint un état de la forme $(\emptyset,\ \Pi)$. Le *résultat* de l'inférence est alors Π.

Exemple 7.5 *Soit à réduire le système* $(\mathcal{E}_0, \Pi_0)$ *avec :*

$$\mathcal{E}_0 = \left\{ \begin{array}{rcll} t_1 \to t_2 & \doteq & t_0 & [t_1], \\ (\omega \to t_3) \to t_4 & \doteq & t_0, t_1 \to t_2 & [t_3] \end{array} \right\}$$

$$\Pi_0 = \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : t_0 \vdash x : t_0}\,(ID) \qquad \dfrac{}{x : t_1 \vdash x : t_1}\,(ID)}{x : t_0, x : t_1 \vdash x\ x : t_2}\,(APPL)}{\vdash \Delta : t_0, t_1 \to t_2}\,(FUN) \qquad \dfrac{\dfrac{}{y : t_3 \vdash y : t_3}\,(ID)}{y : t_3 \vdash \lambda z\ y : \omega \to t_3}\,(FUN)}{y : t_3 \vdash \Delta\ (\lambda z\ y) : t_4}\,(APPL)$$

où $\Delta = \lambda x\ (xx)$.

Seule la deuxième équation est décomposable, c'est la règle (R_n) *avec* $n = 2$ *qui s'applique. On génère donc la substitution :*

$$S_0 = \{t_0 \mapsto (\omega \to t_3^1), \{t_3^1\}\} :: \{t_1 \mapsto (\omega \to t_3^2), \{t_3^2\}\} :: \{t_4 \mapsto t_2, \emptyset\} :: D(2, \{t_3\})$$

et le système évolue en $(\mathcal{E}_1, \Pi_1)$ *où :*

$$\begin{array}{rcl} \mathcal{E}_1 & = & S_0(\{t_1 \to t_2\ \doteq\ t_0\ \ [t_1]\}) \\ & = & \{(\omega \to t_3^2) \to t_2\ \doteq\ \omega \to t_3^1\ \ [t_3^2]\} \end{array}$$

$$\begin{array}{rcl} \Pi_1 & = & \overline{S_0}(\Pi_0) \\ & = & \dfrac{\dfrac{\dfrac{\dfrac{}{x : \omega \to t_3^1 \vdash x : \omega \to t_3^1}\,(ID) \qquad \dfrac{}{x : \omega \to t_3^2 \vdash x : \omega \to t_3^2}\,(ID)}{x : \omega \to t_3^1, x : \omega \to t_3^2 \vdash x\ x : t_2}\,(APPL)}{\vdash \Delta : (\omega \to t_3^1), (\omega \to t_3^2) \to t_2}\,(FUN) \qquad \Pi_1' \qquad \Pi_1''}{y : t_3^1, y : t_2^3 \vdash \Delta\ (\lambda z\ y) : t_2}\,(APPL) \end{array}$$

$$\textit{avec} \quad \Pi_1' = \dfrac{\dfrac{}{y : t_3^1 \vdash y : t_3^1}\,(ID)}{y : t_3^1 \vdash \lambda z\ y : \omega \to t_3^1}\,(FUN) \quad \textit{et} \quad \Pi_1'' = \dfrac{\dfrac{}{y : t_3^2 \vdash y : t_3^2}\,(ID)}{y : t_3^2 \vdash \lambda z\ y : \omega \to t_3^2}\,(FUN)$$

L'équation restante dans $\mathcal{E}_1$ est décomposable par la règle (R_0) cette fois-ci. Ceci génère la substitution :

$$S_1 = \{t_2 \mapsto t_3^1, \emptyset\}$$

et le système évolue en $(\emptyset, \Pi_2)$ où :

$$\Pi_2 = \overline{S_1}(\Pi_1)$$

$$= \dfrac{\dfrac{\dfrac{\overline{x : \omega \to t_3^1 \vdash x : \omega \to t_3^1}^{(ID)} \quad \overline{x : \omega \to t_3^2 \vdash x : \omega \to t_3^2}^{(ID)}}{x : \omega \to t_3^1, x : \omega \to t_3^2 \vdash x\ x : t_3^1}^{(APPL)}}{\vdash \Delta : (\omega \to t_3^1), (\omega \to t_3^2) \to t_3^1}^{(FUN)} \quad \Pi_1' \quad \Pi_1''}{y : t_3^1, y : t_2^3 \vdash \Delta\ (\lambda z\ y) : t_3^1}^{(APPL)}$$

Comme il n'y a plus d'équation à décomposer, l'algorithme termine ici et renvoie Π_2 comme résultat. On peut vérifier qu'il s'agit d'un arbre de typage valide pour le terme $\Delta(\lambda zy)$.

Exemple 7.6 *Soit à réduire le système $(\mathcal{E}_0, \Pi_0)$ suivant :*

$$\mathcal{E}_0 = \left\{ \begin{array}{rcll} t_2 \to t_3 & \doteq & t_1 & [t_2], \\ t_4 \to t_5 & \doteq & t_1, t_2 \to t_3 & [t_4], \\ t_6 \to t_7 & \doteq & t_5 & [t_6], \\ (t_4, t_6 \to t_7) \to t_8 & \doteq & t_0 \to t_0 & [t_1, t_2, t_3, t_4, t_5, t_6, t_7] \end{array} \right\}$$

$$\Pi_0 = \dfrac{\Pi_0' \quad \Pi_0''}{\vdash I(\lambda y(\Delta y)y) : t_8}^{(APPL)} \quad avec \quad \Pi_0' = \dfrac{\overline{x : t_0 \vdash x : t_0}^{(ID)}}{\vdash I : t_0 \to t_0}^{(FUN)}$$

$$et \quad \Pi_0'' = \dfrac{\dfrac{\Pi_0''' \quad \dfrac{\overline{y : t_4 \vdash y : t_4}^{(ID)}}{y : t_4 \vdash \Delta y : t_5}^{(APPL)} \quad \overline{y : t_6 \vdash y : t_6}^{(ID)}}{y : t_4, y : t_6 \vdash (\Delta y)y : t_7}^{(APPL)}}{\vdash \lambda y(\Delta y)y : t_4, t_6 \to t_7}^{(FUN)}$$

$$et \quad \Pi_0''' = \dfrac{\dfrac{\overline{x : t_1 \vdash x : t_1}^{(ID)} \quad \overline{x : t_2 \vdash x : t_2}^{(ID)}}{x : t_1, x : t_2 \vdash x\ x : t_3}^{(APPL)}}{\vdash \Delta : t_1, t_2 \to t_3}^{(FUN)}$$

(comme on pourra le vérifier par la suite, il s'agit en réalité de l'état initial correspondant au terme $I(\lambda y(\Delta y)y)$ où $I = \lambda xx$)

La deuxième et la quatrième équation de $\mathcal{E}_0$ sont décomposables. Nous choisissons de commencer par la deuxième. Celle-ci génère par la règle (R_n) avec $n = 2$ la substitution suivante :

$$S_0 = \{t_1 \mapsto t_4^1, \{t_4^1\}\} :: \{t_2 \mapsto t_4^2, \{t_4^2\}\} :: \{t_5 \mapsto t_3, \emptyset\} :: D(2, \{t_4\})$$

Appliquée aux trois équations restantes de $\mathcal{E}_0$, nous obtenons le nouveau système suivant :

$$\mathcal{E}_1 = \left\{ \begin{array}{rcll} t_4^2 \to t_3 & \doteq & t_4^1 & [t_4^2], \\ t_6 \to t_7 & \doteq & t_3 & [t_6], \\ (t_4^1, t_4^2, t_6 \to t_7) \to t_8 & \doteq & t_0 \to t_0 & [t_3, t_4^1, t_4^2, t_6, t_7] \end{array} \right\}$$

Après application de $\overline{S_0}$, le nouveau squelette de preuve devient quant à lui :

$$\Pi_1 = \dfrac{\Pi_0' \quad \Pi_1''}{\vdash I(\lambda y(\Delta y)y) : t_8}^{(APPL)}$$

$$avec \quad \Pi_1'' = \dfrac{\dfrac{\dfrac{\Pi_1'''}{y : t_4^1, y : t_4^2 \vdash \Delta y : t_3}\,(APPL) \qquad \dfrac{}{y : t_6 \vdash y : t_6}\,(ID)}{y : t_4^1, y : t_4^2, y : t_6 \vdash (\Delta y) y : t_7}\,(APPL)}{\vdash \lambda y (\Delta y) y : t_4^1, t_4^2, t_6 \to t_7}\,(FUN)$$

$$et \quad \Pi_1''' = \dfrac{\dfrac{\dfrac{}{x : t_4^1 \vdash x : t_4^1}\,(ID) \qquad \dfrac{}{x : t_4^2 \vdash x : t_4^2}\,(ID)}{x : t_4^1, x : t_4^2 \vdash x\ x : t_3}\,(APPL)}{\vdash \Delta : t_4^1, t_4^2 \to t_3}\,(FUN)$$

Dans $\mathcal{E}_1$, seule la troisième équation est décomposable : nous pouvons utiliser ici la règle simplifiée (R_1). Elle génère la substitution :

$$S_1 = \{t_0 \mapsto (t_4^1, t_4^2, t_6 \to t_7), \{t_3, t_4^1, t_4^2, t_6, t_7\}\} :: \{t_8 \mapsto t_0, \emptyset\}$$

Celle-ci ne provoque aucun changement sur les deux équations restantes :

$$\mathcal{E}_2 = \left\{ \begin{array}{lcll} t_4^2 \to t_3 & \stackrel{\perp}{=} & t_4^1 & [t_4^2], \\ t_6 \to t_7 & \stackrel{\perp}{=} & t_3 & [t_6] \end{array} \right\}$$

et les modifications suivantes sur le squelette de preuve :

$$\Pi_2 = \dfrac{\Pi_2' \qquad \Pi_1''}{\vdash I(\lambda y(\Delta y)y) : t_4^1, t_4^2, t_6 \to t_7}\,(APPL) \quad avec \quad \Pi_2' = \dfrac{\dfrac{}{x : t_4^1, t_4^2, t_6 \to t_7 \vdash x : t_4^1, t_4^2, t_6 \to t_7}\,(ID)}{\vdash I : (t_4^1, t_4^2, t_6 \to t_7) \to t_4^1, t_4^2, t_6 \to t_7}\,(FUN)$$

Enfin, le système ne contient plus d'équation décomposable par (R_n) ou (R_0), puisqu'il est de la forme :

$$\overline{\mathcal{E}_2} = \left\{ \begin{array}{lcll} t_4^2 \to t_3 & \stackrel{\perp}{=} & t_4^1, & \\ t_6 \to t_7 & \stackrel{\perp}{=} & t_3 & \end{array} \right\}$$

C'est donc la règle (R_f) qui doit être utilisée. Si on l'applique à la deuxième équation, on génère la substitution :

$$\overline{S_2} = \{t_3 \mapsto t_6 \to t_7\}$$

et le système devient :

$$\overline{\mathcal{E}_3} = \left\{ \begin{array}{lcl} t_4^2 \to (t_6 \to t_7) & \stackrel{\perp}{=} & t_4^1 \end{array} \right\}$$

(On notera que cette substitution nous fait effectivement sortir du cadre des types premiers simples et des équations en forme "standard", d'où la nécessité de la conversion de $\mathcal{E}_2$ en $\overline{\mathcal{E}_2}$.)

Enfin, la règle (R_f) génère une dernière substitution :

$$\overline{S_3} = \{t_4^1 \mapsto t_4^2 \to (t_6 \to t_7)\}$$

Il ne reste plus qu'à appliquer $\overline{S_2}$ et $\overline{S_3}$ à Π_2 pour obtenir le résultat de l'algorithme ; on pourra vérifier qu'il s'agit bien d'un arbre de typage valide pour le terme $I(\lambda y(\Delta y)y)$.

Exemple 7.7 *Dans cet exemple, nous allons tenter de réduire le système correspondant à $\Omega = \Delta\Delta$. Pour simplifier, nous ne détaillerons pas la partie squelette du système. L'ensemble initial des contraintes correspondant à ce terme est :*

$$\mathcal{E}_0 = \left\{ \begin{array}{rcll} t_4 \to t_5 & \stackrel{\perp}{=} & t_3 & [t_4], \\ t_1 \to t_2 & \stackrel{\perp}{=} & t_0 & [t_1], \\ (t_3, t_4 \to t_5) \to t_6 & \stackrel{\perp}{=} & t_0, t_1 \to t_2 & [t_3, t_4, t_5] \end{array} \right\}$$

Seule la troisième équation est décomposable par la règle (R_n) avec $n = 2$. Ceci génère la substitution :

$$\begin{aligned} S_0 \;=\; & \{t_0 \mapsto (t_3^1, t_4^1 \to t_5^1), \{t_3^1, t_4^1, t_5^1\}\} :: \{t_1 \mapsto (t_3^2, t_4^2 \to t_5^2), \{t_3^2, t_4^2, t_5^2\}\} \\ & :: \{t_6 \mapsto t_2, \emptyset\} :: D(2, \{t_3, t_4, t_5\}) \end{aligned}$$

En appliquant S_0 aux deux équations restantes de $\mathcal{E}_0$, on trouve :

$$\mathcal{E}_1 = \left\{ \begin{array}{rcll} t_4^1 \to t_5^1 & \doteq & t_3^1 & [t_4^1], \\ t_4^2 \to t_5^2 & \doteq & t_3^2 & [t_4^2], \\ (t_3^2, t_4^2 \to t_5^2) \to t_2 & \doteq & t_3^1, t_4^1 \to t_5^1 & [t_3^2, t_4^2, t_5^2] \end{array} \right\}$$

A un renommage des variables près, il s'agit strictement des mêmes équations que dans $\mathcal{E}_0$. On peut donc affirmer que le système va continuer indéfiniment à se réduire en lui-même (à un renommage près) et donc diverger, ce qui est normal puisque Ω n'est pas normalisable.

7.7 Etat initial

Cette section détaille la construction de l'état initial $Syst_0(M)$ correspondant à un terme M.
Nous commençons par définir les *termes annotés* par :

$$M^{\mathbf{t}} ::= x : t \mid (M^{\mathbf{t}} N^{\mathbf{t}}) : t \mid \lambda x M^{\mathbf{t}} \mid [M^{\mathbf{t}}, N^{\mathbf{t}}]$$

Il s'agit des termes du $\Lambda_{\mathcal{K}}$-calcul auxquels on rajoute une variable de type pour chaque variable et pour chaque noeud-application.

Etant donné un terme $M \in \Lambda_{\mathcal{K}}$, il est possible d'en construire une version annotée qui assigne des variables de type fraîches différentes pour chaque variable et chaque noeud-application de M. Nous noterons $M_0^{\mathbf{t}}(M)$ le résultat de cette opération d'annotation. Il s'agit bien d'un algorithme et non d'une définition mathématique ; en d'autres termes, $M_0^{\mathbf{t}}$ possède une certaine mémoire et il faudrait préciser comment sont choisies les variables fraîches, ce que la notation ne reflète pas. L'utilisation d'un terme obtenu par $M_0^{\mathbf{t}}(M)$ se fait donc toujours "à un renommage près", mais nous rappellerons systématiquement ce point dans la suite lorsque ce sera nécessaire.

Nous allons également avoir besoin des deux fonctions auxiliaires suivantes, l'une pour obtenir le type correspondant à un terme annoté, l'autre pour retrouver toutes les variables de type correspondant à une variable x donnée [21] :

$$\begin{array}{l} Typ(x : t) = t \\ Typ((M^{\mathbf{t}} N^{\mathbf{t}}) : t) = t \\ Typ(\lambda x M^{\mathbf{t}}) = t_1, \ldots, t_n \to Typ(M^{\mathbf{t}}) \quad \text{avec } VarTyp(x, M^{\mathbf{t}}) = \{t_1, \ldots, t_n\} \\ Typ([M^{\mathbf{t}}, N^{\mathbf{t}}]) = Typ(M^{\mathbf{t}}) \end{array}$$

$$\begin{array}{l} VarTyp(x, x : t) = \{t\} \\ VarTyp(x, y : t) = \emptyset \qquad \text{pour } x \neq y \\ VarTyp(x, (M^{\mathbf{t}} N^{\mathbf{t}}) : t) = VarTyp(x, M^{\mathbf{t}}) \cup VarTyp(x, N^{\mathbf{t}}) \\ VarTyp(x, (\lambda x M^{\mathbf{t}}) : \tau) = \emptyset \\ VarTyp(x, (\lambda y M^{\mathbf{t}}) : \tau) = VarTyp(x, M^{\mathbf{t}}) \qquad \text{pour } x \neq y \\ VarTyp(x, [M^{\mathbf{t}}, N^{\mathbf{t}}] : \tau) = VarTyp(x, M^{\mathbf{t}}) \cup VarTyp(x, N^{\mathbf{t}}) \end{array}$$

[21] Pour être précis, il faudrait mentionner que la fonction $VarTyp$ calcule un *multi-ensemble* de variables de types et que par conséquent l'opération d'union $\cup$ utilisée dans sa définition est une union *disjointe*. Cependant, étant donné que nous n'utiliserons $VarTyp$ que sur des termes de la forme $M_0^{\mathbf{t}}(M)$, qui ne contiennent donc que des variables toutes distinctes, cette précision ne change rien.

A partir d'un terme annoté, la fonction suivante définit l'ensemble d'équations initial (exactement une par noeud-application) :

$$\begin{array}{l}\mathcal{E}_0(x:t) = \emptyset \\ \mathcal{E}_0((M^{\mathbf{t}}N^{\mathbf{t}}):t) = \mathcal{E}_0(M^{\mathbf{t}}) \cup \mathcal{E}_0(N^{\mathbf{t}}) \cup \{Typ(N^{\mathbf{t}}) \to t \ \doteq \ Typ(M^{\mathbf{t}}) \ \ [ftv(N^{\mathbf{t}})]\} \\ \mathcal{E}_0(\lambda x M^{\mathbf{t}}) = \mathcal{E}_0(M^{\mathbf{t}}) \\ \mathcal{E}_0([M^{\mathbf{t}}, N^{\mathbf{t}}]) = \mathcal{E}_0(M^{\mathbf{t}}) \cup \mathcal{E}_0(N^{\mathbf{t}})\end{array}$$

Ensuite, la fonction Π_0 suivante définit le squelette de preuve initial correspondant au terme (on note des conversions implicites au passage) :

$$\begin{array}{l}\Pi_0(x:t) = \dfrac{}{x:t \vdash x:t}\,(\text{ID}) \\ \Pi_0((M^{\mathbf{t}}N^{\mathbf{t}}):t) = \dfrac{\Pi_0(M^{\mathbf{t}}) \qquad \Pi_0(N^{\mathbf{t}})}{Env(M^{\mathbf{t}}N^{\mathbf{t}}) \vdash UnTyp(M^{\mathbf{t}}N^{\mathbf{t}}) : t}\,(\text{APPL}) \\ \Pi_0(\lambda x M^{\mathbf{t}}) = \dfrac{\Pi_0(M^{\mathbf{t}})}{Env(\lambda x M^{\mathbf{t}}) \vdash UnTyp(\lambda x M^{\mathbf{t}}) : \overline{Typ(\lambda x M^{\mathbf{t}})}}\,(\text{FUN}) \\ \Pi_0([M^{\mathbf{t}}, N^{\mathbf{t}}]) = \dfrac{\Pi_0(M^{\mathbf{t}}) \qquad \Pi_0(N^{\mathbf{t}})}{Env([M^{\mathbf{t}}, N^{\mathbf{t}}]) \vdash UnTyp([M^{\mathbf{t}}, N^{\mathbf{t}}]) : \overline{Typ([M^{\mathbf{t}}, N^{\mathbf{t}}])}}\,(\text{FORGET})\end{array}$$

Cette fonction fait appel à l'opération $UnTyp(M^{\mathbf{t}})$ qui efface toutes les annotations de type et retourne le terme simple correspondant du $\Lambda_{\mathcal{K}}$-calcul, et à la fonction auxiliaire suivante qui reconstruit un environnement à partir de toutes les liaisons apparaissant dans un terme annoté (il s'agit d'une variante de $VarTyp(x, M^{\mathbf{t}})$ retournant une liste de liaisons pour tous les x à la fois ; on notera aussi que les variables sont implicitement converties en types premiers) :

$$\begin{array}{l}Env(x:t) = x:t \\ Env((M^{\mathbf{t}}N^{\mathbf{t}}):t) = Env(M^{\mathbf{t}}), Env(N^{\mathbf{t}}) \\ Env(\lambda x M^{\mathbf{t}}) = Env(M^{\mathbf{t}}) \setminus x \\ Env([M^{\mathbf{t}}, N^{\mathbf{t}}]) = Env(M^{\mathbf{t}}), Env(N^{\mathbf{t}})\end{array}$$

Enfin, l'état initial du système pour un terme M est défini comme :

$$Syst_0(M) = (\mathcal{E}_0(M^{\mathbf{t}}),\ \Pi_0(M^{\mathbf{t}})) \quad \text{avec } M^{\mathbf{t}} = M_0^{\mathbf{t}}(M)$$

Exemple 7.8 *Soit $M = \Delta(\lambda z y)$ avec $\Delta = \lambda x\ (x x)$. On commence par calculer $M_0^{\mathbf{t}}(M)$:*

$$M^{\mathbf{t}} = M_0^{\mathbf{t}}(M) = ((\lambda x\ (x : t_0\ x : t_1) : t_2)\ (\lambda z\ y : t_3)) : t_4$$

On peut alors calculer l'ensemble initial des contraintes :

$$\mathcal{E}_0(M^{\mathbf{t}}) = \left\{ \begin{array}{rcll} t_1 \to t_2 & \doteq & t_0 & [t_1], \\ (\omega \to t_3) \to t_4 & \doteq & t_0, t_1 \to t_2 & [t_3] \end{array} \right\}$$

ainsi que le squelette de preuve initial :

$$\Pi_0(M^{\mathbf{t}}) = \dfrac{\dfrac{\dfrac{\dfrac{}{x:t_0 \vdash x:t_0}\,(ID) \qquad \dfrac{}{x:t_1 \vdash x:t_1}\,(ID)}{x:t_0, x:t_1 \vdash x\ x : t_2}\,(APPL)}{\vdash \Delta : t_0, t_1 \to t_2}\,(FUN) \qquad \dfrac{\dfrac{}{y:t_3 \vdash y:t_3}\,(ID)}{y:t_3 \vdash \lambda z\ y : \omega \to t_3}\,(FUN)}{y:t_3 \vdash \Delta\ (\lambda z\ y) : t_4}\,(APPL)$$

La suite de l'inférence pour ce terme a déjà été donnée à l'Exemple 7.5.

7.8 Polarité

Dans cette partie, nous justifions (informellement) le choix des définitions pour les substitutions et nous tentons de donner quelques intuitions sur la manière dont l'algorithme fonctionne. La technique utilisée va consister à donner une *polarité* (positive ou négative) aux types manipulés par le système. Cette méthode, déjà utilisée par d'autres auteurs dans la littérature [Jim00], a été formalisée explicitement et utilisée avec succès dans un cadre similaire [KW04, Car02]. Pour notre présente étude, il s'est avéré que cela n'était pas nécessaire ; néanmoins, la mentionner contribue grandement à la clarté de l'algorithme. C'est pourquoi nous la présentons ici.

Polarité des types Nous étiquetons les variables de type avec une polarité, notée t^+ si la variable est positive, et t^- si elle est négative.

La règle pour l'attribution des polarités aux types premiers simples est la suivante : la polarité est inversée à gauche de la flèche et reste la même à droite de la flèche. Ainsi, un type premier simple de polarité positive, noté τ^+, peut être soit une variable positive, soit un type flèche composé de variables-arguments toutes négatives et d'un type-résultat positif :

$$\tau^+ ::= t^+ \mid t_1^-, \ldots, t_n^- \to \tau^+$$

De façon symétrique, un type de polarité négative, noté τ^-, est soit une variable négative, soit un type flèche composé de variables-arguments positives et d'un type négatif :

$$\tau^- ::= t^- \mid t_1^+, \ldots, t_n^+ \to \tau^-$$

De façon similaire, on peut attribuer des polarités $\overline{\tau^+}$ et $\overline{\tau^-}$ aux types premiers.

Polarité des équations Nous donnons également des polarités inverses aux deux membres d'une équation. Par convention, nous choisissons que le type droit est de polarité positive et que celui de gauche est de polarité négative (le territoire reste lui sans polarité). Etant donné la forme particulière du membre gauche d'une équation, nous avons donc les polarités suivantes :

$$eq ::= (\tau^+ \to t^- \ \doteq\ \sigma^+ \ \ [T])$$

où τ^+ et σ^+ (les types correspondant à la fonction et à l'argument de l'application représentée par l'équation) sont positifs, alors que le noeud de l'équation t^- est négatif.

De façon similaire, la polarité pour une équation irréductible est la suivante :

$$\overline{eq} ::= (\overline{\tau^-} \ \doteq\ t^+)$$

Etat initial Montrons que la construction de l'état initial respecte bien la polarité imposée aux équations. Pour cela, considérons les fonctions Typ et $VarTyp$ modifiées suivantes :

$$\begin{array}{l} Typ(x : t) = t^+ \\ Typ((M^{\mathbf{t}} N^{\mathbf{t}}) : t) = t^+ \\ Typ(\lambda x M^{\mathbf{t}}) = t_1^-, \ldots, t_n^- \to Typ(M^{\mathbf{t}}) \quad \text{avec } VarTyp(x, M^{\mathbf{t}}) = \{t_1^-, \ldots, t_n^-\} \\ Typ([M^{\mathbf{t}}, N^{\mathbf{t}}]) = Typ(M^{\mathbf{t}}) \end{array}$$

$$
\begin{array}{l}
VarTyp(x, x:t) = \{t^-\} \\
VarTyp(x, y:t) = \emptyset \qquad \text{pour } x \neq y \\
VarTyp(x, (M^{\mathbf{t}} N^{\mathbf{t}}) : t) = VarTyp(x, M^{\mathbf{t}}) \cup VarTyp(x, N^{\mathbf{t}}) \\
VarTyp(x, (\lambda x M^{\mathbf{t}}) : \tau) = \emptyset \\
VarTyp(x, (\lambda y M^{\mathbf{t}}) : \tau) = VarTyp(x, M^{\mathbf{t}}) \qquad \text{pour } x \neq y \\
VarTyp(x, [M^{\mathbf{t}}, N^{\mathbf{t}}] : \tau) = VarTyp(x, M^{\mathbf{t}}) \cup VarTyp(x, N^{\mathbf{t}})
\end{array}
$$

La fonction $VarTyp$ retourne un (multi)-ensemble de variables de type de polarité négative (ce qui permet de l'utiliser comme argument dans la définition de $Typ(\lambda x M^{\mathbf{t}})$). A l'inverse, la fonction Typ retourne un type de polarité positive τ^+.

Enfin, la construction de l'ensemble d'équations initial par $\mathcal{E}_0$ est donné par :

$$
\begin{array}{l}
\mathcal{E}_0(x:t) = \emptyset \\
\mathcal{E}_0((M^{\mathbf{t}} N^{\mathbf{t}}) : t) = \mathcal{E}_0(M^{\mathbf{t}}) \cup \mathcal{E}_0(N^{\mathbf{t}}) \cup \{Typ(N^{\mathbf{t}}) \to t^- \ \doteq \ Typ(M^{\mathbf{t}}) \ \ [ftv(N^{\mathbf{t}})]\} \\
\mathcal{E}_0(\lambda x M^{\mathbf{t}}) = \mathcal{E}_0(M^{\mathbf{t}}) \\
\mathcal{E}_0([M^{\mathbf{t}}, N^{\mathbf{t}}]) = \mathcal{E}_0(M^{\mathbf{t}}) \cup \mathcal{E}_0(N^{\mathbf{t}})
\end{array}
$$

Ceci est bien correct, car $Typ(M^{\mathbf{t}})$ et $Typ(N^{\mathbf{t}})$ ont une polarité positive et l'on donne une polarité négative au noeud de l'équation. L'état initial $Syst_0(M)$ est donc correctement polarisé.

Propriétés d'invariance du système Le système initial possède une propriété supplémentaire concernant le nombre d'occurrences des variables positives et négatives :

Propriété 7.9

- *chaque variable de type correspondant à un noeud-application possède exactement une occurrence négative et au plus une occurrence positive dans les équations initiales*
- *chaque variable de type correspondant à une variable x possède au plus une occurrence négative et au plus une occurrence positive dans les équations initiales*

Preuve: (informelle)

- Soit $M^{\mathbf{t}} = (N_1^{\mathbf{t}} N_2^{\mathbf{t}}) : t$ une application du terme analysé. La variable t possède une occurrence négative t^- dans $\mathcal{E}_0(M^{\mathbf{t}})$. Elle ne peut en avoir d'autres car il n'est pas possible d'obtenir t^- autrement. L'unique éventuelle occurrence positive t^+ est obtenue en remontant dans le terme à partir de $M^{\mathbf{t}}$ jusqu'à l'endroit où elle apparaît (éventuellement sous des λs) comme fonction ou comme argument d'une autre application, par exemple $P^{\mathbf{t}} = ((\lambda x M^{\mathbf{t}}) \ldots) : t'$. Dans l'équation correspondant à $P^{\mathbf{t}}$, la variable positive t^+ apparaît alors comme le type le plus à droite des flèches dans $Typ(\lambda x M^{\mathbf{t}})$. Il se peut que t^+ n'apparaisse nulle part lorsque $M^{\mathbf{t}}$ se trouve à la racine du terme analysé (éventuellement sous des λs) ou dans le membre droit d'une construction $[\_, \_]$.
- Soit $x : t$ une variable du terme analysé. L'éventuelle occurrence positive de t se traite comme ci-dessus. Son éventuelle occurrence négative s'obtient lorsque x est lié par un λx. La variable t^- apparaît alors à gauche d'une flèche lorsqu'on calcule $Typ(\lambda x \ldots)$ (à condition à nouveau que ce λx ne soit ni à la racine ni dans le membre droit d'un $[\_, \_]$).

□

De plus, cette propriété est invariante par réduction du système. On pourrait le démontrer directement, mais ce n'est pas nécessaire car, comme on le verra au chapitre suivant, les ensembles d'équations intermédiaires correspondent toujours à l'ensemble des équations initiales pour un certain terme.

Evolution du système Considérons maintenant la règle (R_n) en explicitant les polarités de l'équation à décomposer :

$$\begin{array}{c}(\{\tau^+ \to t^- \ \stackrel{\perp}{=}\ t_1^-, \ldots, t_n^- \to \sigma^+ \ \ [T]\} \cup \mathcal{E},\ \Pi) \ \longrightarrow\ (S(\mathcal{E}),\ \overline{S}(\Pi)) \\ \text{avec } S = \{t_i \mapsto \langle\tau\rangle^i, \langle T\rangle^i\}_{1\leq i\leq n} :: \{t \mapsto \sigma, \emptyset\} :: D(n,T)\end{array} \quad (R_n)$$

Vue la propriété ci-dessus, on sait que τ^+, σ^+ et $\mathcal{E}$ ne peuvent plus contenir d'occurrences négatives des variables t, $t_1, \ldots, t_n$, et qu'ils contiennent au plus une occurrence positive pour chacune de ces variables. Par conséquent, les substitutions sur ces variables dans S ne peuvent porter que ces (uniques) occurrences positives éventuelles, et on est en droit de récrire S sous la forme :

$$S = \{t_i^+ \mapsto \langle\tau^+\rangle^i, \langle T\rangle^i\}_{1\leq i\leq n} :: \{t^+ \mapsto \sigma^+, \emptyset\} :: D(n,T)$$

ce qui se lit intuitivement sous la forme "les variables positives t^+ et t_i^+ sont remplacées par les types positifs σ^+ et $\langle\tau^+\rangle^i$ dans $\mathcal{E}$". [22]

Des arguments similaires nous conduisent à récrire la règle (R_0) sous la forme :

$$\begin{array}{c}(\{\tau^+ \to t^- \ \stackrel{\perp}{=}\ \omega \to \sigma^+ \ \ [T]\} \cup \mathcal{E},\ \Pi) \ \longrightarrow\ (S(\mathcal{E}),\ \overline{S}(\Pi)) \\ \text{avec } S = \{t^+ \mapsto \sigma^+, \emptyset\}\end{array} \quad (R_0)$$

Par conséquent, toutes les substitutions de $\mathcal{S}_s$ générées par l'algorithme sont en fait de la forme :

$$S ::= \{t^+ \mapsto \tau^+, T\} \mid D(n,T)$$

Voyons ce que cela implique sur la définition de l'application d'une substitution en explicitant les polarités. Pour $S = \{t^+ \mapsto \tau^+, T\}$:

$$\begin{cases} S(t^+) = \tau^+ \\ S(t'^+) = t'^+ \qquad \text{si } t' \neq t \\ S(t_1^-, \ldots, t_n^- \to \sigma^+) = t_1^-, \ldots, t_n^- \to S(\sigma^+) \\ S(\tau'^+ \to t'^- \ \stackrel{\perp}{=}\ \sigma'^+ \ \ [T']) = \{S(\tau'^+) \to t'^- \ \stackrel{\perp}{=}\ S(\sigma'^+) \ \ [S(T')]\} \end{cases}$$

Nous constatons que t ne saurait apparaître à gauche d'une flèche ou comme noeud d'une équation puisque ces positions sont négatives, alors que nous sommes sûrs que la substitution doit s'effectuer sur des variables positives uniquement. Nous pourrions donc utiliser une définition classique complète pour l'application d'une substitution au lieu de notre application partielle, puisque nous avons la garantie que le résultat pour un type premier simple est un type premier simple, et que le résultat pour une équation est une équation de forme standard (à condition bien sûr de ne considérer que les substitutions générées par l'algorithme).

Cas de la règle finale Les polarités permettent également de justifier pourquoi un système d'équations irréductibles garde la bonne forme. Pour la règle (R_f), la situation est inversée :

$$(\{\overline{\tau^-} \ \stackrel{\perp}{=}\ t^+\} \cup \overline{\mathcal{E}},\ \Pi) \ \longrightarrow_f\ (\overline{S}(\overline{\mathcal{E}}),\ \overline{S}(\Pi)) \qquad \text{avec } \overline{S} = \{t^- \mapsto \overline{\tau^-}\} \quad (R_f)$$

[22] Un raisonnement similaire sur la substitution correspondante $\overline{S}$ appliquée à Π n'est pas possible, car les squelettes de preuve ne vérifient pas forcément les propriétés d'invariance.

En effet, c'est l'occurrence positive de t qui est supprimée et la substitution porte donc sur l'éventuelle unique occurrence négative restante. Si l'on considère maintenant la définition de l'application de cette substitution à une équation irréductible :

$$\overline{S}(\overline{\tau'^{-}} \doteq t'^{+}) = (\overline{S}(\overline{\tau'^{-}}) \doteq t'^{+})$$

on constate qu'elle ne saurait s'appliquer au membre droit d'une équation puisqu'il s'agit toujours d'une variable positive alors que $\overline{S}$ porte sur une variable négative.

7.9 Rang et algorithme d'inférence à rang fini

Dans cette dernière partie, nous donnons une définition du rang et montrons comment elle peut être utilisée pour construire un algorithme d'inférence pour un rang maximal donné. Dans les chapitres suivants, nous verrons que cet algorithme termine toujours et que la typabilité d'un terme pour un rang fini est donc décidable.

Rang Nous définissons le rang d'un type premier comme suit (il s'agit de la définition donnée dans [KMTW99]) :

$$\begin{array}{l} inc(0) = 0 \\ inc(n) = n+1 \quad \text{pour } n > 0 \\ \\ rank(t) = 0 \\ rank(\overline{\tau} \to \overline{\sigma}) = \max(inc(rank(\overline{\tau})), rank(\overline{\sigma})) \\ rank(\overline{\tau_1}, \ldots, \overline{\tau_n} \to \overline{\sigma}) = \max(inc(\max(1, rank(\overline{\tau_1}), \ldots, rank(\overline{\tau_n}))), rank(\overline{\sigma})) \quad \text{pour } n \neq 1 \end{array}$$

On peut remarquer que les types de rang 0 correspondent exactement aux types usuels sans intersection (cependant, du fait de l'absence de contraction, le système de types ne généralise pas celui des types simples). Il n'existe aucun type de rang 1, et si un type a un rang $r \geq 2$, il faut traverser $r-1$ flèches pour atteindre la conjonction non triviale la plus profonde (une conjonction est non triviale lorsque $n = 0$ ou $n \geq 2$).

Le rang d'un squelette de preuve Π est défini comme le rang maximal des types apparaissant dans Π.

Algorithme d'inférence à rang fini On construit un algorithme d'inférence à rang fini en se fixant un rang maximal autorisé r et en vérifiant à chaque étape que pour tout état intermédiaire $(\mathcal{E},\ \Pi)$ du système, on a la condition $rank(\Pi) \leq r$ vérifiée [23] . Si ce n'est pas le cas, l'algorithme s'arrête et le terme est déclaré non typable au rang r.

Exemple 7.10 *Si l'on reprend l'Exemple 7.7 de $\Omega = \Delta\Delta$ en explicitant cette fois-ci le squelette de preuve, on doit constater une augmentation de son rang* [24] *. Le squelette de preuve initial est le*

[23] Il n'est pas nécessaire de considérer aussi $rank(\mathcal{E})$ car tous les types apparaissant dans $\mathcal{E}$ apparaissent aussi dans Π (dans une utilisation "normale" du système, i.e. pour un état obtenu à partir de $Syst_0(M)$ pour un terme M).

[24] On pourra remarquer au passage qu'il ne suffit pas de considérer le rang de $\mathcal{E}$ au lieu de celui de Π. En effet, sur cet exemple, on constate que le rang des équations peut rester constant tout au long des réductions, alors même que le terme diverge...

suivant :

$$\Pi_0 = \dfrac{\dfrac{\dfrac{}{x : t_0 \vdash x : t_0}\,(ID) \qquad \dfrac{}{x : t_1 \vdash x : t_1}\,(ID)}{x : t_0, x : t_1 \vdash x\ x : t_2}\,(APPL)}{\vdash \Delta : t_0, t_1 \to t_2}\,(FUN) \qquad \Pi'_0}{\vdash \Delta\ \Delta : t_6}\,(APPL)$$

$$\textit{avec} \quad \Pi'_0 = \dfrac{\dfrac{\dfrac{}{x : t_3 \vdash x : t_3}\,(ID) \qquad \dfrac{}{x : t_4 \vdash x : t_4}\,(ID)}{x : t_3, x : t_4 \vdash x\ x : t_5}\,(APPL)}{\vdash \Delta : t_3, t_4 \to t_5}\,(FUN)$$

Le rang initial $rank(\Pi_0)$ *vaut donc* 2*. On rappelle que la substitution effectuée en premier est :*

$$\begin{aligned} S_0 \quad = \quad & \{t_0 \mapsto (t_3^1, t_4^1 \to t_5^1), \{t_3^1, t_4^1, t_5^1\}\} :: \{t_1 \mapsto (t_3^2, t_4^2 \to t_5^2), \{t_3^2, t_4^2, t_5^2\}\} \\ & :: \{t_6 \mapsto t_2, \emptyset\} :: D(2, \{t_3, t_4, t_5\}) \end{aligned}$$

Notons $\tau_1 = t_3^1, t_4^1 \to t_5^1$ *et* $\tau_2 = t_3^2, t_4^2 \to t_5^2$*. Si on applique* S *à* Π_0*, on obtient le squelette :*

$$\Pi_1 = \dfrac{\dfrac{\dfrac{}{x : \overline{\tau_1} \vdash x : \overline{\tau_1}}\,(ID) \qquad \dfrac{}{x : \overline{\tau_2} \vdash x : \overline{\tau_2}}\,(ID)}{x : \overline{\tau_1}, x : \overline{\tau_2} \vdash x\ x : t_2}\,(APPL)}{\vdash \Delta : \overline{\tau_1}, \overline{\tau_2} \to t_2}\,(FUN) \qquad \langle \Pi'_0 \rangle^1 \qquad \langle \Pi'_0 \rangle^2}{\vdash \Delta\ \Delta : t_2}\,(APPL)$$

dont le rang est 3*. Si l'on devait pousser la réduction une étape plus loin, la substitution générée serait :*

$$\begin{aligned} S_1 \quad = \quad & \{t_3^1 \mapsto \langle \tau_2 \rangle^1, \{t_3^{2.1}, t_4^{2.1}, t_5^{2.1}\}\} :: \{t_4^1 \mapsto \langle \tau_2 \rangle^2, \{t_3^{2.2}, t_4^{2.2}, t_5^{2.2}\}\} \\ & :: \{t_2 \mapsto t_5^1, \emptyset\} :: D(2, \{t_3^2, t_4^2, t_5^2\}) \end{aligned}$$

et le nouveau squelette $\Pi_2 = \overline{S_1}(\Pi_1)$ *est alors de rang* 4 *car le type pour le premier* Δ *:*

$$\overline{S_1}(\overline{\tau_1}, \overline{\tau_2} \to t_2) = (\langle \overline{\tau_2} \rangle^1, \langle \overline{\tau_2} \rangle^2 \to t_5^1), \langle \overline{\tau_2} \rangle^1, \langle \overline{\tau_2} \rangle^2 \to t_5^1$$

est de rang 4*.*

En continuant ainsi, on peut montrer que le rang du squelette de preuve est incrémenté à chaque étape. Par conséquent, pour n'importe quel rang maximal autorisé r*, l'algorithme à rang fini s'arrête en concluant à la non-typabilité de* Ω *au rang* r*.*

Chapitre 8

Résultats et preuves

Dans ce chapitre, nous énonçons et démontrons les propriétés vérifiées par l'algorithme défini au chapitre précédent. Les preuves se découpent en deux parties : la première s'intéresse uniquement à la partie contraintes des états du système et donne des résultats sur l'*évolution* de celui-ci ; la deuxième se focalise sur la partie squelette de preuve et énonce donc les propriétés relatives au *résultat* fourni par l'algorithme.

8.1 Résultats relatifs à l'évolution

8.1.1 Renommage

Pour des raisons essentiellement techniques, nous commençons par formaliser la notion d'égalité "à un renommage près" :

Définition 8.1 *Un* renommage *est une substitution simple* $\{t \mapsto t'\}$ *d'une variable de type en une autre. Il ne doit être appliqué qu'à des structures* X *telles que* $t' \notin ftv(X)$ *afin d'éviter les captures de noms.*

On dénotera par θ *une séquence (éventuellement vide) de renommages, et on écrira* $X = \theta(Y)$ *lorsqu'une telle opération est valide. On notera* $X \equiv Y$ *et on dira que "X et Y sont égaux à un renommage près" s'il existe* θ *tel que* $X = \theta(Y)$*.*

Lemme 8.2 $\equiv$ *est une relation d'équivalence.*

Preuve:

- Clairement, $X \equiv X$ puisqu'on peut prendre une séquence vide de renommages.
- $\equiv$ est transitive : si $X = \theta(Y)$ et $Y = \theta'(Z)$, alors $X = (\theta \circ \theta')(Z)$.
- Supposons que $X = \{t \mapsto t'\}(Y)$ (avec $t \neq t'$). Clairement, $t \notin ftv(\{t \mapsto t'\}(Y))$, et par conséquent $t \notin ftv(X)$. Il est donc valide d'appliquer ce renommage aux deux membres de l'équation : $\{t' \mapsto t\}(X) = \{t' \mapsto t\}\{t \mapsto t'\}(Y)$. Il n'est alors pas difficile de vérifier que (puisque $t' \notin ftv(Y)$), $\{t' \mapsto t\}\{t \mapsto t'\}(Y) = Y$. La symétrie de $\equiv$ s'ensuit.

□

8.1.2 Lemmes préliminaires

Cette section contient quelques lemmes simples mais utiles pour les preuves à venir.

Lemme 8.3 $ftv(\mathcal{E}_0(M^{\mathbf{t}})) \subseteq ftv(M^{\mathbf{t}})$

Preuve: Par induction sur $M^{\mathbf{t}}$, en utilisant la propriété auxiliaire que $ftv(Typ(M^{\mathbf{t}})) \subseteq ftv(M^{\mathbf{t}})$. □

Le lemme suivant est simple mais essentiel ; sa première partie sera utilisée de nombreuses fois par la suite. Il formalise simplement le fait que $M_0^{\mathbf{t}}$ est un *algorithme* qui annote le terme qu'on lui donne par des variables fraîches ; deux exécutions distinctes de cet algorithme donneront donc des termes annotés sans aucune variable en commun (même si on utilise le même terme de départ).

Lemme 8.4
- $ftv(M_0^{\mathbf{t}}(M)) \cap ftv(M_0^{\mathbf{t}}(N)) = \emptyset$
- *Si* $M^{\mathbf{t}}$ *et* $N^{\mathbf{t}}$ *sont deux instances différentes de* $M_0^{\mathbf{t}}(M)$ *(i.e. l'algorithme* $M_0^{\mathbf{t}}$ *exécuté deux fois pour un même terme de départ* M*), alors* $M^{\mathbf{t}} \equiv N^{\mathbf{t}}$.

Preuve:
- Toutes les variables de type introduites dans $M_0^{\mathbf{t}}(M)$ sont toujours fraîches, par conséquent elles seront différentes pour deux termes M et N (même si ces deux termes sont identiques).
- Il suffit de construire le renommage adéquat en appariant les variables de type se trouvant à la même place dans $M^{\mathbf{t}}$ et $N^{\mathbf{t}}$.

□

Le lemme suivant énonce quelques propriétés simples sur l'application d'une substitution à des structures qui n'ont pas de variables dans le domaine de cette substitution.

Lemme 8.5
- *Si* $dom(S) \cap ftv(\tau) = \emptyset$, *alors* $S(\tau) = \tau$.
- *Si* $dom(S) \cap ftv(\mathcal{E}) = \emptyset$, *alors* $S(\mathcal{E}) = \mathcal{E}$.
- *Si* $dom(\overline{S}) \cap ftv(\Pi) = \emptyset$, *alors* $\overline{S}(\Pi) = \Pi$.

Preuve: Par induction sur τ, $\mathcal{E}$ et Π respectivement. □

Les deux lemmes suivants énoncent des propriétés relatives aux mécanismes d'indexation et de duplication.

Lemme 8.6
- $ftv(\langle X \rangle^i) = \langle ftv(X) \rangle^i$
- $Typ(\langle M^{\mathbf{t}} \rangle^i) = \langle Typ(M^{\mathbf{t}}) \rangle^i$
- $ftv(dupl_n(X)) = dupl_n(ftv(X))$

Preuve:
- Puisque $ftv(-)$ et $\langle - \rangle^-$ sont tous deux des homomorphismes sur des structures construites à partir de $\mathcal{TV}ar$, il suffit de remarquer que : $ftv(\langle t \rangle^i) = \{\langle t \rangle^i\} = \langle \{t\} \rangle^i$.
- Trivial.
- Que $dupl_n(X)$ désigne une séquence ou un ensemble ne change rien, et on a :

$$ftv(dupl_n(X)) = \bigcup_{1 \leq i \leq n} ftv(\langle X \rangle^i) = \bigcup_{1 \leq i \leq n} \langle ftv(X) \rangle^i$$

□

Lemme 8.7
- $\langle \mathcal{E}_0(M^{\mathbf{t}}) \rangle^i = \mathcal{E}_0(\langle M^{\mathbf{t}} \rangle^i)$

– $dupl_n(\mathcal{E}_0(M^{\mathbf{t}})) = \bigcup_{1\leq i\leq n} \mathcal{E}_0(\langle M^{\mathbf{t}}\rangle^i)$

Preuve:

– Par induction sur $M^{\mathbf{t}}$. Nous ne détaillons que le cas de l'application :

$$\begin{aligned}
\langle \mathcal{E}_0((M^{\mathbf{t}}N^{\mathbf{t}}) : t)\rangle^i &= \langle \mathcal{E}_0(M^{\mathbf{t}}) \cup \mathcal{E}_0(N^{\mathbf{t}}) \cup \{Typ(N^{\mathbf{t}}) \to t \doteq Typ(M^{\mathbf{t}}) \ \ [ftv(N^{\mathbf{t}})]\}\rangle^i \\
&= \langle \mathcal{E}_0(M^{\mathbf{t}})\rangle^i \cup \langle \mathcal{E}_0(N^{\mathbf{t}})\rangle^i \\
&\qquad \cup\{\langle Typ(N^{\mathbf{t}})\rangle^i \to \langle t\rangle^i \doteq \langle Typ(M^{\mathbf{t}})\rangle^i \ \ [\langle ftv(N^{\mathbf{t}})\rangle^i]\} \\
&= \mathcal{E}_0(\langle M^{\mathbf{t}}\rangle^i) \cup \mathcal{E}_0(\langle N^{\mathbf{t}}\rangle^i) \\
&\qquad \cup\{Typ(\langle N^{\mathbf{t}}\rangle^i) \to \langle t\rangle^i \doteq Typ(\langle M^{\mathbf{t}}\rangle^i) \ \ [ftv(\langle N^{\mathbf{t}}\rangle^i)]\} \\
&\qquad \text{en utilisant l'induction et le Lemme 8.6} \\
&= \mathcal{E}_0((\langle M^{\mathbf{t}}\rangle^i \ \langle N^{\mathbf{t}}\rangle^i) : \langle t\rangle^i) \\
&= \mathcal{E}_0(\langle (M^{\mathbf{t}}N^{\mathbf{t}}) : t\rangle^i)
\end{aligned}$$

– D'après la définition de $dupl_n(X)$ pour un ensemble X, on a :

$$dupl_n(\mathcal{E}_0(M^{\mathbf{t}})) = \bigcup_{1\leq i\leq n} \langle \mathcal{E}_0(M^{\mathbf{t}})\rangle^i = \bigcup_{1\leq i\leq n} \mathcal{E}_0(\langle M^{\mathbf{t}}\rangle^i)$$

□

Lemme 8.8 *Soit* $(\{eq\} \cup \mathcal{E}, \Pi) \longrightarrow (S(\mathcal{E}), \overline{S}(\Pi))$ *une instance des règles* (R_n) *où* (R_0)*. Alors,* $dom(S) \subseteq ftv(eq)$*.*

Preuve: Trivial. □

8.1.3 Lemmes principaux

Afin de pouvoir prouver des lemmes génériques, la notion de *contexte* est indispensable :

Définition 8.9 *Un* contexte annoté *possède la même structure qu'un terme annoté, mis à part qu'il contient un trou* $\square$ *:*

$$C^{\mathbf{t}} ::= \square \mid (C^{\mathbf{t}}N^{\mathbf{t}}) : t \mid (M^{\mathbf{t}}C^{\mathbf{t}}) : t \mid \lambda x C^{\mathbf{t}} \mid [C^{\mathbf{t}}, N^{\mathbf{t}}] \mid [M^{\mathbf{t}}, C^{\mathbf{t}}]$$

On écrira $C^{\mathbf{t}}[M^{\mathbf{t}}]$ *pour le terme annoté* $C^{\mathbf{t}}$ *dans lequel le trou a été remplacé par* $M^{\mathbf{t}}$*. De plus, on désignera par* $bv(C^{\mathbf{t}})$ *l'ensemble des variables capturées par* $C^{\mathbf{t}}$*, c'est-à-dire toutes les variables* x *telles que le trou de* $C^{\mathbf{t}}$ *apparaît sous un* λx*.*

Le lemme suivant relie l'application d'une certaine substitution avec le remplacement d'une variable $x : t$ par un terme $M^{\mathbf{t}}$ dans un contexte annoté.

Lemme 8.10 *Soient* $C^{\mathbf{t}}$ *un contexte annoté,* $M^{\mathbf{t}}$ *un terme annoté tel que* $fv(M^{\mathbf{t}}) \cap bv(C^{\mathbf{t}}) = \emptyset$*,* t *une variable de type fraîche (qui n'apparaisse pas dans les deux précédentes expressions), et* x *une variable telle que* $x \notin bv(C^{\mathbf{t}})$*. Si on dénote par* S *la substitution* $\{t \mapsto Typ(M^{\mathbf{t}}), ftv(M^{\mathbf{t}})\}$*, alors :*

– $\mathcal{E}_0(C^{\mathbf{t}}[M^{\mathbf{t}}]) = S(\mathcal{E}_0(C^{\mathbf{t}}[x : t])) \cup \mathcal{E}_0(M^{\mathbf{t}})$
– $Typ(C^{\mathbf{t}}[M^{\mathbf{t}}]) = S(Typ(C^{\mathbf{t}}[x : t]))$
– $ftv(C^{\mathbf{t}}[M^{\mathbf{t}}]) = S(ftv(C^{\mathbf{t}}[x : t]))$

Preuve: Par induction sur la structure de $C^{\mathbf{t}}$.

Cas $C^{\mathbf{t}} = \Box$ **:** Dans ce cas, $S(\mathcal{E}_0(C^{\mathbf{t}}[x:t])) = S(\mathcal{E}_0(x:t)) = S(\emptyset) = \emptyset$. Et, $\mathcal{E}_0(C^{\mathbf{t}}[M^{\mathbf{t}}]) = \mathcal{E}_0(M^{\mathbf{t}})$. De plus, $S(Typ(x:t)) = S(t) = Typ(M^{\mathbf{t}})$ et $S(ftv(x:t)) = S(\{t\}) = ftv(M^{\mathbf{t}})$.

Cas $C^{\mathbf{t}} = (N^{\mathbf{t}} C_1^{\mathbf{t}}) : t'$ **:** On a : $C^{\mathbf{t}}[M^{\mathbf{t}}] = (N^{\mathbf{t}}\ C_1^{\mathbf{t}}[M^{\mathbf{t}}]) : t'$ et $C^{\mathbf{t}}[x:t] = (N^{\mathbf{t}}\ C_1^{\mathbf{t}}[x:t]) : t'$.

$$\begin{aligned}\mathcal{E}_0(C^{\mathbf{t}}[M^{\mathbf{t}}]) &= \mathcal{E}_0(N^{\mathbf{t}}) \cup \mathcal{E}_0(C_1^{\mathbf{t}}[M^{\mathbf{t}}]) \cup \{Typ(C_1^{\mathbf{t}}[M^{\mathbf{t}}]) \to t' \doteq Typ(N^{\mathbf{t}})\ \ [ftv(C_1^{\mathbf{t}}[M^{\mathbf{t}}])]\} \\ &= \mathcal{E}_0(N^{\mathbf{t}}) \cup S(\mathcal{E}_0(C_1^{\mathbf{t}}[x:t])) \cup \mathcal{E}_0(M^{\mathbf{t}}) \\ &\qquad \cup \{S(Typ(C_1^{\mathbf{t}}[x:t])) \to t' \doteq Typ(N^{\mathbf{t}})\ \ [S(ftv(C_1^{\mathbf{t}}[x:t]))]\}\end{aligned}$$

par hypothèse d'induction. Puisque t a été choisie fraîche, $t \neq t'$ et $t \notin ftv(N^{\mathbf{t}})$. Par conséquent, l'expression ci-dessus est égale à :

$$\begin{aligned}& S(\mathcal{E}_0(N^{\mathbf{t}})) \cup S(\mathcal{E}_0(C_1^{\mathbf{t}}[x:t])) \cup \mathcal{E}_0(M^{\mathbf{t}}) \\ &\qquad \cup\ S(\{Typ(C_1^{\mathbf{t}}[x:t]) \to t' \doteq Typ(N^{\mathbf{t}})\ \ [ftv(C_1^{\mathbf{t}}[x:t])]\}) \\ =\ & S(\mathcal{E}_0((N^{\mathbf{t}}\ C_1^{\mathbf{t}}[x:t]) : t')) \cup \mathcal{E}_0(M^{\mathbf{t}})\end{aligned}$$

En ce qui concerne les autres conditions, puisque $t \neq t'$, on a :

$$Typ(C^{\mathbf{t}}[M^{\mathbf{t}}]) = t' = S(t') = S(Typ(C^{\mathbf{t}}[x:t]))$$

Et :

$$\begin{aligned}ftv(C^{\mathbf{t}}[M^{\mathbf{t}}]) &= ftv(N^{\mathbf{t}}) \cup ftv(C_1^{\mathbf{t}}[M^{\mathbf{t}}]) \cup \{t'\} \\ &= ftv(N^{\mathbf{t}}) \cup S(ftv(C_1^{\mathbf{t}}[x:t])) \cup \{t'\} \quad \text{par hypothèse d'induction} \\ &= S(ftv(N^{\mathbf{t}})) \cup S(ftv(C_1^{\mathbf{t}}[x:t])) \cup S(\{t'\}) \quad \text{comme précédemment} \\ &= S(ftv((N^{\mathbf{t}}\ C_1^{\mathbf{t}}[x:t]) : t')) \\ &= S(ftv(C^{\mathbf{t}}[x:t]))\end{aligned}$$

Cas $C^{\mathbf{t}} = \lambda y\ C_1^{\mathbf{t}}$ **:** On a : $C^{\mathbf{t}}[M^{\mathbf{t}}] = \lambda y\ (C_1^{\mathbf{t}}[M^{\mathbf{t}}])$ et $C^{\mathbf{t}}[x:t] = \lambda y\ (C_1^{\mathbf{t}}[x:t])$.

$$\begin{aligned}\mathcal{E}_0(C^{\mathbf{t}}[M^{\mathbf{t}}]) &= \mathcal{E}_0(C_1^{\mathbf{t}}[M^{\mathbf{t}}]) \\ &= S(\mathcal{E}_0(C_1^{\mathbf{t}}[x:t])) \cup \mathcal{E}_0(M^{\mathbf{t}}) \quad \text{par hypothèse d'induction} \\ &= S(\mathcal{E}_0(C^{\mathbf{t}}[x:t])) \cup \mathcal{E}_0(M^{\mathbf{t}})\end{aligned}$$

Concernant le type : $Typ(C^{\mathbf{t}}[M^{\mathbf{t}}]) = VarTyp(y, C_1^{\mathbf{t}}[M^{\mathbf{t}}]) \to Typ(C_1^{\mathbf{t}}[M^{\mathbf{t}}])$. Puisque $M^{\mathbf{t}}$ ne peut pas avoir de variable libre dans $bv(C^{\mathbf{t}})$, on a également : $x \neq y$. Une conséquence immédiate de ces deux derniers faits est que : $VarTyp(y, C_1^{\mathbf{t}}[M^{\mathbf{t}}]) = VarTyp(y, C_1^{\mathbf{t}}[x:t])$ (i.e. la variable y ne peut apparaître que dans $C_1^{\mathbf{t}}$). Par hypothèse d'induction, $Typ(C_1^{\mathbf{t}}[M^{\mathbf{t}}]) = S(Typ(C_1^{\mathbf{t}}[x:t]))$. Et finalement :

$$Typ(C^{\mathbf{t}}[M^{\mathbf{t}}]) = VarTyp(y, C_1^{\mathbf{t}}[x:t]) \to S(Typ(C_1^{\mathbf{t}}[x:t])) = S(Typ(C^{\mathbf{t}}[x:t]))$$

Pour l'ensemble des variables de type libres :

$$\begin{aligned}ftv(C^{\mathbf{t}}[M^{\mathbf{t}}]) &= ftv(C_1^{\mathbf{t}}[M^{\mathbf{t}}]) \\ &= S(ftv(C_1^{\mathbf{t}}[x:t])) \quad \text{par hypothèse d'induction} \\ &= S(ftv(C^{\mathbf{t}}[x:t]))\end{aligned}$$

Autres cas : Similaires.

□

Le prochain lemme est le lemme principal et également celui qui demande le plus de travail : il relie les réductions dans le $\Lambda_{\mathcal{K}}$-calcul avec les transitions effectuées par l'algorithme.

Lemme 8.11 *Soit $M \longrightarrow_{\mathcal{K}} N$ une $\mathcal{K}$-réduction. Soient $M^{\mathbf{t}} = M_0^{\mathbf{t}}(M)$ et $N^{\mathbf{t}} = M_0^{\mathbf{t}}(N)$. Alors il existe une réduction $(\mathcal{E}_0(M^{\mathbf{t}}), \Pi) \longrightarrow (S(\mathcal{E}), \overline{S}(\Pi))$ par la règle (R_n) ou (R_0), et on a :*
- *il existe une équation décomposable eq telle que $\mathcal{E}_0(M^{\mathbf{t}}) = \mathcal{E} \cup \{eq\}$*
- *$S(\mathcal{E}) \equiv \mathcal{E}_0(N^{\mathbf{t}})$*
- *$S(Typ(M^{\mathbf{t}})) \equiv Typ(N^{\mathbf{t}})$*
- *$S(VarTyp(x, M^{\mathbf{t}})) \equiv VarTyp(x, N^{\mathbf{t}})$ pour tout x*
- *$S(ftv(M^{\mathbf{t}})) \equiv ftv(N^{\mathbf{t}})$*

où les quatre équivalences ci-dessus font référence aux mêmes renommages θ.

Preuve: Par induction sur la dérivation de $M \longrightarrow_{\mathcal{K}} N$.
- Supposons que $MN \longrightarrow_{\mathcal{K}} M'N$ dérive de $M \longrightarrow_{\mathcal{K}} M'$.
  Soient $M^{\mathbf{t}} = M_0^{\mathbf{t}}(M)$, $M^{\mathbf{t}'} = M_0^{\mathbf{t}}(M')$, $N^{\mathbf{t}} = M_0^{\mathbf{t}}(N)$, $P^{\mathbf{t}} = M_0^{\mathbf{t}}(MN) = (M^{\mathbf{t}}N^{\mathbf{t}}) : t$ et $P^{\mathbf{t}'} = M_0^{\mathbf{t}}(M'N) = (M^{\mathbf{t}'}N^{\mathbf{t}}) : t'$, où t et t' sont fraîches. Par hypothèse d'induction, on sait que $(\mathcal{E}_0(M^{\mathbf{t}}), \Pi)$ se réduit en $(S(\mathcal{E}), \overline{S}(\Pi))$, et qu'il existe une équation décomposable eq et un renommage θ tels que : $\mathcal{E}_0(M^{\mathbf{t}}) = \mathcal{E} \cup \{eq\}$, $S(\mathcal{E}) = \theta(\mathcal{E}_0(M^{\mathbf{t}'}))$, $S(Typ(M^{\mathbf{t}})) = \theta(Typ(M^{\mathbf{t}'}))$, $S(VarTyp(x, M^{\mathbf{t}})) = \theta(VarTyp(x, M^{\mathbf{t}'}))$ pour tout x, et $S(ftv(M^{\mathbf{t}})) = \theta(ftv(M^{\mathbf{t}'}))$.
  Considérons maintenant les équations pour $P^{\mathbf{t}}$:

$$\begin{aligned}\mathcal{E}_0(P^{\mathbf{t}}) &= \mathcal{E}_0(M^{\mathbf{t}}) \cup \mathcal{E}_0(N^{\mathbf{t}}) \cup \{Typ(N^{\mathbf{t}}) \to t \ \doteq\ Typ(M^{\mathbf{t}}) \ \ [ftv(N^{\mathbf{t}})]\} \\ &= \mathcal{E} \cup \{eq\} \cup \mathcal{E}_0(N^{\mathbf{t}}) \cup \{Typ(N^{\mathbf{t}}) \to t \ \doteq\ Typ(M^{\mathbf{t}}) \ \ [ftv(N^{\mathbf{t}})]\}\end{aligned}$$

  Par conséquent, le système $(\mathcal{E}_0(P^{\mathbf{t}}), \Pi)$ peut se réduire en $(S(\mathcal{E}'), \overline{S}(\Pi))$, où $\mathcal{E}' = \mathcal{E} \cup \mathcal{E}_0(N^{\mathbf{t}}) \cup \{Typ(N^{\mathbf{t}}) \to t \ \doteq\ Typ(M^{\mathbf{t}}) \ \ [ftv(N^{\mathbf{t}})]\}$.
  Quelle est la valeur de $S(\mathcal{E}')$? Nous savons déjà que $S(\mathcal{E}) = \theta(\mathcal{E}_0(M^{\mathbf{t}'}))$. Par le Lemme 8.8, on a $dom(S) \subseteq ftv(eq) \subseteq ftv(\mathcal{E}_0(M^{\mathbf{t}}))$. Alors, en utilisant le Lemme 8.3, $dom(S) \subseteq ftv(M^{\mathbf{t}})$. Par le Lemme 8.4, $ftv(M^{\mathbf{t}}) \cap ftv(N^{\mathbf{t}}) = \emptyset$. Donc, $dom(S) \cap ftv(N^{\mathbf{t}}) = \emptyset$. En utilisant à nouveau le Lemme 8.3 pour $N^{\mathbf{t}}$, on obtient $dom(S) \cap ftv(\mathcal{E}_0(N^{\mathbf{t}})) = \emptyset$. Grâce au Lemme 8.5, on trouve finalement : $S(\mathcal{E}_0(N^{\mathbf{t}})) = \mathcal{E}_0(N^{\mathbf{t}})$.
  Par un raisonnement identique, on obtient : $S(Typ(N^{\mathbf{t}})) = Typ(N^{\mathbf{t}})$ et $S(ftv(N^{\mathbf{t}})) = ftv(N^{\mathbf{t}})$. De plus, puisque t fut choisie fraîche après $M^{\mathbf{t}} = M_0^{\mathbf{t}}(M)$, on a : $t \notin ftv(M^{\mathbf{t}})$. Par conséquent, $t \notin dom(S)$ et $S(t) = t$.
  En réunissant ces résultats, on obtient pour $S(\mathcal{E}')$:

$$S(\mathcal{E}') = \theta(\mathcal{E}_0(M^{\mathbf{t}'})) \cup \mathcal{E}_0(N^{\mathbf{t}}) \cup \{Typ(N^{\mathbf{t}}) \to t \ \doteq\ \theta(Typ(M^{\mathbf{t}'})) \ \ [ftv(N^{\mathbf{t}})]\}$$

  Une fois encore, puisque $ftv(\mathcal{E}_0(M^{\mathbf{t}})) \cap (ftv(\mathcal{E}_0(N^{\mathbf{t}})) \cup \{t\}) = \emptyset$, on aurait pu choisir θ de telle sorte que $dom(\theta) \cap (ftv(\mathcal{E}_0(N^{\mathbf{t}})) \cup \{t\}) = \emptyset$. On a donc :

$$S(\mathcal{E}') = \theta(\mathcal{E}_0(M^{\mathbf{t}'})) \cup \theta(\mathcal{E}_0(N^{\mathbf{t}})) \cup \{\theta(Typ(N^{\mathbf{t}})) \to \theta(t) \ \doteq\ \theta(Typ(M^{\mathbf{t}'})) \ \ [\theta(ftv(N^{\mathbf{t}}))]\}$$

  Définissons $\theta' = \theta \circ \{t' \mapsto t\}$. En considérant que, puisque t' est fraîche et n'apparaît nulle part ailleurs, on peut sans danger l'appliquer à toutes ces équations, et introduire t' comme le nouveau noeud-application :

$$S(\mathcal{E}') = \theta'(\mathcal{E}_0(M^{\mathbf{t}'})) \cup \theta'(\mathcal{E}_0(N^{\mathbf{t}})) \cup \{\theta'(Typ(N^{\mathbf{t}})) \to \theta'(t') \ \doteq\ \theta'(Typ(M^{\mathbf{t}'})) \ \ [\theta'(ftv(N^{\mathbf{t}}))]\}$$

Finalement, $S(\mathcal{E}') \equiv \mathcal{E}_0(P^{\mathbf{t}'})$.
Il reste à vérifier les autres propriétés :

$$S(Typ(P^{\mathbf{t}})) = S(t) = t = \theta'(t') = \theta'(Typ(P^{\mathbf{t}'}))$$

Pour tout x :

$$\begin{array}{rcl} S(VarTyp(x, P^{\mathbf{t}})) & = & S(VarTyp(x, M^{\mathbf{t}})) \cup S(VarTyp(x, N^{\mathbf{t}})) \\ & = & \theta(VarTyp(x, M^{\mathbf{t}'})) \cup S(VarTyp(x, N^{\mathbf{t}})) \quad \text{par hypothèse d'induction} \\ & = & \theta(VarTyp(x, M^{\mathbf{t}'})) \cup VarTyp(x, N^{\mathbf{t}}) \quad \text{puisque } dom(S) \cap ftv(N^{\mathbf{t}}) = \emptyset \\ & = & \theta'(VarTyp(x, M^{\mathbf{t}'})) \cup \theta'(VarTyp(x, N^{\mathbf{t}})) \quad \text{mêmes remarques que ci-dessus} \\ & = & \theta'(VarTyp(x, P^{\mathbf{t}'})) \end{array}$$

Et finalement, avec les mêmes arguments :

$$\begin{array}{rcl} S(ftv(P^{\mathbf{t}})) & = & S(ftv(M^{\mathbf{t}})) \cup S(ftv(N^{\mathbf{t}})) \cup S(\{t\}) \\ & = & \theta(ftv(M^{\mathbf{t}'})) \cup ftv(N^{\mathbf{t}}) \cup \{t\} \\ & = & \theta'(ftv(M^{\mathbf{t}'})) \cup \theta'(ftv(N^{\mathbf{t}})) \cup \theta'(\{t'\}) \\ & = & \theta'(ftv(P^{\mathbf{t}'})) \end{array}$$

– Le cas de l'induction $\dfrac{N \longrightarrow_{\mathcal{K}} N'}{MN \longrightarrow_{\mathcal{K}} MN'}$ est exactement le même que le précédent, en échangeant les rôles de M et N.
Les cas de $\dfrac{M \longrightarrow_{\mathcal{K}} M'}{[M, N] \longrightarrow_{\mathcal{K}} [M', N]}$ et $\dfrac{N \longrightarrow_{\mathcal{K}} N'}{[M, N] \longrightarrow_{\mathcal{K}} [M, N']}$ sont en fait similaires et même plus simples, car il n'y a pas à traiter les noeuds-applications t et t', et les équations supplémentaires qu'ils induisent.
Le cas de $\dfrac{M \longrightarrow_{\mathcal{K}} M'}{\lambda x M \longrightarrow_{\mathcal{K}} \lambda x M'}$ est également très similaire, et nous ne le détaillerons donc pas ici (c'est ici que l'hypothèse inductive sur $VarTyp$ est nécessaire).
– Les seuls cas restants sont ceux des axiomes pour un redex. Nous commencerons par la règle

$$P = [\lambda x M, N_1, \ldots, N_p]T \longrightarrow_{\mathcal{K}} Q = [M, N_1, \ldots, N_p, T]$$

avec $x \notin fv(M)$.
Avec des notations évidentes, on convertit P en sa version annotée, et on obtient :

$$P^{\mathbf{t}} = ([\lambda x M^{\mathbf{t}}, N_1^{\mathbf{t}}, \ldots, N_p^{\mathbf{t}}] \;\; T^{\mathbf{t}}) : t$$

où t est fraîche. Si on note τ pour $Typ(M^{\mathbf{t}})$, alors $Typ([\lambda x M^{\mathbf{t}}, N_1^{\mathbf{t}}, \ldots, N_p^{\mathbf{t}}]) = \omega \to \tau$. On obtient l'ensemble des équations pour $P^{\mathbf{t}}$:

$$\mathcal{E}_0(P^{\mathbf{t}}) = \mathcal{E}_0(M^{\mathbf{t}}) \cup \bigcup_{1 \le i \le p} \mathcal{E}_0(N_i^{\mathbf{t}}) \cup \mathcal{E}_0(T^{\mathbf{t}}) \cup \{Typ(T^{\mathbf{t}}) \to t \;\stackrel{\perp}{=}\; \omega \to \tau \;\; [ftv(T^{\mathbf{t}})]\}$$

Arrivé à ce point, nous pouvons appliquer la règle (R_0) à cette dernière équation, de telle sorte que le système $(\mathcal{E}_0(P^{\mathbf{t}}), \Pi)$ évolue en $(S(\mathcal{E}), \overline{S}(\Pi))$ avec :

$$\mathcal{E} = \mathcal{E}_0(M^{\mathbf{t}}) \cup \bigcup_{1 \le i \le p} \mathcal{E}_0(N_i^{\mathbf{t}}) \cup \mathcal{E}_0(T^{\mathbf{t}})$$

et $S = \{t \mapsto \tau, \emptyset\}$.
Maintenant, puisque t est à la racine et a été choisie fraîche, cette variable n'apparaît ni dans $M^{\mathbf{t}}$, ni dans $N_i^{\mathbf{t}}$, ni dans $T^{\mathbf{t}}$. Par conséquent, $S(\mathcal{E}) = \mathcal{E}$. De plus, $S(Typ(P^{\mathbf{t}})) = S(t) = \tau = Typ([M^{\mathbf{t}}, N_1^{\mathbf{t}}, \ldots, N_p^{\mathbf{t}}, T^{\mathbf{t}}])$. Et,

$$\begin{aligned} S(ftv(P^{\mathbf{t}})) &= S(ftv(M^{\mathbf{t}})) \cup \bigcup_{1 \leq i \leq p} S(ftv(N_i^{\mathbf{t}})) \cup S(ftv(T^{\mathbf{t}})) \cup S(\{t\}) \\ &= ftv(M^{\mathbf{t}}) \cup \bigcup_{1 \leq i \leq p} ftv(N_i^{\mathbf{t}}) \cup ftv(T^{\mathbf{t}}) \\ &= ftv([M^{\mathbf{t}}, N_1^{\mathbf{t}}, \ldots, N_p^{\mathbf{t}}, T^{\mathbf{t}}]) \end{aligned}$$

Pour les mêmes raisons :

$$\begin{aligned} S(VarTyp(y, P^{\mathbf{t}})) &= S(VarTyp(y, M^{\mathbf{t}})) \cup \bigcup_{1 \leq i \leq p} S(VarTyp(y, N_i^{\mathbf{t}})) \cup S(VarTyp(y, T^{\mathbf{t}})) \\ &= VarTyp(y, M^{\mathbf{t}}) \cup \bigcup_{1 \leq i \leq p} VarTyp(y, N_i^{\mathbf{t}}) \cup VarTyp(y, T^{\mathbf{t}}) \\ &= VarTyp(y, [M^{\mathbf{t}}, N_1^{\mathbf{t}}, \ldots, N_p^{\mathbf{t}}, T^{\mathbf{t}}]) \end{aligned}$$

pour tout y (même si $y = x$ car $x \notin fv(M)$).
Finalement, il est facile de vérifier que $Q^{\mathbf{t}} \equiv [M^{\mathbf{t}}, N_1^{\mathbf{t}}, \ldots, N_p^{\mathbf{t}}, T^{\mathbf{t}}]$, et donc d'en déduire les quatre propriétés d'équivalence recherchées.

– Le dernier cas à considérer est celui d'un "vrai" redex :

$$P = [\lambda x M, N_1, \ldots, N_p]T \longrightarrow_{\mathcal{K}} Q = [\{x \mapsto T\}M, N_1, \ldots, N_p]$$

avec $x \in fv(M)$.
Comme dans le cas précédent, on commence par convertir P en un terme annoté :

$$P^{\mathbf{t}} = ([\lambda x M^{\mathbf{t}}, N_1^{\mathbf{t}}, \ldots, N_p^{\mathbf{t}}] \;\; T^{\mathbf{t}}) : t$$

où t est fraîche. Si on note $VarTyp(x, M^{\mathbf{t}}) = \{t_1, \ldots, t_n\}$ (avec $n > 0$) et $\tau = Typ(M^{\mathbf{t}})$, alors $Typ([\lambda x M^{\mathbf{t}}, N_1^{\mathbf{t}}, \ldots, N_p^{\mathbf{t}}]) = t_1, \ldots, t_n \to \tau$. On obtient l'ensemble des équations correspondant :

$$\mathcal{E}_0(P^{\mathbf{t}}) = \mathcal{E}_0(M^{\mathbf{t}}) \cup \bigcup_{1 \leq i \leq p} \mathcal{E}_0(N_i^{\mathbf{t}}) \cup \mathcal{E}_0(T^{\mathbf{t}}) \cup \{Typ(T^{\mathbf{t}}) \to t \;\stackrel{\perp}{=}\; t_1, \ldots, t_n \to \tau \;\; [ftv(T^{\mathbf{t}})]\}$$

La dernière équation a la bonne forme pour appliquer la règle (R_n), et le système $(\mathcal{E}_0(P^{\mathbf{t}}), \Pi)$ évolue en $(S(\mathcal{E}), \overline{S}(\Pi))$ avec :

$$\mathcal{E} = \mathcal{E}_0(M^{\mathbf{t}}) \cup \bigcup_{1 \leq i \leq p} \mathcal{E}_0(N_i^{\mathbf{t}}) \cup \mathcal{E}_0(T^{\mathbf{t}})$$

et $S = \{t_i \mapsto \langle Typ(T^{\mathbf{t}})\rangle^i, \langle ftv(T^{\mathbf{t}})\rangle^i\}_{1 \leq i \leq n} :: \{t \mapsto \tau, \emptyset\} :: D(n, ftv(T^{\mathbf{t}}))$. On décompose cette substitution en $S = S_1 :: S_2 :: S_3$.
Par le Lemme 8.4, $ftv(M^{\mathbf{t}}) \cap ftv(T^{\mathbf{t}}) = \emptyset$. Par conséquent (Lemme 8.5), $S_3(\mathcal{E}_0(M^{\mathbf{t}})) = \mathcal{E}_0(M^{\mathbf{t}})$. De plus, puisque t est fraîche, $t \notin ftv(M^{\mathbf{t}})$. Donc, $S(\mathcal{E}_0(M^{\mathbf{t}})) = S_1(\mathcal{E}_0(M^{\mathbf{t}}))$.

Pour les mêmes raisons, la même conclusion s'applique à $\mathcal{E}_0(N_i^{\mathbf{t}})$. Mais, puisque $t_i \in ftv(M^{\mathbf{t}})$, on ne peut avoir $t_i \in ftv(N_i^{\mathbf{t}})$. Et donc : $S(\mathcal{E}_0(N_i^{\mathbf{t}})) = \mathcal{E}_0(N_i^{\mathbf{t}})$ pour tout i (i.e. les termes "oubliés" ne sont pas affectés par la réduction).
De façon évidente, pour toute équation dans $\mathcal{E}_0(T^{\mathbf{t}})$, son noeud-application se trouve dans le domaine de S_3 (par définition). Donc :

$$S_3(\mathcal{E}_0(T^{\mathbf{t}})) = dupl_n(\mathcal{E}_0(T^{\mathbf{t}})) = \bigcup_{1\leq i\leq n} \mathcal{E}_0(\langle T^{\mathbf{t}}\rangle^i)$$

par le Lemme 8.7. En outre, $t_i \notin \langle ftv(T^{\mathbf{t}})\rangle^i = ftv(\langle T^{\mathbf{t}}\rangle^i)$ (Lemme 8.6). Le même raisonnement s'applique pour t. Finalement, $S(\mathcal{E}_0(T^{\mathbf{t}})) = \bigcup_{1\leq i\leq n} \mathcal{E}_0(\langle T^{\mathbf{t}}\rangle^i)$.
En réunissant ces résultats, on obtient finalement la valeur de $S(\mathcal{E})$:

$$S(\mathcal{E}) = S_1(\mathcal{E}_0(M^{\mathbf{t}})) \cup \bigcup_{1\leq i\leq p} \mathcal{E}_0(N_i^{\mathbf{t}}) \cup \bigcup_{1\leq i\leq n} \mathcal{E}_0(\langle T^{\mathbf{t}}\rangle^i)$$

On sait déjà que $M^{\mathbf{t}}$ contient exactement une occurrence de chaque axiome $x : t_i$. Donc, on peut voir ce terme comme l'imbrication de n contextes : $M^{\mathbf{t}} = C^{\mathbf{t}}[x : t_1, \ldots, x : t_n]$ (on utilise une notation évidente ; comme on va le voir, l'ordre de l'imbrication n'a pas d'influence), où $x \notin bv(C^{\mathbf{t}})$. De plus, puisqu'il est toujours possible de renommer les noms liés dans P, on peut considérer que $fv(T^{\mathbf{t}}) \cap bv(C^{\mathbf{t}}) = \emptyset$. Par conséquent, on peut appliquer le Lemme 8.10 n fois et on obtient :

$$S(\mathcal{E}) = \bigcup_{1\leq i\leq p} \mathcal{E}_0(N_i^{\mathbf{t}}) \cup \mathcal{E}_0(C^{\mathbf{t}}[\langle T^{\mathbf{t}}\rangle^1, \ldots, \langle T^{\mathbf{t}}\rangle^n])$$

(Pour être plus précis, pour appliquer ce Lemme, il faudrait appliquer partiellement la substitution S_1 aux termes $\mathcal{E}_0(\langle T^{\mathbf{t}}\rangle^i)$, suivant la valeur de i et le niveau courant d'imbrication ; mais, comme on l'a déjà vu, ceci est valide, puisque $t_i \notin ftv(\langle T^{\mathbf{t}}\rangle^i)$.)
Il ne reste plus qu'à remarquer que $C^{\mathbf{t}}[\langle T^{\mathbf{t}}\rangle^1, \ldots, \langle T^{\mathbf{t}}\rangle^n]$ correspond exactement au terme $M_0^{\mathbf{t}}(\{x \mapsto T\}M)$ à un renommage près, i.e. M dans lequel les n occurrences de x ont été remplacées par n copies de T. Et, en conclusion, $S(\mathcal{E}) \equiv \mathcal{E}_0(Q^{\mathbf{t}})$ (nous ne détaillons pas les renommages).
Il ne reste qu'à vérifier les autres propriétés :

$$\begin{aligned}
S(Typ(P^{\mathbf{t}})) &= S(t) \\
&= S_1 :: S_2(t) \quad \text{puisque } t \notin ftv(T^{\mathbf{t}}) \\
&= S_1(\tau) \\
&= Typ(C^{\mathbf{t}}[\langle T^{\mathbf{t}}\rangle^1, \ldots, \langle T^{\mathbf{t}}\rangle^n]) \quad \text{en appliquant le Lemme 8.10 } n \text{ fois} \\
&\equiv Typ(Q^{\mathbf{t}})
\end{aligned}$$

De même (nous ne détaillons pas les arguments) :

$$\begin{aligned}
S(ftv(P^{\mathbf{t}})) &= S(ftv(M^{\mathbf{t}})) \cup \bigcup_{1\leq i\leq p} S(ftv(N_i^{\mathbf{t}})) \cup S(ftv(T^{\mathbf{t}})) \cup S(\{t\}) \\
&= S_1(ftv(M^{\mathbf{t}})) \cup \bigcup_{1\leq i\leq p} ftv(N_i^{\mathbf{t}}) \cup \bigcup_{1\leq i\leq n} ftv(\langle T^{\mathbf{t}}\rangle^i) \\
&= ftv(C^{\mathbf{t}}[\langle T^{\mathbf{t}}\rangle^1, \ldots, \langle T^{\mathbf{t}}\rangle^n]) \cup \bigcup_{1\leq i\leq p} ftv(N_i^{\mathbf{t}}) \cup \bigcup_{1\leq i\leq n} ftv(\langle T^{\mathbf{t}}\rangle^i)
\end{aligned}$$

Notons que ces derniers termes $ftv(\langle T^{\mathbf{t}}\rangle^i)$ sont inclus dans le premier terme. Donc, $S(ftv(P^{\mathbf{t}})) \equiv ftv(Q^{\mathbf{t}})$.
Pour l'ensemble des variables de type, il faut distinguer les cas pour x et pour $y \neq x$:

$$\begin{aligned} S(VarTyp(x,P^{\mathbf{t}})) &= S(\emptyset) \cup \bigcup_{1\leq i\leq p} S(VarTyp(x,N_i^{\mathbf{t}})) \cup S(VarTyp(x,T^{\mathbf{t}})) \\ &= \bigcup_{1\leq i\leq p} VarTyp(x,N_i^{\mathbf{t}}) \cup \bigcup_{1\leq i\leq n} VarTyp(x,\langle T^{\mathbf{t}}\rangle^i) \\ &= \bigcup_{1\leq i\leq p} VarTyp(x,N_i^{\mathbf{t}}) \cup VarTyp(x,C^{\mathbf{t}}[\langle T^{\mathbf{t}}\rangle^1,\ldots,\langle T^{\mathbf{t}}\rangle^n]) \end{aligned}$$

Notons que x ne peut apparaître libre dans $C^{\mathbf{t}}$ puisque nous connaissons toutes ses occurrences dans $M^{\mathbf{t}}$; ses seules occurrences sont donc dans $\langle T^{\mathbf{t}}\rangle^i$, comme prévu.
Pour $y \neq x$:

$$\begin{aligned} S(VarTyp(y,P^{\mathbf{t}})) &= S(VarTyp(y,M^{\mathbf{t}})) \cup \bigcup_{1\leq i\leq p} S(VarTyp(y,N_i^{\mathbf{t}})) \cup S(VarTyp(y,T^{\mathbf{t}})) \\ &= VarTyp(y,M^{\mathbf{t}}) \cup \bigcup_{1\leq i\leq p} VarTyp(y,N_i^{\mathbf{t}}) \cup \bigcup_{1\leq i\leq n} VarTyp(y,\langle T^{\mathbf{t}}\rangle^i) \\ &= \bigcup_{1\leq i\leq p} VarTyp(y,N_i^{\mathbf{t}}) \cup VarTyp(y,C^{\mathbf{t}}[\langle T^{\mathbf{t}}\rangle^1,\ldots,\langle T^{\mathbf{t}}\rangle^n]) \end{aligned}$$

Ici y apparaît dans $M^{\mathbf{t}}$ si et seulement si elle apparaît dans $C^{\mathbf{t}}$; en outre, notons que l'ensemble $VarTyp(y,M^{\mathbf{t}})$ ne peut pas être affecté par S_1, puisque des variables différentes ont des types différents, et les t_i sont ceux de x seulement.
Dans les deux cas, on obtient finalement $S(VarTyp(z,P^{\mathbf{t}})) \equiv VarTyp(z,Q^{\mathbf{t}})$, ce qui termine la preuve.

□

Le lemme suivant traite le cas d'un terme qui ne peut plus être réduit.

Lemme 8.12 *$M \in \mathcal{N}_{\mathcal{K}}$ si et seulement si $(\mathcal{E}_0(M_0^{\mathbf{t}}(M)),\Pi)$ ne peut se réduire par la règle (R_n) ou (R_0).*

Preuve: Soit $M \in \Lambda_{\mathcal{K}}$; examinons l'ensemble d'équations $\mathcal{E} = \mathcal{E}_0(M_0^{\mathbf{t}}(M))$ qui lui correspond. A chaque application dans M correspond exactement une équation dans $\mathcal{E}$. Comme on le voit sur la règle, la seule condition pour qu'un système soit réductible est qu'il contienne une équation avec un type fonctionnel dans sa partie droite. Avec la définition de $\mathcal{E}_0$, on peut voir que ceci a lieu si et seulement si le type de droite dans l'équation correspondante (i.e. le type du terme de gauche dans l'application) est non trivial. Et, d'après la définition de Typ, on note que ceci arrive si et seulement si ce terme de gauche est de la forme $[\lambda x N, N_1, \ldots, N_p]$. En conclusion, le système $(\mathcal{E},\Pi)$ se réduit par (R_n) ou (R_0) si et seulement si M possède un redex, c'est-à-dire si et seulement si $M \notin \mathcal{N}_{\mathcal{K}}$. □

Enfin, il nous reste à énoncer une réciproque au Lemme 8.11 :

Lemme 8.13 *Soit $M^{\mathbf{t}} = M_0^{\mathbf{t}}(M)$. Si $(\mathcal{E}_0(M^{\mathbf{t}}),\Pi) \longrightarrow (S(\mathcal{E}),\overline{S}(\Pi))$ par une des règles (R_n) ou (R_0), alors il existe un terme N tel que $M \longrightarrow_{\mathcal{K}} N$ et $S(\mathcal{E}) \equiv \mathcal{E}_0(M_0^{\mathbf{t}}(N))$.*

Preuve: Elle suit celle du lemme précédent. Si le système peut réduire, cela signifie que M possède un redex, et qu'il existe un terme N tel que $M \longrightarrow_{\mathcal{K}} N$. Par le Lemme 8.11, nous savons qu'il existe S' et $\mathcal{E}'$ tels que $(\mathcal{E}_0(M^{\mathbf{t}}), \Pi) \longrightarrow (S'(\mathcal{E}'), \overline{S'}(\Pi))$ et $S'(\mathcal{E}') \equiv \mathcal{E}_0(M_0^{\mathbf{t}}(N))$. A l'aide de la correspondance redex/équation décomposable vue précédemment, il est alors facile de vérifier que ces réductions du système sont en fait la même. □

8.1.4 Résultat principal

Grâce aux lemmes de la partie précédente et aux résultats théoriques concernant le $\Lambda_{\mathcal{K}}$-calcul, on peut finalement montrer que l'algorithme termine exactement pour les termes typables.

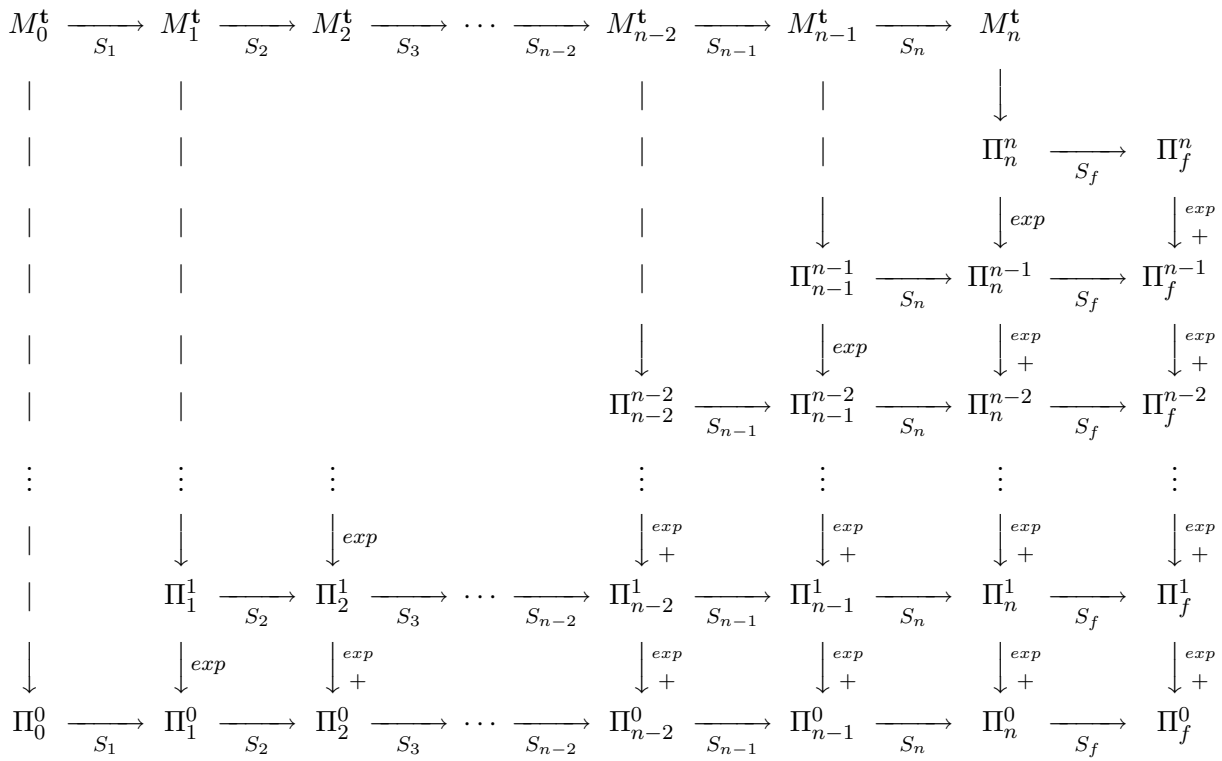
Théorème 8.14 *Un terme M est typable si et seulement si le système $Syst_0(M)$ converge.*

Preuve: Nous avons vu (Lemme 6.6 et Théorème 6.7) que pour le $\Lambda_{\mathcal{K}}$-calcul : $\mathcal{SN}_{\mathcal{K}} = \mathcal{WN}_{\mathcal{K}} =$ l'ensemble des termes typables. Si M est typable, il est normalisable. Par les Lemmes 8.11 et 8.12, le système $Syst_0(M)$ est normalisable pour les règles (R_n) et (R_0). Puis, comme nous l'avons déjà noté dans la présentation, la règle (R_f) ne peut être appliquée qu'un nombre fini de fois (puisque le nombre d'équations restantes décroît strictement). Donc, $Syst_0(M)$ converge.

D'un autre côté, si $Syst_0(M)$ converge, par les Lemmes 8.13 et 8.12, nous pouvons conclure que $M \in \mathcal{WN}_{\mathcal{K}}$. □

8.2 Résultats relatifs au squelette de preuve

Pour la seconde partie de la preuve concernant les squelettes, il sera utile de se référer au diagramme commutatif suivant :



Dans ce diagramme, la première ligne $M_0^{\mathbf{t}} \longrightarrow \cdots \longrightarrow M_n^{\mathbf{t}}$ représente les réductions d'un terme M jusqu'à sa forme normale. Les substitutions S_i sont celles induites par l'algorithme. Les flèches verticales en pointillés représentent Π_0, autrement dit $\Pi_i^i = \Pi_0(M_i^{\mathbf{t}})$. Le squelette de preuve Π_j^i est le résultat de l'application des substitutions S_1 à S_j à Π_i^i. La dernière substitution S_f représente les substitutions induites par la règle finale (R_f). Par conséquent, Π_f^0 est en fait le résultat fourni par $Syst_0(M)$.

Nous allons définir une relation de *β-expansion* $\xrightarrow{exp}$ entre les squelettes de preuve. Le reste de la preuve sera consacré à montrer que les flèches $\xrightarrow{exp}$ dans le schéma ci-dessus sont vérifiées, et que les squelettes de la dernière colonne sont des arbres de typage valides.

8.2.1 Cas d'un terme en forme normale

Dans cette section, nous montrons que pour un terme normal l'algorithme retourne un arbre de typage valide.

Tout d'abord, de façon analogue au λ-calcul, il est possible de donner une caractérisation syntaxique des termes normaux pour le $\Lambda_{\mathcal{K}}$-calcul :

Définition 8.15 *Soit $M \in \Lambda_{\mathcal{K}}$. M est dit en* forme normale *s'il appartient à la sous-grammaire suivante de $\Lambda_{\mathcal{K}}$:*

$$\begin{cases} M & ::= & P \mid \lambda x M \mid [M_1, M_2] \\ P & ::= & x \mid PM \mid [P, M] \end{cases}$$

Lemme 8.16 *$M \in \mathcal{N}_{\mathcal{K}}$ si et seulement si M est en forme normale.*

Preuve: Un terme M est normal si et seulement si il n'a pas de redex, c'est-à-dire s'il ne possède pas de λ en partie gauche d'une application, éventuellement sous un opérateur $[\_, \_]$. Il n'est pas difficile de vérifier que la grammaire des formes normales correspond exactement à ces conditions. □

Définition 8.17 *Si Π est un squelette de preuve, on notera $Typ(\Pi)$, $Env(\Pi)$ et $Term(\Pi)$ respectivement le type, l'environnement et le terme apparaissant à la racine de l'arbre.*

Pour les termes en forme normale, il existe un algorithme standard de construction d'un arbre de typage valide ; il s'agit de l'adaptation de l'algorithme équivalent pour le λ-calcul.

Définition 8.18 *L'algorithme suivant construit un squelette de preuve pour les termes en forme*

normale :

$$Canonic_1(P) = Canonic_2(P, t) \quad \text{où } t \text{ est une variable de type fraîche}$$

$$Canonic_1(\lambda x M) = \begin{cases} \texttt{let}\ \ \Pi = Canonic_1(M) \\ \texttt{return}\ \dfrac{\Pi}{Env(\Pi) \setminus x \vdash \lambda x M : Env(\Pi)(x) \to Typ(\Pi)}\,(FUN) \end{cases}$$

$$Canonic_1([M_1, M_2]) = \begin{cases} \texttt{let}\ \ \Pi_1 = Canonic_1(M_1) \\ \texttt{let}\ \ \Pi_2 = Canonic_1(M_2) \\ \texttt{return}\ \dfrac{\Pi_1 \qquad \Pi_2}{Env(\Pi_1), Env(\Pi_2) \vdash [M_1, M_2] : Typ(\Pi_1)}\,(FORGET) \end{cases}$$

$$Canonic_2(x, \overline{\tau}) = \ \frac{}{x : \overline{\tau} \vdash x : \overline{\tau}}\,(ID)$$

$$Canonic_2(PM, \overline{\tau}) = \begin{cases} \texttt{let}\ \ \Pi_2 = Canonic_1(M) \\ \texttt{let}\ \ \Pi_1 = Canonic_2(P, Typ(\Pi_2) \to \overline{\tau}) \\ \texttt{return}\ \dfrac{\Pi_1 \qquad \Pi_2}{Env(\Pi_1), Env(\Pi_2) \vdash PM : \overline{\tau}}\,(APPL) \end{cases}$$

$$Canonic_2([P, M], \overline{\tau}) = \begin{cases} \texttt{let}\ \ \Pi_1 = Canonic_2(P, \overline{\tau}) \\ \texttt{let}\ \ \Pi_2 = Canonic_1(M) \\ \texttt{return}\ \dfrac{\Pi_1 \qquad \Pi_2}{Env(\Pi_1), Env(\Pi_2) \vdash [P, M] : \overline{\tau}}\,(FORGET) \end{cases}$$

Lemme 8.19 *Pour un terme M en forme normale, $Canonic_1(M)$ est un arbre de typage valide pour M. De plus, le typage retourné est principal.*

Preuve: Il s'agit d'un résultat classique en λ-calcul (voir par exemple [CDC80] ou [Kri90], Chapitre III, Section 3), dont l'adaptation au $\Lambda_{\mathcal{K}}$-calcul se fait sans difficulté. □

De plus, cet arbre de typage est précisément celui retourné par l'algorithme pour les termes en forme normale :

Lemme 8.20 *Soient $M \in \mathcal{N}_{\mathcal{K}}$ et $Syst_0(M) \longrightarrow_f^* (\emptyset, \Pi)$. Alors, on a : $\Pi \equiv Canonic_1(M)$.*

Preuve: Par induction sur M en utilisant la grammaire caractéristique des formes normales données en Définition 8.15. □

Corollaire 8.21 *Pour tout $M \in \mathcal{N}_{\mathcal{K}}$, $Syst_0(M)$ renvoie un arbre de typage valide pour M.*

8.2.2 Prédicat de compatibilité

Pour des raisons essentiellement techniques, il n'est pas possible de donner des résultats généraux sur l'application d'une duplication $D(n, T)$ quelconque à un squelette de preuve Π quelconque. Afin de pouvoir mener les inductions à leur terme, nous avons besoin que T vérifie quelques propriétés vis-à-vis de Π. Pour un squelette donné, seuls certains territoires T seront donc "valides" ; le prédicat $Compat(T, \Pi)$ formalise précisément cette notion. Bien entendu, nous verrons dans la suite que les duplications $D(n, T)$ générées par l'algorithme dans une utilisation normale (i.e. en partant d'un état $Syst_0(M)$) sont compatibles avec le squelette courant du système.

Définition 8.22 *Pour un type $\overline{\tau}$, on définit son* type le plus à droite *comme la variable de type apparaissant le plus à droite des flèches :*

$$\begin{cases} rgt(t) = t \\ rgt(\overline{\tau_1}, \ldots, \overline{\tau_n} \to \overline{\sigma}) = rgt(\overline{\sigma}) \end{cases}$$

Dans l'algorithme, les territoires sont issus de l'ensemble des variables de types contenus dans un terme argument $N^{\mathbf{t}}$ (pour une application $(M^{\mathbf{t}}N^{\mathbf{t}}) : t$), par exemple $T = ftv(N^{\mathbf{t}})$. Comme on le montrera plus loin, cet ensemble est aussi égal à $ftv(\Pi_0(N^{\mathbf{t}}))$, et cet ensemble caractérise de façon unique le sous-arbre $\Pi_0(N^{\mathbf{t}})$. Intuitivement, les seuls territoires admissibles pour un squelette de preuve donné sont tels qu'ils correspondent uniquement aux variables de types contenues dans un sous-arbre droit d'un noeud (APPL). Ceci peut se vérifier simplement en s'assurant que pour tous les autres sous-arbres Π, la variable $rgt(\overline{\tau})$, où $\overline{\tau}$ est le type à la racine de Π, n'est pas dans le territoire. De plus, comme il n'y a pas de duplication pour les β_K-redex, on imposera que le type à la racine du sous-arbre gauche soit de la forme $\overline{\tau_1}, \ldots, \overline{\tau_n} \to \overline{\sigma}$ avec $n \geq 1$.

Définition 8.23 *Soit T un ensemble de variables de type. On dira que T est* "compatible avec" *un squelette de preuve Π si le prédicat $Compat(T, \Pi)$ défini par induction ci-dessous est vérifié :*

$$Compat\left(T, \frac{}{\Gamma \vdash x : \overline{\tau}}\,(ID)\right) = (rgt(\overline{\tau}) \notin T)$$

$$Compat\left(T, \frac{\Pi}{\Gamma \vdash \lambda x M : \overline{\tau}}\,(FUN)\right) = Compat(T, \Pi) \quad et \quad (rgt(\overline{\tau}) \notin T)$$

$$Compat\left(T, \frac{\Pi_1 \quad \Pi_2}{\Gamma \vdash [M, N] : \overline{\tau}}\,(FORGET)\right) =$$
$$Compat(T, \Pi_1) \quad et \quad Compat(T, \Pi_2) \quad et \quad (rgt(\overline{\tau}) \notin T)$$

$$Compat\left(\frac{\Pi \quad \Pi_1 \cdots \Pi_n}{\Gamma \vdash MN : \overline{\tau}}\,(APPL)\right) = Compat(T, \Pi) \quad et \quad (rgt(\overline{\tau}) \notin T) \quad et$$
$$\forall_{1 \leq i \leq n} \left(Compat(T, \Pi_i) \quad ou \quad (Typ(\Pi) = \overline{\tau_1}, \ldots, \overline{\tau_n} \to \overline{\sigma} \quad avec\ n \geq 1 \ et \quad ftv(\Pi_i) \subseteq T)\right)$$

Lemme 8.24 *Si $T \cap ftv(\Pi) = \emptyset$, alors $Compat(T, \Pi)$ est vrai.*

Preuve: Toutes les variables de type qui apparaisse dans Π sont bien entendu incluses dans $ftv(\Pi)$, de telle sorte que la condition $rgt(\overline{\tau}) \notin T$ est vérifiée pour tout sous-arbre de Π. Il est alors trivial d'en déduire $Compat(T, \Pi)$ par induction sur Π. □

Lemme 8.25
- $Typ(\Pi_0(M^{\mathbf{t}})) = \overline{Typ(M^{\mathbf{t}})}$
- $ftv(\Pi_0(M^{\mathbf{t}})) = ftv(M^{\mathbf{t}})$

Preuve: Trivial. □

Dans la suite, nous allons mener un raisonnement par induction sur le nombre d'étapes de l'algorithme. Nous montrerons plus loin qu'à tout moment les territoires contenus dans les équations sont compatibles avec le squelette de preuve en cours de considération. Afin de pouvoir démarrer le raisonnement à partir de l'état initial, il faut vérifier qu'il y a compatibilité entre le squelette de preuve initial et les territoires apparaissant dans les duplications potentielles :

Lemme 8.26 *Soient M un terme et $M^{\mathbf{t}} = M_0^{\mathbf{t}}(M)$. Soit T le territoire d'une équation de $\mathcal{E}_0(M^{\mathbf{t}})$ qui soit immédiatement réductible par la règle (R_n). Alors, le prédicat $Compat(T, \Pi_0(M^{\mathbf{t}}))$ est vrai.*

Preuve: Par induction sur $M^{\mathbf{t}}$.

- Si $M^{\mathbf{t}} = x : t$, alors $\mathcal{E}_0(M^{\mathbf{t}}) = \emptyset$, et le résultat est vérifié.
- Supposons que $M^{\mathbf{t}} = \lambda x N^{\mathbf{t}}$. Dans ce cas, $\mathcal{E}_0(M^{\mathbf{t}}) = \mathcal{E}_0(N^{\mathbf{t}})$. Si T est un territoire satisfaisant la condition pour $M^{\mathbf{t}}$, il la satisfait aussi pour $N^{\mathbf{t}}$. Par hypothèse d'induction, $Compat(T, \Pi_0(N^{\mathbf{t}}))$ est vrai. Ceci implique que $rgt(Typ(\Pi_0(N^{\mathbf{t}}))) \notin T$ (facile). Finalement, puisque $rgt(\overline{Typ(\lambda x N^{\mathbf{t}})}) = rgt(\overline{Typ(N^{\mathbf{t}})}) = rgt(Typ(\Pi_0(N^{\mathbf{t}})))$ (par le Lemme 8.25), on peut dériver $Compat(T, \Pi_0(M^{\mathbf{t}}))$.
- Supposons que $M^{\mathbf{t}} = [N_1^{\mathbf{t}}, N_2^{\mathbf{t}}]$. Dans ce cas, $\mathcal{E}_0(M^{\mathbf{t}}) = \mathcal{E}_0(N_1^{\mathbf{t}}) \cup \mathcal{E}_0(N_2^{\mathbf{t}})$, de telle sorte que l'équation dont le territoire est T doit se trouver dans l'un de ces deux ensembles. Supposons que l'équation choisie est dans $\mathcal{E}_0(N_1^{\mathbf{t}})$. Par hypothèse d'induction, on obtient $Compat(T, \Pi_0(N_1^{\mathbf{t}}))$. Par le Lemme 8.4, $ftv(N_1^{\mathbf{t}}) \cap ftv(N_2^{\mathbf{t}}) = \emptyset$. Donc, avec le Lemme 8.25, $ftv(N_1^{\mathbf{t}}) \cap ftv(\Pi_0(N_2^{\mathbf{t}})) = \emptyset$. Puisque T est le territoire d'une équation de $\mathcal{E}_0(N_1^{\mathbf{t}})$, c'est un ensemble de variables de type d'un sous-terme de $N_1^{\mathbf{t}}$. Par conséquent, $T \cap ftv(\Pi_0(N_2^{\mathbf{t}})) = \emptyset$. Par le Lemme 8.24, on a $Compat(T, \Pi_0(N_2^{\mathbf{t}}))$.
  Si T avait été le territoire d'une équation de $\mathcal{E}_0(N_2^{\mathbf{t}})$, il aurait suffit d'intervertir $N_1^{\mathbf{t}}$ et $N_2^{\mathbf{t}}$. Donc, dans les deux cas, on a $Compat(T, \Pi_0(N_1^{\mathbf{t}}))$ et $Compat(T, \Pi_0(N_2^{\mathbf{t}}))$. Il reste encore à ajouter que $rgt(\overline{Typ(M^{\mathbf{t}})}) = rgt(\overline{Typ(N_1^{\mathbf{t}})}) = rgt(Typ(\Pi_0(N_1^{\mathbf{t}}))) \notin T$, et on peut finalement conclure que $Compat(T, \Pi_0(M^{\mathbf{t}}))$.
- Supposons que $M^{\mathbf{t}} = (N_1^{\mathbf{t}} N_2^{\mathbf{t}}) : t$ où $t \notin ftv(N_1^{\mathbf{t}}) \cup ftv(N_2^{\mathbf{t}})$. Dans ce cas, $\mathcal{E}_0(M^{\mathbf{t}}) = \mathcal{E}_0(N_1^{\mathbf{t}}) \cup \mathcal{E}_0(N_2^{\mathbf{t}}) \cup \{Typ(N_2^{\mathbf{t}}) \to t \stackrel{\perp}{=} Typ(N_1^{\mathbf{t}}) \ \ [ftv(N_2^{\mathbf{t}})]\}$. Si T est le territoire d'une équation de $\mathcal{E}_0(N_1^{\mathbf{t}})$ ou de $\mathcal{E}_0(N_2^{\mathbf{t}})$, on raisonne comme précédemment pour obtenir $Compat(T, \Pi_0(N_1^{\mathbf{t}}))$ et $Compat(T, \Pi_0(N_2^{\mathbf{t}}))$. Et comme précédemment, puisque T est un sous-ensemble de $ftv(N_1^{\mathbf{t}})$ ou de $ftv(N_2^{\mathbf{t}})$, on a $t \notin T$, et on peut dériver $Compat(T, \Pi_0(M^{\mathbf{t}}))$.
  Le seul cas restant est celui où l'équation choisie est celle induite par cette application, à savoir $Typ(N_2^{\mathbf{t}}) \to t \stackrel{\perp}{=} Typ(N_1^{\mathbf{t}}) \ \ [ftv(N_2^{\mathbf{t}})]$. Dans ce cas, $T = ftv(N_2^{\mathbf{t}})$ et, puisque par hypothèse cette équation doit être réductible par (R_n), on doit avoir : $Typ(N_1^{\mathbf{t}}) = t_1, \ldots, t_n \to \tau$ avec $n \geq 1$. Donc, la condition alternative pour $Compat(T, \Pi_0(N_2^{\mathbf{t}}))$ dans les prémisses de $Compat(T, \Pi_0(M^{\mathbf{t}}))$ est vraie. Comme précédemment, puisque $ftv(N_1^{\mathbf{t}}) \cap ftv(N_2^{\mathbf{t}}) = \emptyset$, on a $ftv(N_1^{\mathbf{t}}) \cap T = \emptyset$, et donc $Compat(T, N_1^{\mathbf{t}})$ par le Lemme 8.24. Et, comme plus haut, $t \notin T$. En conséquence, nous sommes à même de dériver $Compat(T, \Pi_0(M^{\mathbf{t}}))$.

□

8.2.3 Relation de β-expansion

Afin de pouvoir "revenir en arrière" dans les réductions, nous définissons une relation de β-expansion directement sur les squelettes de preuve. Intuitivement, cette relation est l'inverse de la β-réduction.

Définition 8.27 *On définit une relation binaire $\xrightarrow{exp}$ entre squelettes de preuve, appelée* β-expansion. *Le premier et principal axiome de cette relation est le suivant :*

$$\dfrac{\dfrac{\dfrac{\Pi_1 \quad \cdots \quad \Pi_n}{\vdots}}{\dfrac{\Pi' \qquad \Pi'_1}{\vdash [M\{x \leftarrow N\}, N_1] : \overline{\tau}}\ (FORGET)}}{\dfrac{\vdots}{\dfrac{\Pi'_p}{\vdash [M\{x \leftarrow N\}, N_1, \ldots, N_p] : \overline{\tau}}\ (FORGET)}} \xrightarrow{exp}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash x : \overline{\tau_1} \quad \cdots \quad \vdash x : \overline{\tau_n}}{\vdots}}{\dfrac{\Pi}{\vdash \lambda x M : \overline{\tau_1}, \ldots, \overline{\tau_n} \to \overline{\tau}}\ (FUN)} \qquad \Pi'_1}{\vdash [\lambda x M, N_1] : \overline{\tau_1}, \ldots, \overline{\tau_n} \to \overline{\tau}}\ (FORGET)}}{\vdots}}{\dfrac{\dfrac{\Pi'_p}{\vdash [\lambda x M, N_1, \ldots, N_p] : \overline{\tau_1}, \ldots, \overline{\tau_n} \to \overline{\tau}}\ (FORGET) \qquad \Pi_1 \quad \cdots \quad \Pi_n}{\vdash [\lambda x M, N_1, \ldots, N_p] N : \overline{\tau}}\ (APPL)}$$

où :

- *n est le nombre d'axiomes $\vdash x : \overline{\sigma}$ apparaissant dans Π, et $n \geq 1$.*
- *$Typ(\Pi_i) = \overline{\tau_i}$ pour tout $1 \leq i \leq n$*
- *$Term(\Pi_i) = N$ pour tout $1 \leq i \leq n$*
- *Π' a la même structure que Π mis à part que les axiomes $\vdash x : \overline{\tau_i}$ sont remplacés par Π_i respectivement, et que les sous-termes de M sont remplacés par les sous-termes correspondants de $M\{x \leftarrow N\}$.*
- *Les environnements Γ, qui manquent dans la figure ci-dessus, sont "corrects" pour les deux squelettes, i.e. à chaque étape $\Gamma(y)$ est exactement la liste des types pour les axiomes y apparaissant libres dans l'arbre au-dessus, et ceci pour tout y. Grossièrement, les environnements de Π diffèrent de ceux de Π' en remplaçant les hypothèses provenant de Π_i par une seule hypothèse $x : \overline{\tau_i}$. Ces hypothèses sur x sont enlevées au moment de l'abstraction $\lambda x M$, et les hypothèses provenant de Π_i sont réintroduites au niveau du dernier noeud $(APPL)$. Par conséquent, on note que l'environnement à la racine est le même pour les deux arbres ; ceci explique pourquoi on utilise le même Γ dans les définitions inductives ci-dessous.*

Le second axiome ne concerne que les termes M pour lesquels $x \notin fv(M)$:

$$\dfrac{\dfrac{\dfrac{\Pi \quad \Pi'_1}{\vdash [M, N_1] : \overline{\tau}}\ (FORGET)}{\vdots}}{\dfrac{\dfrac{\Pi'_p}{\vdash [M, N_1, \ldots, N_p] : \overline{\tau}}\ (FORGET) \qquad \Pi_1}{\vdash [M, N_1, \ldots, N_p, N] : \overline{\tau}}\ (FORGET)} \xrightarrow{exp}$$

$$\dfrac{\dfrac{\dfrac{\Pi}{\vdash \lambda x M : \omega \to \overline{\tau}}\,(FUN) \qquad \Pi'_1}{\vdash [\lambda x M, N_1] : \omega \to \overline{\tau}}\,(FORGET)}{\begin{array}{c}\vdots \\ \dfrac{\dfrac{\Pi'_p}{\vdash [\lambda x M, N_1, \ldots, N_p] : \omega \to \overline{\tau}}\,(FORGET) \qquad \Pi_1}{\vdash [\lambda x M, N_1, \ldots, N_p] N : \overline{\tau}}\,(APPL)\end{array}}$$

où :

- *il n'y a pas d'axiome pour x dans Π.*
- *$Term(\Pi_1) = N$*
- *des conditions et des remarques similaires à celles du premier axiome sont vérifiées en ce qui concerne les environnements.*

Ensuite, pour que $\Pi \xrightarrow{exp} \Pi'$ soit vrai, l'un de ces deux axiomes doit être vérifié en un point unique du squelette. Autrement dit, nous ajoutons les dérivations inductives suivantes aux axiomes précédents. Si $\Pi \xrightarrow{exp} \Pi'$, alors :

$$\begin{array}{rcl}
\dfrac{\Pi \quad \Pi_1 \cdots \Pi_n}{\Gamma \vdash MN : \overline{\tau}}\,(APPL) & \xrightarrow{exp} & \dfrac{\Pi' \quad \Pi_1 \cdots \Pi_n}{\Gamma \vdash M'N : \overline{\tau}}\,(APPL) \\
\dfrac{\Pi_0 \quad \Pi_1 \cdots \Pi \cdots \Pi_n}{\Gamma \vdash MN : \overline{\tau}}\,(APPL) & \xrightarrow{exp} & \dfrac{\Pi_0 \quad \Pi_1 \cdots \Pi' \cdots \Pi_n}{\Gamma \vdash MN' : \overline{\tau}}\,(APPL) \\
\dfrac{\Pi}{\Gamma \vdash \lambda x M : \overline{\tau}}\,(FUN) & \xrightarrow{exp} & \dfrac{\Pi'}{\Gamma \vdash \lambda x M' : \overline{\tau}}\,(FUN) \\
\dfrac{\Pi \quad \Pi_2}{\Gamma \vdash [M, N] : \overline{\tau}}\,(FORGET) & \xrightarrow{exp} & \dfrac{\Pi' \quad \Pi_2}{\Gamma \vdash [M', N] : \overline{\tau}}\,(FORGET) \\
\dfrac{\Pi_1 \quad \Pi}{\Gamma \vdash [M, N] : \overline{\tau}}\,(FORGET) & \xrightarrow{exp} & \dfrac{\Pi_1 \quad \Pi'}{\Gamma \vdash [M, N'] : \overline{\tau}}\,(FORGET)
\end{array}$$

La β-expansion préserve la validité de l'arbre de typage...

Lemme 8.28 *Soient Π et Π' deux squelettes de preuve tels que $\Pi \xrightarrow{exp} \Pi'$. Si Π est un arbre de typage valide, alors Π' est aussi un arbre de typage valide.*

Preuve: Par induction sur la dérivation de $\Pi \xrightarrow{exp} \Pi'$ (on utilisera le fait que $\Pi \xrightarrow{exp} \Pi'$ implique $Typ(\Pi) = Typ(\Pi')$). □

Lemme 8.29 *Si $\Pi \xrightarrow{exp} \Pi'$, alors $ftv(\Pi) = ftv(\Pi')$.*

Preuve: Par induction sur $\Pi \xrightarrow{exp} \Pi'$. □

... et la β-expansion préserve la compatibilité :

Lemme 8.30 *Si $Compat(T, \Pi)$ et $\Pi \xrightarrow{exp} \Pi'$, alors $Compat(T, \Pi')$.*

Preuve: Par induction sur la dérivation de $\Pi \xrightarrow{exp} \Pi'$.

– Considérons tout d'abord le premier axiome :

$$\dfrac{\dfrac{\dfrac{\Pi_1 \quad \cdots \quad \Pi_n}{\vdots}}{\dfrac{\Pi' \qquad \Pi'_1}{\vdash [M\{x \leftarrow N\}, N_1] : \overline{\tau}}\,(FORGET)}}{\dfrac{\dfrac{\vdots}{\Pi'_p}}{\vdash [M\{x \leftarrow N\}, N_1, \ldots, N_p] : \overline{\tau}}\,(FORGET)} \xrightarrow{exp}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash x : \overline{\tau_1} \quad \cdots \quad \vdash x : \overline{\tau_n}}{\vdots}}{\dfrac{\Pi}{\vdash \lambda x M : \overline{\tau_1}, \ldots, \overline{\tau_n} \to \overline{\tau}}\,(FUN)} \qquad \Pi'_1}{\vdash [\lambda x M, N_1] : \overline{\tau_1}, \ldots, \overline{\tau_n} \to \overline{\tau}}\,(FORGET)}}{\dfrac{\dfrac{\dfrac{\vdots}{\Pi'_p}}{\vdash [\lambda x M, N_1, \ldots, N_p] : \overline{\tau_1}, \ldots, \overline{\tau_n} \to \overline{\tau}}\,(FORGET) \qquad \Pi_1 \quad \cdots \quad \Pi_n}{\vdash [\lambda x M, N_1, \ldots, N_p] N : \overline{\tau}}\,(APPL)}$$

Puisque T est compatible avec l'arbre de gauche, nous savons que les affirmations suivantes sont vraies : $Compat(T, \Pi')$, $Compat(T, \Pi'_i)$ pour tout $1 \leq i \leq p$, et $rgt(\overline{\tau}) \notin T$. Afin de dériver que T est compatible avec l'arbre de droite, nous devons vérifier quelques propriétés. Premièrement, notons que $rgt(\overline{\tau_1}, \ldots, \overline{\tau_n} \to \overline{\tau}) = rgt(\overline{\tau})$, de telle sorte que les conditions requises sur T pour les noeuds $(FORGET)$ seront toutes satisfaites.
Afin de construire une dérivation pour le noeud $(APPL)$, puisque nous savons que le type de gauche possède la bonne structure, il est suffisant de montrer que pour tout $1 \leq i \leq n$, on a soit $ftv(\Pi_i) \subseteq T$ soit $Compat(T, \Pi_i)$. Considérons maintenant le fait que Π' contient Π_i en tant que sous-arbre. A cause de la définition inductive de $Compat$, deux cas peuvent se présenter dans la dérivation de $Compat(T, \Pi')$. Soit on remonte jusqu'à rencontrer Π_i et alors $Compat(T, \Pi_i)$ est vraie. Soit on s'arrête à un noeud $(APPL)$ quelque part en cours de route, en utilisant la condition alternative pour la branche de droite Π'' qui contient Π_i. Dans ce cas, $ftv(\Pi'') \subseteq T$, et puisque Π_i est un sous-arbre de Π'', $ftv(\Pi_i) \subseteq T$ est vrai.
Finalement, nous devons dériver $Compat(T, \Pi)$. Pour ceci, on suit exactement les mêmes dérivations que pour $Compat(T, \Pi')$, en montrant que remplacer les Π_i par des axiomes $\vdash x : \overline{\tau_i}$ ne change pas les conditions. On rappelle que $Typ(\Pi_i) = \overline{\tau_i}$, de telle sorte que $ftv(\vdash x : \overline{\tau_i}) \subseteq ftv(\Pi_i)$. Pour les sous-arbres Π'' qui auraient satisfait la condition alternative $ftv(\Pi'') \subseteq T$, le nouveau sous-arbre correspondant aurait un ensemble de variables de type plus petit, donc toujours inclus dans T. Et, si on arrive à un point où $Compat(T, \Pi_i)$ est utilisé, cette dernière affirmation implique que $rgt(Typ(\Pi_i)) \notin T$, autrement dit $rgt(\overline{\tau_i}) \notin T$ et on peut sans danger la remplacer par l'axiome $Compat(T, \vdash x : \overline{\tau_i})$ dans la preuve.

– Considérons maintenant le second axiome :

$$\dfrac{\dfrac{\Pi \quad \Pi'_1}{\vdash [M, N_1] : \overline{\tau}}\,(FORGET)}{\vdots}$$
$$\dfrac{\dfrac{\Pi'_p}{\vdash [M, N_1, \ldots, N_p] : \overline{\tau}}\,(FORGET) \qquad \Pi_1}{\vdash [M, N_1, \ldots, N_p, N] : \overline{\tau}}\,(FORGET) \qquad \xrightarrow{exp}$$

$$\dfrac{\dfrac{\dfrac{\Pi}{\vdash \lambda x M : \omega \to \overline{\tau}}\,(FUN) \qquad \Pi'_1}{\vdash [\lambda x M, N_1] : \omega \to \overline{\tau}}\,(FORGET)}{\vdots}$$
$$\dfrac{\dfrac{\Pi'_p}{\vdash [\lambda x M, N_1, \ldots, N_p] : \omega \to \overline{\tau}}\,(FORGET) \qquad \Pi_1}{\vdash [\lambda x M, N_1, \ldots, N_p] N : \overline{\tau}}\,(APPL)$$

Puisque T est compatible avec l'arbre de gauche, on en déduit que les prédicats suivant doivent être vrais : $Compat(T, \Pi)$, $Compat(T, \Pi_1)$, $Compat(T, \Pi'_i)$ pour $1 \leq i \leq p$, et $rgt(\overline{\tau}) \notin T$. Avec ces affirmations, et comme $rgt(\omega \to \overline{\tau}) = rgt(\overline{\tau})$, reconstruire la preuve que T est compatible avec l'arbre de droite est une tâche facile.

– Nous ne détaillerons qu'un seul cas inductif qui mérite une attention spécifique. Supposons que

$$\dfrac{\Pi_0 \quad \Pi_1 \cdots \Pi \cdots \Pi_n}{\Gamma \vdash MN : \overline{\tau}}(\text{APPL}) \xrightarrow{exp} \dfrac{\Pi_0 \quad \Pi_1 \cdots \Pi' \cdots \Pi_n}{\Gamma \vdash MN' : \overline{\tau}}(\text{APPL})$$

dérive de $\Pi \xrightarrow{exp} \Pi'$, et que T est compatible avec l'arbre de gauche. Considérons maintenant quelles alternatives ont été utilisées pour Π. Si c'est $Compat(T, \Pi)$, alors par hypothèse d'induction, on a $Compat(T, \Pi')$ et la compatibilité de T avec l'arbre de droite en découle aisément. Sinon, si la condition alternative $ftv(\Pi) \subseteq T$ a été utilisée (tandis que le type à la racine de Π_0 a la bonne forme), comme $ftv(\Pi') = ftv(\Pi)$ par le Lemme 8.29, on a également $ftv(\Pi') \subseteq T$. La conclusion s'ensuit facilement.

– Pour tous les autres cas, la preuve suit un argument inductif simple, comme nous l'avons fait pour le cas précédent.

□

Enfin, la relation de β-expansion est également préservée par substitution. On notera qu'en ce qui concerne la duplication, le résultat n'est vrai que pour des territoires compatibles.

Lemme 8.31 *On suppose que* $\Pi \xrightarrow{exp} \Pi'$. *On a :*

– $\langle \Pi \rangle^i \xrightarrow{exp} \langle \Pi' \rangle^i$ *pour tout* i
– $\overline{S}(\Pi) \xrightarrow{exp} \overline{S}(\Pi')$ *pour tout* $\overline{S} = \{t \mapsto \overline{\tau}\}$
– $\overline{S}(\Pi) \xrightarrow{exp}{}^{+} \overline{S}(\Pi')$ *pour tout* $\overline{S} = D(n, T)$ *tel que* $Compat(T, \Pi)$

Preuve:

- Puisque $\langle \Pi \rangle^i$ est exactement le même squelette que Π dans lequel toutes les variables de type t ont été remplacées par $\langle t \rangle^i$ partout, il est facile de vérifier que toutes les conditions sont encore satisfaites, et que $\langle \Pi \rangle^i \xrightarrow{exp} \langle \Pi' \rangle^i$.
- Comme précédemment, si l'on regarde la définition de $\overline{S}$, on peut voir $\overline{S}(\Pi)$ n'est rien d'autre que Π dans lequel toutes les occurrences de t ont été remplacées par $\overline{\tau}$ partout. Sa structure générale n'est pas modifiée. Puisque la même chose s'applique à Π', on a $\overline{S}(\Pi) \xrightarrow{exp} \overline{S}(\Pi')$.
- On raisonne par induction sur la dérivation de $\Pi \xrightarrow{exp} \Pi'$.
  - Considérons le premier axiome :

$$\dfrac{\dfrac{\dfrac{\dfrac{\Pi_1 \quad \cdots \quad \Pi_n}{\vdots}}{\Pi'} \qquad \Pi'_1}{\vdash [M\{x \leftarrow N\}, N_1] : \overline{\tau}}\ (FORGET)}{\vdots}}{\dfrac{\Pi'_p}{\vdash [M\{x \leftarrow N\}, N_1, \ldots, N_p] : \overline{\tau}}\ (FORGET)} \xrightarrow{exp}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash x : \overline{\tau_1} \quad \cdots \quad \vdash x : \overline{\tau_n}}{\vdots}}{\Pi}}{\vdash \lambda x M : \overline{\tau_1}, \ldots, \overline{\tau_n} \to \overline{\tau}}\ (FUN) \qquad \Pi'_1}{\vdash [\lambda x M, N_1] : \overline{\tau_1}, \ldots, \overline{\tau_n} \to \overline{\tau}}\ (FORGET)}{\vdots}$$

$$\dfrac{\dfrac{\dfrac{\Pi'_p}{\vdash [\lambda x M, N_1, \ldots, N_p] : \overline{\tau_1}, \ldots, \overline{\tau_n} \to \overline{\tau}}\ (FORGET) \qquad \Pi_1 \quad \cdots \quad \Pi_n}{\vdash [\lambda x M, N_1, \ldots, N_p] N : \overline{\tau}}\ (APPL)}$$

Afin d'aboutir à la conclusion désirée, il faut principalement montrer que les duplications que nous pourrions faire sur Π_i pour le noeud racine $(APPL)$ correspondent exactement aux duplications des sous-arbres de Π contenant l'axiome $\vdash x : \overline{\tau_i}$ et aux duplications des types $\overline{\tau_i}$ dans $\overline{\tau_1}, \ldots, \overline{\tau_n} \to \overline{\tau}$ (grossièrement).
Plus précisément, considérons un i en particulier. A partir des hypothèses, on sait que $Compat(T, \Pi')$ est vrai. On procède maintenant par induction sur Π', en suivant le chemin menant au sous-arbre Π_i, avec l'invariant que T est toujours compatible avec le sous-arbre considéré. Etant donnée la définition de $Compat$, il ne s'agit que d'une simple étape inductive dans tous les cas, sauf un : lorsqu'on tombe sur un noeud $(APPL)$ pour lequel Π_i est un sous-arbre de la branche de droite.
On considère maintenant ce cas spécifique, en appelant Π'' la branche de droite contenant Π_i. Supposons que $ftv(\Pi'') \subseteq T$. Dans ce cas, la branche Π'' est dupliquée au moment de calculer $\overline{S}(\Pi)$, et donc Π_i est aussi dupliqué. Ensuite, puisque $ftv(\Pi_i) \subseteq ftv(\Pi'') \subseteq T$, Π_i sera aussi dupliqué pour le noeud racine $(APPL)$ dans la conclusion. De plus, $ftv(\overline{\tau_i}) \subseteq ftv(\Pi_i) \subseteq T$ de telle sorte que le sous-arbre correspondant de Π qui contient l'axiome $\vdash x : \overline{\tau_i}$ sera aussi dupliqué dans $\overline{S}(\Pi)$; et les types $\overline{\tau_i}$ apparaissant dans les noeuds $(FORGET)$ seront aussi dupliqués.

Supposons maintenant que $ftv(\Pi'') \not\subseteq T$ (Π'' n'est pas dupliqué). Dans ce cas, on doit avoir $Compat(T, \Pi'')$ et l'induction peut être continuée sur Π''. De plus, ceci implique que $rgt(Typ(\Pi'')) \notin T$. Comme le sous-arbre correspondant dans Π a la même structure que $Typ(\Pi'')$, il ne peut pas non plus être dupliqué (en d'autres termes, les points de duplication dans Π' et dans Π sont exactement les mêmes, même si l'ensemble des variables libres des sous-arbres de Π peut être plus petit que celui de Π').
Finalement, on peut aboutir à un point où l'on considère Π_i lui-même, avec la propriété que $Compat(T, \Pi_i)$. Dans ce cas, $rgt(Typ(\Pi_i)) \notin T$. Ceci implique que $ftv(\Pi_i) \not\subseteq T$, donc Π_i n'est pas dupliqué mais simplement remplacé par $\overline{S}(\Pi_i)$ dans les deux arbres. En outre, $rgt(\overline{\tau_i}) = rgt(Typ(\Pi_i)) \notin T$. Par conséquent, les axiomes $\vdash x : \overline{\tau_i}$ ne sont pas dupliqués non plus, tout comme les types $\overline{\tau_i}$ dans $(FORGET)$.
Pour en finir cette partie, les prémisses requises ci-dessus sont toutes satisfaites pour tous les i. Les sous-arbres Π'_j sont remplacés par $\overline{S}(\Pi'_j)$ dans les deux arbres, et la même chose s'applique aux autres branches de Π' et Π. Finalement, on obtient la dérivation désirée.

– On considère le second axiome :

$$\dfrac{\dfrac{\dfrac{\Pi \quad \Pi'_1}{\vdash [M, N_1] : \overline{\tau}}\,(FORGET)}{\vdots}\quad \Pi'_p}{\vdash [M, N_1, \ldots, N_p] : \overline{\tau}}\,(FORGET) \quad \Pi_1}{\vdash [M, N_1, \ldots, N_p, N] : \overline{\tau}}\,(FORGET) \qquad \xrightarrow{exp}$$

$$\dfrac{\dfrac{\dfrac{\Pi}{\vdash \lambda x M : \omega \to \overline{\tau}}\,(FUN) \quad \Pi'_1}{\vdash [\lambda x M, N_1] : \omega \to \overline{\tau}}\,(FORGET)}{\vdots}\quad \Pi'_p}{\vdash [\lambda x M, N_1, \ldots, N_p] : \omega \to \overline{\tau}}\,(FORGET) \quad \Pi_1}{\vdash [\lambda x M, N_1, \ldots, N_p]N : \overline{\tau}}\,(APPL)$$

Puisque T est compatible avec l'arbre de gauche, on a en particulier $Compat(T, \Pi_1)$. Ceci implique que $rgt(Typ(\Pi_1)) \notin T$ et $ftv(\Pi_1) \not\subseteq T$. Donc, Π_1 ne peut pas être dupliqué. En conclusion, il suffit de remplacer Π par $\overline{S}(\Pi)$, Π'_j par $\overline{S}(\Pi'_j)$, Π_1 par $\overline{S}(\Pi_1)$ et $\overline{\tau}$ par $\overline{S}(\overline{\tau})$ dans les deux arbres pour obtenir la dérivation désirée et qui demeure valide.

– Nous ne détaillerons qu'un seul cas inductif qui requiert une attention particulière. Supposons que

$$\dfrac{\Pi_0 \quad \Pi_1 \cdots \Pi \cdots \Pi_n}{\Gamma \vdash MN : \overline{\tau}}(\text{APPL}) \xrightarrow{exp} \dfrac{\Pi_0 \quad \Pi_1 \cdots \Pi' \cdots \Pi_n}{\Gamma \vdash MN' : \overline{\tau}}(\text{APPL})$$

dérive de $\Pi \xrightarrow{exp} \Pi'$. Par le Lemme 8.29, $ftv(\Pi) = ftv(\Pi')$.
Si $ftv(\Pi) \not\subseteq T$, il n'y a rien à faire. Aussi bien Π que Π' ne sont pas dupliqués par $\overline{S}$, et on doit aussi avoir $Compat(T, \Pi)$. On procède juste par induction et on dérive $\overline{S}(\Pi) \xrightarrow{exp}{}^{+} \overline{S}(\Pi')$. Ceci étant donné, on aboutit finalement à la conclusion :

$$\dfrac{\overline{S}(\Pi_0) \quad D'(\Pi_1) \cdots \overline{S}(\Pi) \cdots D'(\Pi_n)}{\overline{S}(\Gamma) \vdash MN : \overline{S}(\overline{\tau})}(\text{APPL}) \xrightarrow{exp}{}^{+} \dfrac{\overline{S}(\Pi_0) \quad D'(\Pi_1) \cdots \overline{S}(\Pi') \cdots D'(\Pi_n)}{\overline{S}(\Gamma) \vdash MN' : \overline{S}(\overline{\tau})}(\text{APPL})$$

où $D'(\Pi) = dupl_n(\Pi)$ si $ftv(\Pi) \subseteq T$, ou $\overline{S}(\Pi)$ sinon.
Si $ftv(\Pi) \subseteq T$, alors à la fois Π et Π' doivent être dupliqués. D'après le premier résultat ci-dessus, on a $\langle\Pi\rangle^i \xrightarrow{exp} \langle\Pi'\rangle^i$ pour tout $1 \leq i \leq n$. Il est alors facile de dériver :

$$\frac{\overline{S}(\Pi_0) \quad D'(\Pi_1)\cdots dupl_n(\Pi)\cdots D'(\Pi_n)}{\overline{S}(\Gamma) \vdash MN : \overline{S}(\overline{\tau})}(\text{APPL}) \xrightarrow{exp}{}^{+} \frac{\overline{S}(\Pi_0) \quad D'(\Pi_1)\cdots dupl_n(\Pi')\cdots D'(\Pi_n)}{\overline{S}(\Gamma) \vdash MN' : \overline{S}(\overline{\tau})}(\text{APPL})$$

(c'est le seul cas pour lequel une étape de β-expansion peut être remplacée par n expansions, d'où le "+" dans $\xrightarrow{exp}{}^{+}$.)

– Tous les autres cas inductifs sont triviaux et se traitent comme le premier cas ci-dessus.

□

Il reste à montrer la propriété d'expansion proprement dite : lorsqu'un terme est réduit, la β-expansion appliquée au squelette de preuve nous permet de retrouver un squelette de preuve pour le terme de départ.

Lemme 8.32 *Soit $M \longrightarrow_{\mathcal{K}} N$ une $\mathcal{K}$-réduction. Par le Lemme 8.11, on sait qu'il existe $\mathcal{E}$ et Π tels que $Syst_0(M) \longrightarrow (\mathcal{E}, \Pi)$ par l'une des règles (R_n) ou (R_0). Alors, $\Pi_0(M_0^{\mathbf{t}}(N)) \xrightarrow{exp}_{\equiv} \Pi$.*

Preuve: On part du résultat du Lemme 8.11, en instanciant le squelette de preuve Π par $\Pi_0(M^{\mathbf{t}})$. Ce lemme affirme maintenant : "Soit $M \longrightarrow_{\mathcal{K}} N$ une $\mathcal{K}$-réduction. Soient $M^{\mathbf{t}} = M_0^{\mathbf{t}}(M)$ et $N^{\mathbf{t}} = M_0^{\mathbf{t}}(N)$. Alors, $(\mathcal{E}_0(M^{\mathbf{t}}), \Pi_0(M^{\mathbf{t}})) \longrightarrow (S(\mathcal{E}), \overline{S}(\Pi_0(M^{\mathbf{t}})))$ par l'une des règles (R_n) ou (R_0)." Ce que nous voulons montrer est $\Pi_0(N^{\mathbf{t}}) \xrightarrow{exp}_{\equiv} \overline{S}(\Pi_0(M^{\mathbf{t}}))$. On procède par induction sur la dérivation de $M \longrightarrow_{\mathcal{K}} N$. Afin de ne pas s'encombrer avec les renommages, on "choisira" les variables de $M^{\mathbf{t}}$ et $N^{\mathbf{t}}$ judicieusement, même si, pour être strict, ils devraient être choisis tous différents puis égalisés par renommages par la suite.

– On commence avec la règle principale pour un redex :

$$P = [\lambda x M, N_1, \ldots, N_p]\ T \longrightarrow_{\mathcal{K}} Q = [M\{x \leftarrow T\}, N_1, \ldots, N_p]$$

avec $x \in fv(M)$. On convertit P en sa version annotée :

$$P^{\mathbf{t}} = ([\lambda x M^{\mathbf{t}}, N_1^{\mathbf{t}}, \ldots, N_p^{\mathbf{t}}]\ T^{\mathbf{t}}) : t \qquad Q^{\mathbf{t}} = [(M\{x \leftarrow T\})^{\mathbf{t}}, N_1^{\mathbf{t}}, \ldots, N_p^{\mathbf{t}}]$$

On écrira $VarTyp(x, M^{\mathbf{t}}) = \{t_1, \ldots, t_n\}$, $\tau = Typ(M^{\mathbf{t}})$ et $\sigma = Typ(T^{\mathbf{t}})$. Alors, à partir de la preuve du Lemme 8.11, nous savons déjà que $Typ([\lambda x M^{\mathbf{t}}, N_1^{\mathbf{t}}, \ldots, N_p^{\mathbf{t}}]) = t_1, \ldots, t_n \to \tau$, que l'équation à réduire est

$$\sigma \to t \ \stackrel{\perp}{=}\ t_1, \ldots, t_n \to \tau \ \ [ftv(T^{\mathbf{t}})]$$

et que la substitution correspondante est $\overline{S} = \{t_i \mapsto \langle\overline{\sigma}\rangle^i\}_{1 \leq i \leq n} :: \{t \mapsto \overline{\tau}\} :: D(n, ftv(T^{\mathbf{t}}))$. Nous sommes maintenant prêts à calculer la valeur de $\Pi_0(P^{\mathbf{t}})$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash x : t_1 \quad \cdots \quad \vdash x : t_n}{\vdots}}{\Pi_0(M^{\mathbf{t}})}}{\vdash \lambda x M : t_1, \ldots, t_n \to \overline{\tau}}(FUN) \qquad \Pi_0(N_1^{\mathbf{t}})}{\vdash [\lambda x M, N_1] : t_1, \ldots, t_n \to \overline{\tau}}(FORGET)}{\vdots} \qquad \dfrac{\Pi_0(N_p^{\mathbf{t}})}{\vdash [\lambda x M, N_1, \ldots, N_p] : t_1, \ldots, t_n \to \overline{\tau}}(FORGET) \qquad \Pi_0(T^{\mathbf{t}})}{\vdash [\lambda x M, N_1, \ldots, N_p]\ T : t}(APPL)$$

D'après le Lemme 8.4, on a $ftv(T^{\mathbf{t}}) \cap ftv(X^{\mathbf{t}})$ pour tous les autres termes annotés $X^{\mathbf{t}}$ dans le squelette, et le Lemme 8.25 nous dit que $ftv(\Pi_0(X^{\mathbf{t}})) = ftv(X^{\mathbf{t}})$. Par conséquent, la duplication $D(n, ftv(T^{\mathbf{t}}))$ ne dupliquera que le sous-arbre $\Pi_0(T^{\mathbf{t}})$ et aucun autre sous-arbre ou même type dans la branche gauche du noeud $(APPL)$ ci-dessus. De plus, puisque t est fraîche, elle n'apparaît qu'au noeud racine, de telle sorte que le squelette $(\{t \mapsto \overline{\tau}\} :: D(n, ftv(T^{\mathbf{t}})))(\Pi_0(P^{\mathbf{t}}))$ est le suivant :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash x : t_1 \quad \cdots \quad \vdash x : t_n \\ \vdots \\ \Pi_0(M^{\mathbf{t}})}{\vdash \lambda x M : t_1, \ldots, t_n \to \overline{\tau}}\,(FUN) \qquad \Pi_0(N_1^{\mathbf{t}})}{\vdash [\lambda x M, N_1] : t_1, \ldots, t_n \to \overline{\tau}}\,(FORGET) \\ \vdots \\ \Pi_0(N_p^{\mathbf{t}})}{\vdash [\lambda x M, N_1, \ldots, N_p] : t_1, \ldots, t_n \to \overline{\tau}}\,(FORGET) \qquad \langle \Pi_0(T^{\mathbf{t}}) \rangle^1 \quad \cdots \quad \langle \Pi_0(T^{\mathbf{t}}) \rangle^n}{\vdash [\lambda x M, N_1, \ldots, N_p]\ T : \overline{\tau}}\,(APPL)$$

Il reste encore à appliquer les substitutions $\overline{S'} = \{t_i \mapsto \langle \overline{\sigma} \rangle^i\}$. Pour des raisons similaires, les variables t_i ne peuvent apparaître que dans $\Pi_0(M^{\mathbf{t}})$ et dans les noeuds sous lui (et éventuellement dans $\overline{\tau}$). Finalement, la valeur de $\overline{S}(\Pi_0(P^{\mathbf{t}}))$ est la suivante :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash x : \langle \overline{\sigma} \rangle^1 \quad \cdots \quad \vdash x : \langle \overline{\sigma} \rangle^n \\ \vdots \\ \overline{S'}(\Pi_0(M^{\mathbf{t}}))}{\vdash \lambda x M : \langle \overline{\sigma} \rangle^1, \ldots, \langle \overline{\sigma} \rangle^n \to \overline{S'}(\overline{\tau})}\,(FUN) \qquad \Pi_0(N_1^{\mathbf{t}})}{\vdash [\lambda x M, N_1] : \langle \overline{\sigma} \rangle^1, \ldots, \langle \overline{\sigma} \rangle^n \to \overline{S'}(\overline{\tau})}\,(FORGET) \\ \vdots \\ \Pi_0(N_p^{\mathbf{t}})}{\vdash [\lambda x M, N_1, \ldots, N_p] : \langle \overline{\sigma} \rangle^1, \ldots, \langle \overline{\sigma} \rangle^n \to \overline{S'}(\overline{\tau})}\,(FORGET) \qquad \langle \Pi_0(T^{\mathbf{t}}) \rangle^1 \quad \cdots \quad \langle \Pi_0(T^{\mathbf{t}}) \rangle^n}{\vdash [\lambda x M, N_1, \ldots, N_p]\ T : \overline{S'}(\overline{\tau})}\,(APPL)$$

Par ailleurs, considérons maintenant $Q^{\mathbf{t}} = [(M\{x \leftarrow T\})^{\mathbf{t}}, N_1^{\mathbf{t}}, \ldots, N_p^{\mathbf{t}}]$. Le sous-terme annoté $(M\{x \leftarrow T\})^{\mathbf{t}}$ est en fait $M^{\mathbf{t}}$ dans lequel les $x : t_i$ ont été remplacés par différentes instances de $T^{\mathbf{t}}$. Via le renommage, on peut choisir d'utiliser $\langle T^{\mathbf{t}} \rangle^i$ pour ces instances. Donc, $Q^{\mathbf{t}} = [M^{\mathbf{t}}\{x \leftarrow \langle T^{\mathbf{t}} \rangle^i\}, N_1^{\mathbf{t}}, \ldots, N_p^{\mathbf{t}}]$. Considérons maintenant $\Pi_0(M^{\mathbf{t}}\{x \leftarrow \langle T^{\mathbf{t}} \rangle^i\})$. C'est le même squelette que $\Pi_0(M^{\mathbf{t}})$ dans lequel les axiomes sur x ont été remplacés par $\Pi_0(\langle T^{\mathbf{t}} \rangle^i) = \langle \Pi_0(T^{\mathbf{t}}) \rangle^i$, et les types t_i par $Typ(\langle \Pi_0(T^{\mathbf{t}}) \rangle^i) = \langle \overline{\sigma} \rangle^i$ en conséquence. Donc, il est en fait identique au remplacement des axiomes sur x par $\langle \Pi_0(T^{\mathbf{t}}) \rangle^i$ dans $\overline{S'}(\Pi_0(M^{\mathbf{t}}))$. Notons Π'' ce sous-arbre. Il reste à mentionner que $Typ(\overline{S'}(\Pi_0(M^{\mathbf{t}}))) = \overline{S'}(Typ(\Pi_0(M^{\mathbf{t}}))) =$

$\overline{S'}(\overline{Typ(M^{\mathbf{t}})}) = \overline{S'}(\overline{\tau})$, et on peut écrire le squelette pour $\Pi_0(Q^{\mathbf{t}})$:

$$\dfrac{\dfrac{\dfrac{\langle\Pi_0(T^{\mathbf{t}})\rangle^1 \quad \cdots \quad \langle\Pi_0(T^{\mathbf{t}})\rangle^n}{\vdots}}{\dfrac{\Pi'' \qquad\qquad \Pi_0(N_1^{\mathbf{t}})}{\vdash [M\{x \leftarrow N\}, N_1] : \overline{S'}(\overline{\tau})}\ (FORGET)}}{\vdots}}{\dfrac{\qquad\qquad\qquad\qquad \Pi_0(N_p^{\mathbf{t}})}{\vdash [M\{x \leftarrow N\}, N_1, \ldots, N_p] : \overline{S'}(\overline{\tau})}\ (FORGET)}$$

Finalement, c'est une tâche simple de vérifier les conditions pour la β-expansion, et on obtient : $\Pi_0(Q^{\mathbf{t}}) \xrightarrow{exp} \overline{S}(\Pi_0(P^{\mathbf{t}}))$.

– Considérons maintenant l'autre règle pour un redex :

$$P = [\lambda x M, N_1, \ldots, N_p]\ T \longrightarrow_{\mathcal{K}} Q = [M, N_1, \ldots, N_p, T]$$

avec $x \notin fv(M)$. On convertit ces termes en leur version annotée :

$$P^{\mathbf{t}} = ([\lambda x M^{\mathbf{t}}, N_1^{\mathbf{t}}, \ldots, N_p^{\mathbf{t}}]\ T^{\mathbf{t}}) : t \qquad\qquad Q^{\mathbf{t}} = [M^{\mathbf{t}}, N_1^{\mathbf{t}}, \ldots, N_p^{\mathbf{t}}, T^{\mathbf{t}}]$$

Notons $\tau = Typ(M^{\mathbf{t}})$. A partir de la preuve du Lemme 8.11, nous savons déjà qu'on a $Typ([\lambda x M^{\mathbf{t}}, N_1^{\mathbf{t}}, \ldots, N_p^{\mathbf{t}}]) = \omega \to \tau$, que l'équation réduite est :

$$Typ(T^{\mathbf{t}}) \to t \ \doteq\ \omega \to \tau \ \ [ftv(T^{\mathbf{t}})]$$

et la substitution correspondante $\overline{S} = \{t \mapsto \overline{\tau}\}$. Par conséquent, on peut écrire le squelette $\Pi_0(P^{\mathbf{t}})$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Pi_0(M^{\mathbf{t}})}{\vdash \lambda x M : \omega \to \overline{\tau}}\ (FUN) \qquad \Pi_0(N_1^{\mathbf{t}})}{\vdash [\lambda x M, N_1] : \omega \to \overline{\tau}}\ (FORGET)}{\vdots}}{\dfrac{\dfrac{\qquad\qquad\qquad\qquad \Pi_0(N_p^{\mathbf{t}})}{\vdash [\lambda x M, N_1, \ldots, N_p] : \omega \to \overline{\tau}}\ (FORGET) \qquad \Pi_0(T^{\mathbf{t}})}{\vdash [\lambda x M, N_1, \ldots, N_p]\ T : t}\ (APPL)}$$

Puisque t est fraîche, elle n'apparaît nulle part ailleurs qu'à la racine, et on obtient la valeur pour $\overline{S}(\Pi_0(P^{\mathbf{t}}))$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Pi_0(M^{\mathbf{t}})}{\vdash \lambda x M : \omega \to \overline{\tau}}\ (FUN) \qquad \Pi_0(N_1^{\mathbf{t}})}{\vdash [\lambda x M, N_1] : \omega \to \overline{\tau}}\ (FORGET)}{\vdots}}{\dfrac{\dfrac{\qquad\qquad\qquad\qquad \Pi_0(N_p^{\mathbf{t}})}{\vdash [\lambda x M, N_1, \ldots, N_p] : \omega \to \overline{\tau}}\ (FORGET) \qquad \Pi_0(T^{\mathbf{t}})}{\vdash [\lambda x M, N_1, \ldots, N_p]\ T : \overline{\tau}}\ (APPL)}$$

Et donc, le squelette pour $\Pi_0(Q^{\mathbf{t}})$ est :

$$\dfrac{\dfrac{\dfrac{\Pi_0(M^{\mathbf{t}}) \quad \Pi_0(N_1^{\mathbf{t}})}{\vdash [M, N_1] : \overline{\tau}}\ (FORGET)}{\vdots}}{\dfrac{\dfrac{\Pi_0(N_p^{\mathbf{t}})}{\vdash [M, N_1, \ldots, N_p] : \overline{\tau}}\ (FORGET) \qquad \Pi_0(T^{\mathbf{t}})}{\vdash [M, N_1, \ldots, N_p, T] : \overline{\tau}}\ (FORGET)}$$

et c'est une tâche facile de vérifier que $\Pi_0(Q^{\mathbf{t}}) \xrightarrow{exp} \overline{S}(\Pi_0(P^{\mathbf{t}}))$.

– Supposons que $MN \longrightarrow_{\mathcal{K}} M'N$ dérive de $M \longrightarrow_{\mathcal{K}} M'$. Avec des notations évidentes (et modulo les renommages adéquats), on note :

$$P^{\mathbf{t}} = M_0^{\mathbf{t}}(MN) = (M^{\mathbf{t}} N^{\mathbf{t}}) : t \qquad Q^{\mathbf{t}} = M_0^{\mathbf{t}}(M'N) = (M^{\mathbf{t}'} N^{\mathbf{t}}) : t$$

Par hypothèse d'induction, on sait que $M^{\mathbf{t}}$ génère une réduction et une substitution $\overline{S}$, et qu'on a : $\Pi_0(M^{\mathbf{t}'}) \xrightarrow{exp} \overline{S}(\Pi_0(M^{\mathbf{t}}))$. Par conséquent, on obtient :

$$\Pi_0(Q^{\mathbf{t}}) = \dfrac{\Pi_0(M^{\mathbf{t}'}) \qquad \Pi_0(N^{\mathbf{t}})}{Env(Q^{\mathbf{t}}) \vdash M'N : t}(\text{APPL}) \xrightarrow{exp} \dfrac{\overline{S}(\Pi_0(M^{\mathbf{t}})) \qquad \Pi_0(N^{\mathbf{t}})}{Env(Q^{\mathbf{t}}) \vdash MN : t}(\text{APPL})$$

Par le Lemme 8.8, on a $dom(\overline{S}) \subseteq ftv(M^{\mathbf{t}})$. Puisque $ftv(M^{\mathbf{t}}) \cap ftv(N^{\mathbf{t}}) = \emptyset$, on peut conclure d'un côté que $ftv(\Pi_0(N^{\mathbf{t}})) \nsubseteq dom(\overline{S})$ de telle sorte que $\Pi_0(N^{\mathbf{t}})$ ne peut pas être dupliqué par $\overline{S}$, et de l'autre que $\overline{S}(\Pi_0(N^{\mathbf{t}})) = \Pi_0(N^{\mathbf{t}})$. Pour les mêmes raisons, $\overline{S}(t) = t$. En outre, l'une des conséquences du Lemme 8.11 était que $S(VarTyp(x, P^{\mathbf{t}})) = VarTyp(x, Q^{\mathbf{t}})$ pour tout x. On peut donc conclure que : $\overline{S}(Env(P^{\mathbf{t}})) = Env(Q^{\mathbf{t}})$. Finalement, on obtient :

$$\Pi_0(Q^{\mathbf{t}}) \xrightarrow{exp} \dfrac{\overline{S}(\Pi_0(M^{\mathbf{t}})) \qquad \overline{S}(\Pi_0(N^{\mathbf{t}}))}{\overline{S}(Env(P^{\mathbf{t}})) \vdash MN : \overline{S}(t)}(\text{APPL}) = \overline{S}(\Pi_0(P^{\mathbf{t}}))$$

– Supposons que $MN \longrightarrow_{\mathcal{K}} MN'$ dérive de $N \longrightarrow_{\mathcal{K}} N'$. On note :

$$P^{\mathbf{t}} = M_0^{\mathbf{t}}(MN) = (M^{\mathbf{t}} N^{\mathbf{t}}) : t \qquad Q^{\mathbf{t}} = M_0^{\mathbf{t}}(MN') = (M^{\mathbf{t}} N^{\mathbf{t}'}) : t$$

Par hypothèse d'induction, on sait que $N^{\mathbf{t}}$ génère une réduction et une substitution $\overline{S}$, et qu'on a : $\Pi_0(N^{\mathbf{t}'}) \xrightarrow{exp} \overline{S}(\Pi_0(N^{\mathbf{t}}))$. Par conséquent, on obtient :

$$\Pi_0(Q^{\mathbf{t}}) = \dfrac{\Pi_0(M^{\mathbf{t}}) \qquad \Pi_0(N^{\mathbf{t}'})}{Env(Q^{\mathbf{t}}) \vdash MN' : t}(\text{APPL}) \xrightarrow{exp} \dfrac{\Pi_0(M^{\mathbf{t}}) \qquad \overline{S}(\Pi_0(N^{\mathbf{t}}))}{Env(Q^{\mathbf{t}}) \vdash MN : t}(\text{APPL})$$

Comme dans le cas précédent, on a : $\overline{S}(\Pi_0(M^{\mathbf{t}})) = \Pi_0(M^{\mathbf{t}})$, $\overline{S}(Env(P^{\mathbf{t}})) = Env(Q^{\mathbf{t}})$ et $\overline{S}(t) = t$. Mais il faut encore justifier pourquoi $\Pi_0(N^{\mathbf{t}})$ ne peut pas être dupliqué par $\overline{S}$. On sait que la réduction a dû avoir lieu au niveau d'un noeud application dans $N^{\mathbf{t}}$. Ceci signifie que les variables de type dupliquées sont celles de la branche droite de ce noeud. Mais la branche gauche, qui fait toujours partie de $N^{\mathbf{t}}$, doit contenir au moins une autre variable de type. Et donc, on ne peut avoir $ftv(N^{\mathbf{t}}) \subseteq T$. Finalement, on obtient :

$$\Pi_0(Q^{\mathbf{t}}) \xrightarrow{exp} \dfrac{\overline{S}(\Pi_0(M^{\mathbf{t}})) \qquad \overline{S}(\Pi_0(N^{\mathbf{t}}))}{\overline{S}(Env(P^{\mathbf{t}})) \vdash MN : \overline{S}(t)}(\text{APPL}) = \overline{S}(\Pi_0(P^{\mathbf{t}}))$$

– Supposons que $\lambda x M \longrightarrow_{\mathcal{K}} \lambda x N$ dérive de $M \longrightarrow_{\mathcal{K}} N$. On note $P^{\mathbf{t}} = \lambda x M^{\mathbf{t}}$ et $Q^{\mathbf{t}} = \lambda x N^{\mathbf{t}}$. Par hypothèse d'induction, on sait que $M^{\mathbf{t}}$ génère une réduction et une substitution $\overline{S}$, et qu'on a : $\Pi_0(N^{\mathbf{t}}) \xrightarrow{exp} \overline{S}(\Pi_0(M^{\mathbf{t}}))$. Par conséquent, on obtient :

$$\Pi_0(Q^{\mathbf{t}}) = \frac{\Pi_0(N^{\mathbf{t}})}{Env(Q^{\mathbf{t}}) \vdash \lambda x N : \overline{Typ(Q^{\mathbf{t}})}}(\text{FUN}) \xrightarrow{exp} \frac{\overline{S}(\Pi_0(M^{\mathbf{t}}))}{Env(Q^{\mathbf{t}}) \vdash \lambda x M : \overline{Typ(Q^{\mathbf{t}})}}(\text{FUN})$$

Et toujours d'après les conséquences du Lemme 8.11, on a $\overline{S}(Env(P^{\mathbf{t}})) = Env(Q^{\mathbf{t}})$ et $S(Typ(P^{\mathbf{t}})) = Typ(Q^{\mathbf{t}})$. Finalement, nous pouvons dériver $\Pi_0(Q^{\mathbf{t}}) \xrightarrow{exp} \overline{S}(\Pi_0(P^{\mathbf{t}}))$.

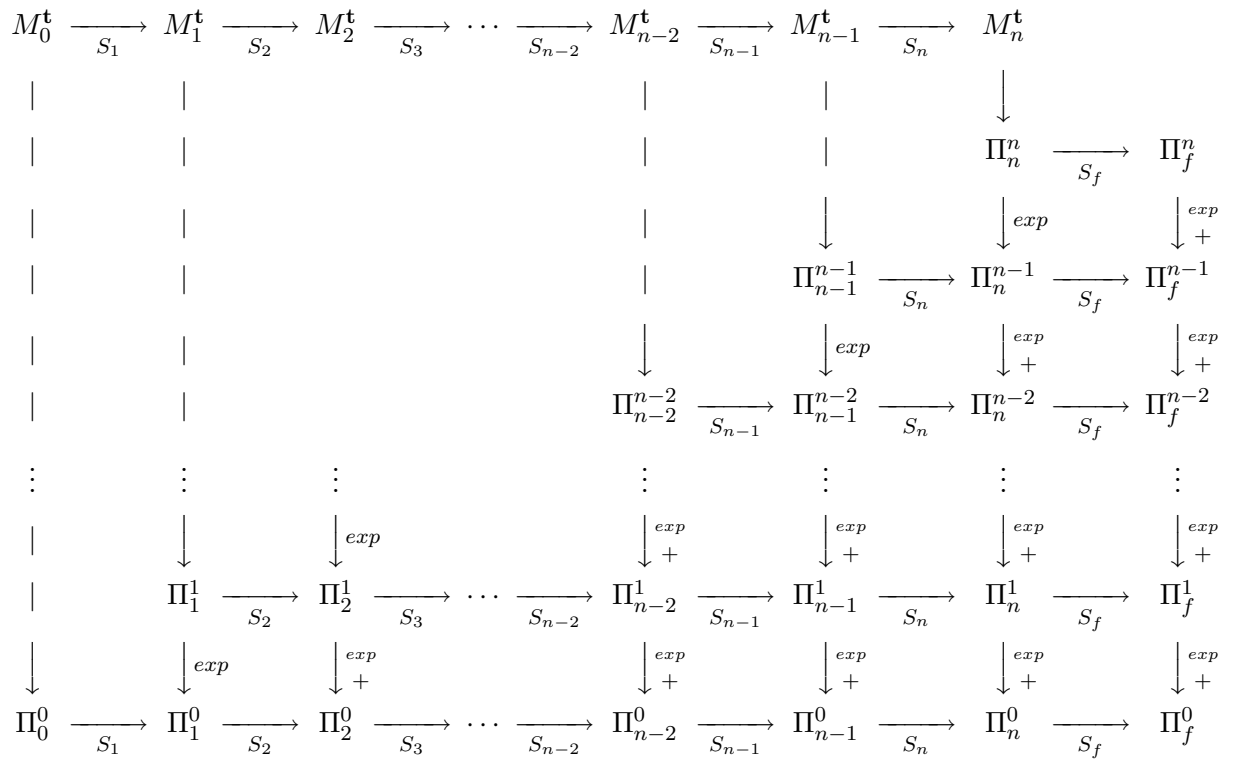
– Les deux derniers cas d'induction pour les noeuds $[M, N]$ sont très similaires aux précédents. □

8.2.4 Résultat principal

En rassemblant tous les lemmes précédents, on aboutit finalement à la validité du résultat fourni par l'algorithme en tant qu'arbre de typage :

Théorème 8.33 *Soit M un terme typable. Alors, le résultat de $Syst_0(M)$ est un arbre de typage valide pour M.*

Preuve: On rappelle les notations introduites plus haut :



où $M_0^{\mathbf{t}} = M$, $M_n^{\mathbf{t}} \in \mathcal{N}_{\mathcal{K}}$, $\Pi_i^i = \Pi_0(M_i^{\mathbf{t}})$ et Π_f^0 est le résultat de $Syst_0(M)$.

On procède par induction sur i. Supposons que nous avons prouvé que le diagramme commutatif est vrai jusqu'à la réduction S_i. Par le Lemme 8.32, on sait que $\Pi_{i+1}^{i+1} \xrightarrow{exp}_{\equiv} \overline{S_{i+1}}(\Pi_i^i) = \Pi_{i+1}^i$. Par hypothèse d'induction, $\Pi_i^i \xrightarrow{exp}{}^{+}_{\equiv} \Pi_i^0$. Si S_{i+1} est le résultat d'une application de la règle (R_n), elle

contient une duplication dont le territoire est T (si (R_0) a été utilisée, nous n'avons pas besoin de cette argumentation et nous sautons cette partie). Par le Lemme 8.26, $Compat(T, \Pi_{i+1}^{i+1})$ est vrai. A l'aide d'un usage répété des Lemmes 8.30 et 8.31, nous pouvons conclure que $\Pi_{i+1}^{i+1} \overset{exp}{\longrightarrow}^{+}_{\equiv} \Pi_{i+1}^{0}$.

En conclusion, le diagramme est vérifié jusqu'à la colonne $i = n$. Une argumentation similaire pour les dernières substitutions S_f (qui ne peuvent pas contenir de duplications, donc il n'est pas nécessaire de se préoccuper de la compatibilité) nous permet d'écrire que : $\Pi_f^n \overset{exp}{\longrightarrow}^{+}_{\equiv} \Pi_f^0$.

Puisque $M_n^{\mathbf{t}} \in \mathcal{N}_\mathcal{K}$, le Corollaire 8.21 nous dit que Π_f^n est un arbre de typage valide pour $M_n^{\mathbf{t}}$. Finalement, par application répétée du Lemme 8.28, le résultat Π_f^0 est également un arbre de typage valide pour $M_0^{\mathbf{t}}$. □

8.2.5 Résultat concernant le typage seul

Au lieu de considérer l'arbre de typage en entier, il est possible de ne s'intéresser qu'au *typage* $\Gamma \vdash M : \overline{\tau}$ à la racine de l'arbre [25]. Celui-ci peut être obtenu directement en appliquant l'algorithme à des états équations-type $(\mathcal{E}, \overline{\sigma})$ à partir d'un état initial $Syst'_0(M) = (\mathcal{E}_0(M^{\mathbf{t}}), Typ(M^{\mathbf{t}}))$ (ce que l'on ferait dans une implémentation pour un langage réel où *a priori* seul le type final du terme nous intéresse ; cette façon de procéder ne change en rien la terminaison de l'algorithme).

Le Théorème 8.33 nous dit déjà que si M est typable, l'algorithme retourne un typage pour M, mais il est possible d'être plus précis. En effet, il a été prouvé ([CDCV80, RV84]) que pour tout λ-terme M possédant une forme normale N, le typage canonique de N donné par l'algorithme $Canonic_1$ est un *typage principal* pour M, dans le sens qu'il s'agit d'un typage valide pour M, dont on peut dériver tous les autres typages de M par un certain ensemble d'opérations.

Théorème 8.34 *Si M est typable, le typage retourné par $Syst'_0(M)$ est un typage principal pour M.*

Preuve: En utilisant les notations du diagramme ci-dessus, les résultats de [CDCV80, RV84] impliquent que le typage à la racine de l'arbre Π_f^n est un typage principal pour $M_0^{\mathbf{t}}$ puisque $M_n^{\mathbf{t}}$ est sa forme normale et que $\Pi_f^n = Canonic_1(M_n^{\mathbf{t}})$ (Lemme 8.20). Comme de plus, la relation $\Pi \overset{exp}{\longrightarrow} \Pi'$ ne modifie pas le typage à la racine des squelettes de preuve, on en déduit que le typage à la racine de Π_f^0 est un typage principal pour $M_0^{\mathbf{t}}$. Le théorème s'ensuit. □

Dans le Chapitre 11, nous irons encore un peu plus loin en montrant que l'arbre de typage retourné par l'algorithme est principal dans son intégralité.

8.2.6 Résultats concernant le rang

Nous conjecturons que l'algorithme pour un rang fini r termine si et seulement si le terme est typable au rang r, mais nous n'avons pu le démontrer directement. Le Chapitre 11 traitera ce point en se raccrochant aux résultats du système $\mathbb{I}$.

Néanmoins, il est possible de donner quelques faits à ce sujet :

- pour montrer que l'algorithme à rang fini termine toujours, il faudrait montrer que le rang du système augmente forcément au fil des réductions ; comme on l'a constaté sur l'Exemple 7.10, c'est le rang du squelette qui doit augmenter, celui des équations pouvant rester constant.
- seules les substitutions générées par la règle (R_n) avec $n \geq 2$ sont susceptibles de faire augmenter le rang ; en effet, les règles (R_0) et (R_n) pour $n = 1$ ne font pas de duplication et

[25] On rappelle que $\Gamma \vdash M : \overline{\tau}$ est un *typage* pour M s'il existe un arbre de typage permettant de l'inférer.

ne réalisent que des substitutions simples de variables de types par des types déjà présents dans le squelette de preuve. Cependant, il est possible de trouver des exemples pour lesquels une utilisation de la règle (R_n) avec $n \geq 2$ ne fait pas augmenter strictement le rang de façon systématique.

– si l'algorithme diverge, il passe obligatoirement par une infinité de réductions (R_n) avec $n \geq 2$; en effet, les règles (R_0) et (R_n) pour $n = 1$ font strictement décroître le nombre d'équations restantes dans le système.

Chapitre 9

Implémentation prototype : TypI

Comme dans le cas de la récursion, le développement en parallèle d'un interpréteur nous a permis de faciliter la mise au point de l'algorithme d'inférence grâce à l'expérimentation rapide et automatisée. Ce prototype, baptisé TypI, implémente strictement l'algorithme à rang fini présenté jusqu'ici et permet de suivre et de diriger son exécution pas à pas. Il est écrit en Objective Caml et peut être téléchargé à l'adresse suivante :

`http://www-sop.inria.fr/mimosa/Pascal.Zimmer/typi.html`

Nous allons maintenant présenter les possibilités de ce logiciel sur un exemple.

9.1 Exemple d'interaction

Au lancement, le programme rappelle quel est le rang maximal autorisé (10 par défaut). Il est possible de donner un entier en ligne de commande et de choisir ainsi un rang maximal quelconque.

```
     Intersection Types Inference System

Using default maximal rank: 10

Term:
```

Le système demande alors quel est le λ-terme à analyser. On utilise la syntaxe suivante :
- $\lambda x M$ s'écrit "`\x M`" et $[M, N]$ s'écrit "`[M,N]`"
- afin de faciliter l'entrée, certaines constantes sont prédéfinies :

constante	valeur
`I`	$\lambda x\ x$
`D`	$\lambda x\ (x\ x) = \Delta$
`F`	$\lambda x \lambda y\ y$
`K`	$\lambda x \lambda y\ x$
`S`	$\lambda x \lambda y \lambda z\ (x\ z\ (y\ z))$
`@`	$\lambda f \lambda x\ (f\ x)$
`0`, `1`, `2` ...	$\lambda f \lambda x\ (f\ (f \ldots (f\ x)))$ (les entiers de Church)

constante	valeur
`succ`	$\lambda n \lambda f \lambda x\ (f\ (n\ f\ x))$
`add`	$\lambda m \lambda n \lambda g \lambda z\ (m\ g\ (n\ g\ z))$
`mult`	$\lambda m \lambda n \lambda g\ (m\ (n\ g))$
`exp`	$\lambda m \lambda n\ (n\ m)$
`mkpair`	$\lambda x \lambda y \lambda b\ (b\ x\ y)$
`fst`	$\lambda p\ (p\ (\lambda x \lambda y\ x))$
`snd`	$\lambda p\ (p\ (\lambda x \lambda y\ y))$

En guise d'exemple simple, nous allons inférer le typage de $I\ (\lambda x\ \Delta x)$. Le système commence par détailler les abréviations, et par décorer l'arbre syntaxique avec des variables fraîches et par le type de chaque noeud. Les variables de type sont de la forme "`t_i`" et plus généralement "`t_i_j_k...`" dans la suite après duplication. Les types intersection sont notés "`(t_1,...,t_2)`" et l'intersection vide "`()`". Le système construit ensuite l'ensemble initial des contraintes :

```
Term: I (\x D x)

Tree:
((\x x:t_0):t_0 -> t_0 (\x ((\x (x:t_1 x:t_2):t_3):(t_1,t_2) -> t_3 x:t_4):t_5)
:t_4 -> t_5):t_6

Constraints:
 [ 1]  t_2 -> t_3 = t_1   [t_2]
D[ 2]  t_4 -> t_5 = (t_1,t_2) -> t_3   [t_4]
D[ 3]  (t_4 -> t_5) -> t_6 = t_0 -> t_0   [t_1,t_2,t_3,t_4,t_5]

Current proof rank: 2

[S]tep(n)   [G]o   [A]ll:
```

On rentre alors dans la boucle principale d'interaction. A chaque étape, le système donne la liste des contraintes restant à résoudre. Celles-ci sont numérotées à partir de 1, et un `D` indique celles qui peuvent être décomposées à cet instant précis. Pour information, le système indique aussi le rang courant de l'arbre de typage (si celui-ci dépasse le rang maximal admis, le programme s'arrête et conclut à l'impossibilité de typer le terme pour ce rang).

Trois commandes sont alors proposées :

- `s` permet de décomposer la première contrainte décomposable dans la liste. Il aussi possible de demander la décomposition d'une autre contrainte que celle par défaut en indiquant son numéro (par exemple, `s3`).
- `g` rentre dans un mode non-interactif, en décomposant les contraintes une à une sans demander l'avis de l'utilisateur, jusqu'à finir le typage ou aboutir à un échec. Cette commande revient à taper systématiquement `s`.
- `a` rentre dans un mode non-interactif dans lequel toutes les stratégies de décomposition possibles sont testées de façon récursive. Etant donné qu'il y en a autant que de réductions normalisantes, l'explosion combinatoire réserve cette commande à de la vérification pour des petits termes (à l'origine dans un but de débogage de l'interpréteur).

Nous allons ici choisir de décomposer d'abord la 3$^{\text{e}}$ contrainte. Le système donne les substitutions que cela induit et met à jour la liste des contraintes :

```
[S]tep(n)   [G]o   [A]ll: s3

Decomposing: (t_4 -> t_5) -> t_6 = t_0 -> t_0   [t_1,t_2,t_3,t_4,t_5]
Substituting: t_6 => t_0   []
Substituting: t_0 => t_4 -> t_5   [t_1,t_2,t_3,t_4,t_5]

Constraints:
 [ 1]  t_2 -> t_3 = t_1   [t_2]
D[ 2]  t_4 -> t_5 = (t_1,t_2) -> t_3   [t_4]

Current proof rank: 2
```

```
[S]tep(n)   [G]o   [A]ll:
```

Ici, une seule décomposition est possible ; elle conduit cette fois-ci à une duplication, ce que le système nous signale.

```
[S]tep(n)   [G]o   [A]ll: s

Decomposing: t_4 -> t_5 = (t_1,t_2) -> t_3   [t_4]
Duplicating: dupl(2,t_4)
Substituting: t_5 => t_3   []
Substituting: t_1 => t_4_1   [t_4_1]
Substituting: t_2 => t_4_2   [t_4_2]

Constraints:
 [ 1]  t_4_2 -> t_3 = t_4_1   [t_4_2]

Current proof rank: 3

[S]tep(n)   [G]o   [A]ll:
```

Enfin, la dernière contrainte n'est pas décomposable par (R_n) ou (R_0) ; un dernier `s` va donc appliquer la règle (R_f) pour résoudre les contraintes restantes :

```
[S]tep(n)   [G]o   [A]ll: s

Substituting: t_4_1 => t_4_2 -> t_3

Current proof rank: 3
```

Ensuite, le système commence par récapituler toutes les substitutions et duplications effectuées (il s'agit du $\overline{S}$ global) :

```
Substitutions:
t_6 => t_0
t_0 => t_4 -> t_5
dupl(2,t_4)
t_5 => t_3
t_1 => t_4_1
t_2 => t_4_2
t_4_1 => t_4_2 -> t_3
```

Puis, il termine en donnant l'arbre de typage complet pour le terme, sous la forme d'une déduction logique : chaque ligne numérotée est soit un axiome, soit se déduit à partir des lignes précédentes dont les numéros sont donnés. Les rangs de l'arbre de typage et du type à la racine sont également mentionnés.

```
Proof:
(1): x:((t_4_2 -> t_3),t_4_2) -> t_3 |- x : ((t_4_2 -> t_3),t_4_2) -> t_3
(2): (1) => |- \x x : (((t_4_2 -> t_3),t_4_2) -> t_3) -> ((t_4_2 -> t_3),t_4_2)
 -> t_3
(3): x:t_4_2 -> t_3 |- x : t_4_2 -> t_3
(4): x:t_4_2 |- x : t_4_2
(5): (3) & (4) => x:t_4_2 -> t_3, x:t_4_2 |- x x : t_3
```

```
(6): (5) => |- \x x x : ((t_4_2 -> t_3),t_4_2) -> t_3
(7): x:t_4_2 -> t_3 |- x : t_4_2 -> t_3
(8): x:t_4_2 |- x : t_4_2
(9): (6) & (7) & (8) => x:t_4_2 -> t_3, x:t_4_2 |- (\x x x) x : t_3
(10): (9) => |- \x (\x x x) x : ((t_4_2 -> t_3),t_4_2) -> t_3
(11): (2) & (10) => |- (\x x) (\x (\x x x) x) : ((t_4_2 -> t_3),t_4_2) -> t_3

Proof rank: 3
Final type rank: 2
```

Chapitre 10

Variante $n = 0$

Dans ce chapitre, nous considérons une variante de l'algorithme pour laquelle la règle spéciale (R_0) est remplacée par la règle générale (R_n) lorsque $n = 0$.

Pour des raisons qui vont devenir évidentes, nous nous limiterons au λ-calcul Λ, à la place du $\Lambda_\mathcal{K}$-calcul, dont l'unique règle de réduction est :

$$\lambda x M N \longrightarrow_\beta \{x \mapsto N\}M$$

même lorsque $x \notin fv(M)$.

Modifications apportées à l'algorithme La règle principale de résolution des contraintes de l'algorithme devient donc :

$$\boxed{\begin{array}{c}(\{\tau \to t \ \stackrel{\perp}{=}\ t_1, \ldots, t_n \to \sigma \ \ [T]\} \cup \mathcal{E},\ \Pi) \ \longrightarrow\ (S(\mathcal{E}),\ \overline{S}(\Pi)) \\ \text{avec } S = \{t_i \mapsto \langle\tau\rangle^i, \langle T\rangle^i\}_{1\leq i\leq n} :: \{t \mapsto \sigma, \emptyset\} :: D(n, T)\end{array}} \quad (R_n)$$

pour **tout** $n \geq 0$ (la règle finale (R_f) demeure quant à elle inchangée).

La seule différence avec l'algorithme précédent a lieu dans le cas $n = 0$: une équation de la forme $\tau \to t \ \stackrel{\perp}{=}\ \omega \to \sigma \ \ [T]$ produit maintenant la substitution $\{t \mapsto \sigma, \emptyset\} :: D(0, T)$, au lieu de $\{t \mapsto \sigma, \emptyset\}$ seule. Les définitions pour $D(0, T)$ étendent les précédentes : grossièrement, $D(0, T)(X)$ est remplacé par $dupl_0(X)$ lorsque $ftv(X) \subseteq T$, ce qui signifie soit la séquence vide ω soit l'ensemble vide $\emptyset$. Par conséquent, des variables de type, des sous-squelettes dans des arbres de typage, ou des équations du système peuvent être supprimés lorsqu'ils correspondent à un terme "oublié".

Modifications apportées au système de types Naturellement, le système de types correspondant à cet algorithme doit être lui aussi adapté. Il s'agit d'une variante de celui utilisé jusqu'ici : les règles (Typ Appl Gen) et (Typ Appl ω) sont fondues en une seule règle :

$$\boxed{\begin{array}{c}\dfrac{}{x : \overline{\tau} \vdash x : \overline{\tau}}\,(\text{Typ Id}) \\[2ex] \dfrac{\Gamma \vdash M : \overline{\tau}}{\Gamma \setminus x \vdash \lambda x M : \Gamma(x) \to \overline{\tau}}\,(\text{Typ } \lambda) \\[2ex] \dfrac{\Gamma \vdash M : \overline{\tau_1}, \ldots, \overline{\tau_n} \to \overline{\sigma} \qquad \forall i,\ \Gamma_i \vdash N : \overline{\tau_i}}{\Gamma, \Gamma_1, \ldots, \Gamma_n \vdash MN : \overline{\sigma}}\,(\text{Typ Appl}) \qquad (n \geq 0)\end{array}}$$

On peut montrer que ce système est équivalent à un autre système de types avec intersection plusieurs fois étudié dans la littérature, souvent appelé $\mathcal{D}\Omega$ ([Kri90, AC98]) ou encore $\lambda\cap$ chez Barendregt [Bar92] et $\vdash_{s\omega}$ dans [SM97]. Nous en présentons une version légèrement adaptée à notre syntaxe des types. Sa caractéristique principale est d'expliciter le cas $n = 0$ de la règle (Typ Appl) ci-dessus à l'aide d'un axiome qui permet de donner un type ω à n'importe quel terme, qu'il soit typable sans ω ou non [26]. Une autre différence plus mineure est l'utilisation de *séquences binaires*, définies par :

$$\overline{s} ::= \omega \mid \overline{\tau} \mid \overline{s_1} \wedge \overline{s_2}$$

Les règles de ce système sont les suivantes :

$$\frac{}{x : \overline{\tau} \vdash x : \overline{\tau}}\,(\text{Typ Id})$$

$$\frac{\Gamma \vdash M : \overline{\tau}}{\Gamma \setminus x \vdash \lambda x M : \Gamma(x) \to \overline{\tau}}\,(\text{Typ } \lambda)$$

$$\frac{\Gamma_1 \vdash M : \overline{s} \to \overline{\sigma} \qquad \Gamma_2 \vdash N : \overline{s}}{\Gamma_1, \Gamma_2 \vdash MN : \overline{\sigma}}\,(\text{Typ Appl}) \qquad (\text{où } \overline{s} \text{ dans } \overline{s} \to \overline{\sigma} \text{ est “linéarisé” en une séquence})$$

$$\frac{\Gamma_1 \vdash M : \overline{s_1} \qquad \Gamma_2 \vdash M : \overline{s_2}}{\Gamma_1, \Gamma_2 \vdash M : \overline{s_1} \wedge \overline{s_2}}\,(\text{Typ } \wedge)$$

$$\frac{}{\vdash M : \omega}\,(\text{Typ } \omega)$$

Exemple 10.1 *Reprenons l'Exemple 7.5 à partir de l'état intermédiaire $(\mathcal{E}_1, \Pi_1)$ où :*

$$\mathcal{E}_1 = \{(\omega \to t_3^2) \to t_2 \;\doteq\; \omega \to t_3^1 \;\; [t_3^2]\}$$

$$\Pi_1 = \dfrac{\dfrac{\dfrac{\dfrac{\overline{x : \omega \to t_3^1 \vdash x : \omega \to t_3^1}\,(ID) \qquad \overline{x : \omega \to t_3^2 \vdash x : \omega \to t_3^2}\,(ID)}{x : \omega \to t_3^1, x : \omega \to t_3^2 \vdash x\ x : t_2}\,(APPL)}{\vdash \Delta : (\omega \to t_3^1), (\omega \to t_3^2) \to t_2}\,(FUN) \qquad \Pi_1' \qquad \Pi_1''}{y : t_3^1, y : t_2^3 \vdash \Delta\ (\lambda z\ y) : t_2}\,(APPL)$$

$$\textit{avec} \quad \Pi_1' = \frac{\overline{y : t_3^1 \vdash y : t_3^1}\,(ID)}{y : t_3^1 \vdash \lambda z\ y : \omega \to t_3^1}\,(FUN) \quad \textit{et} \quad \Pi_1'' = \frac{\overline{y : t_3^2 \vdash y : t_3^2}\,(ID)}{y : t_3^2 \vdash \lambda z\ y : \omega \to t_3^2}\,(FUN)$$

La variante de l'algorithme nous conduit à appliquer la règle (R_n) avec $n = 0$, ce qui génère la substitution :

$$S_1 = \{t_2 \mapsto t_3^1, \emptyset\} :: D(0, \{t_3^2\})$$

[26] On prendra garde à ne pas confondre ce système avec celui obtenu en remplaçant la règle (Typ ω) par :

$$\frac{\Gamma \vdash M : \overline{\tau}}{\Gamma \vdash M : \omega}\,(\text{Typ } \omega)$$

qui ne donne le type ω qu'aux termes qui sont typables par ailleurs. Ce système est équivalent à celui vu jusqu'ici dans les chapitres précédents.

Appliquée à Π_1, cette substitution a pour effet de supprimer les deux sous-arbres qui ne contiennent que t_3^2 comme variable libre (notamment Π_1'' en entier), ainsi que les sous-types dans le même cas. On obtient alors :

$$\Pi_2 = \dfrac{\dfrac{\dfrac{\dfrac{}{x : \omega \to t_3^1 \vdash x : \omega \to t_3^1}\,(ID)}{x : \omega \to t_3^1 \vdash x\ x : t_3^1}\,(APPL)}{\vdash \Delta : (\omega \to t_3^1) \to t_3^1}\,(FUN) \qquad \dfrac{\dfrac{}{y : t_3^1 \vdash y : t_3^1}\,(ID)}{y : t_3^1 \vdash \lambda z\ y : \omega \to t_3^1}\,(FUN)}{y : t_3^1 \vdash \Delta\ (\lambda z\ y) : t_3^1}\,(APPL)$$

Comme il n'y a plus d'équation à résoudre, Π_2 est le résultat de l'algorithme et on peut vérifier qu'il s'agit bien d'un arbre de typage valide pour le système de types modifié.

Modifications des preuves et des résultats Tous les résultats donnés au Chapitre 8 demeurent valides, avec quelques adaptations liées au changement de calcul et de système de types. Le Lemme 8.11 nécessite les adaptations les plus cruciales :

Lemme 10.2 *Soit $M \longrightarrow_\beta N$ une β-réduction. Soient $M^{\mathbf{t}} = M_0^{\mathbf{t}}(M)$ et $N^{\mathbf{t}} = M_0^{\mathbf{t}}(N)$. Alors il existe une réduction $(\mathcal{E}_0(M^{\mathbf{t}}), \Pi) \longrightarrow (S(\mathcal{E}), \overline{S}(\Pi))$ par la règle (R_n), et de plus :*

- *il existe une équation décomposable eq telle que $\mathcal{E}_0(M^{\mathbf{t}}) = \mathcal{E} \cup \{eq\}$*
- *$S(\mathcal{E}) \equiv \mathcal{E}_0(N^{\mathbf{t}})$*
- *$S(Typ(M^{\mathbf{t}})) \equiv Typ(N^{\mathbf{t}})$*
- *$S(VarTyp(x, M^{\mathbf{t}})) \equiv VarTyp(x, N^{\mathbf{t}})$ pour tout x*
- *$S(ftv(M^{\mathbf{t}})) \equiv ftv(N^{\mathbf{t}})$*

où les quatre équivalences ci-dessus font référence aux mêmes renommages θ.

Preuve: Très similaire à la preuve du Lemme 8.11. Les cas traités par induction sont exactement les mêmes ; seul le cas de l'axiome pour $x \notin fv(M)$ demande une attention particulière. □

Des adaptations similaires des Lemmes 8.12 et 8.13 sont également vérifiées. On obtient finalement :

Proposition 10.3 *Un terme M est normalisable si et seulement si le système $Syst_0(M)$ possède une réduction qui converge.*

Notons maintenant que, puisque dans le λ-calcul, les normalisations faible et forte ne sont plus équivalentes, un système peut cette fois-ci avoir à la fois des réductions divergentes et convergentes, suivant l'ordre dans lequel les équations sont résolues, alors qu'en $\Lambda_{\mathcal{K}}$-calcul, l'équivalence des normalisations empêchait ce double comportement non déterministe.

Exemple 10.4 *Le système $Syst_0(K\ x\ \Omega)$ diverge si on réduit systématiquement le redex de Ω et converge si on réduit ceux de K.*

Nous n'avons mentionné pour l'instant que la normalisation du terme analysé. Qu'en est-il du typage ? Il s'avère que, pour ce système de types, être typable n'est pas exactement équivalent à être normalisable (fortement ou faiblement). Leivant [Lei86] a étudié ce système et a donné diverses caractérisations utiles à notre cas précis, reprises par Sayag et Mauny [SM96, SM97], dont nous suivons la présentation (voir aussi Théorèmes 3 et 4, Chapitre *IV* de [Kri90], ou encore van Bakel [vB93]) :

Définition 10.5 *On définit le* niveau *d'une occurrence de ω dans un type $\overline{\tau}$ par induction sur $\overline{\tau}$:*

- *si* $\overline{\tau} = \omega \to \overline{\sigma}$*, alors le niveau de* ω *dans* $\overline{\tau}$ *est* 1
- *si* $\overline{\tau} = \overline{\tau_1}, \ldots, \overline{\tau_n} \to \overline{\sigma}$*, alors le niveau d'une occurrence de* ω *dans* $\overline{\tau}$ *est le niveau de l'occurrence correspondante de* ω *dans* $\overline{\sigma}$ *ou le niveau de l'occurrence correspondante de* ω *dans un des* $\overline{\tau_i} + 1$*.*

On dit que $\overline{\tau}$ *est un type propre si toutes les occurrences de* ω *dans* $\overline{\tau}$ *ont des niveaux impairs. On dit que* Γ *est un environnement de typage propre si toutes les occurrences de* ω *dans un type de* Γ *ont des niveaux pairs.*

Proposition 10.6 *Un terme est normalisable si et seulement si il est typable avec un type propre dans un environnement de typage propre.*

Proposition 10.7 *Un terme possède une forme normale de tête si et seulement si il est typable par un type non trivial (autre que* ω*).*

De la première caractérisation et de la Proposition 10.3, nous pouvons donc tirer l'implication suivante :

Corollaire 10.8 *Si* $Syst_0(M)$ *possède une réduction qui converge, alors* M *est typable.*

En revanche, la deuxième caractérisation nous apprend que la réciproque n'est pas vraie pour ce système de types. Il suffit de choisir un terme qui possède une forme normale de tête, mais qui ne soit pas normalisable : par exemple, le terme $x\Omega$ est typable, mais $Syst_0(x\Omega)$ diverge toujours.

Validité de l'arbre de typage La seconde partie des preuves reste quant à elle valide, avec quelques adaptations : il faut supprimer le second axiome de $\xrightarrow{exp}$ et étendre le premier au cas $n \geq 0$. On obtient alors sans difficulté la validité du résultat fourni par l'algorithme (vis-à-vis du système de types modifié) :

Proposition 10.9 *Si* $Syst_0(M)$ *converge, alors son résultat* Π *est un arbre de typage valide.*

Chapitre 11

Lien avec le système $\mathbb{I}$

Dans ce chapitre, nous étudions plus précisément le système $\mathbb{I}$ de Kfoury et Wells [KW04, Kfo99] et nous montrons qu'il existe une correspondance entre les opérations effectuées par ce système et notre algorithme. Etant donné que les deux systèmes utilisent des formalismes très différents et que le système $\mathbb{I}$ est généralement présenté par ces auteurs avec une abondance de notations et de détails, nous ne pourrons pas donner une preuve formelle complète. Cependant, nous pourrons en conjecturer sans que le doute soit permis que certains résultats importants déjà démontrés sont également vrais dans notre cas, notamment la principalité de l'arbre de typage inféré, ainsi que la décidabilité du typage à rang fini [KW99].

11.1 Système $\mathbb{I}$

Nous donnons une présentation succincte et légèrement adaptée à notre formalisme du système $\mathbb{I}$ tel qu'il est présenté dans [KW04], en nous attachant à montrer les similitudes et les différences. Nous renvoyons le lecteur à cet article pour les détails et les précisions manquantes. Pour être précis, il faudrait étendre ce système avec toutes les définitions nécessaires pour les constructions $[\_, \_]$ ([KW04] se limite au λ-calcul pur).

Variables de types et variables d'expansion Comme pour notre algorithme, la brique de base sur laquelle toutes les autres structures sont construites est la variable de type. On lui adjoint également les *variables d'expansion* qui vont constituer la différence fondamentale entre les deux systèmes. Au contraire de notre algorithme, dans le système $\mathbb{I}$ le mécanisme d'indexation fait partie intégrante des variables.

Partant de deux ensembles dénombrables de variables de type et d'expansion de base :

$$TVar_b = \{a_i \mid i \in \mathbb{N}\} \qquad EVar_b = \{F_i \mid i \in \mathbb{N}\}$$

les variables de types et d'expansion proprement dites sont définies comme des variables de base indexées par des chaînes de booléens :

$$TVar = \{a_i^s \mid i \in \mathbb{N}, s \in \{0,1\}^*\} \qquad EVar = \{F_i^s \mid i \in \mathbb{N}, s \in \{0,1\}^*\}$$

Les lettres α, β servent de métavariables pour les éléments de $TVar$, et la lettre F de métavariable pour ceux de $EVar$.

Types Les types manipulés par le système $\mathbb{I}$ sont similaires aux types premiers, si ce n'est que les conjonctions ne sont pas linéaires, mais suivent une structure binaire (comme les séquences binaires vues au Chapitre 10), non associative, et que des variables d'expansion peuvent apparaître à gauche des flèches :

$$\begin{array}{lll} \overline{\psi} \in \mathbb{T}^{\rightarrow} & ::= & \alpha \mid (\psi \rightarrow \overline{\psi}) \\ \psi \in \mathbb{T} & ::= & \overline{\psi} \mid (\psi \wedge \psi') \mid (F\psi) \end{array}$$

(avec la convention d'écriture que $F\psi$ est plus liant que $\psi \wedge \psi'$, lui-même plus liant que $\psi \rightarrow \overline{\psi}$)

Afin d'éviter toute confusion avec τ et $\overline{\tau}$, nous utilisons les symboles ψ et $\overline{\psi}$; on prendra garde à ce que la barre n'a pas la même signification dans les deux cas. De façon générale, on notera également $\vec{F}X$ une suite de variables d'expansion portant sur X, i.e. $F_1(\ldots(F_nX)\ldots)$.

Expansions Une expansion est un "contexte de conjonction" à n trous, contenant éventuellement des variables d'expansion :

$$e \in \mathbb{E} ::= \Box \mid (e \wedge e') \mid (Fe)$$

Comme d'habitude avec les contextes, on note $e[\psi_1, \ldots, \psi_n]$ le type obtenu en remplaçant les n trous de e par les types ψ_i. On appelle également *chemins* de e les n chaînes de booléens $s_i \in \{0,1\}^*$ qui décrivent le chemin à suivre pour aboutir au i^{e} trou (0 pour "prendre à gauche du $\wedge$", 1 pour "à droite").

Substitutions Une substitution $\mathbf{S}$ est une fonction totale $(TVar \cup EVar) \rightarrow (\mathbb{E} \cup \mathbb{T}^{\rightarrow})$ qui respecte les "sortes". i.e. $\mathbf{S}(F) \in \mathbb{E}$ et $\mathbf{S}(\alpha) \in \mathbb{T}^{\rightarrow}$. La substitution identité est notée $\{\!\!\{\ \}\!\!\}$. On peut distinguer deux sortes de substitutions portant sur une seule variable :

- $\mathbf{S} = \{\!\!\{\alpha := \overline{\tau}\}\!\!\}$ qui effectue la substitution des occurrences de α par $\overline{\tau}$ partout où celle-ci apparaît (une substitution classique)
- $\mathbf{S} = \{\!\!\{F := e\}\!\!\}$ qui remplace les occurrences de la variable d'expansion F par l'expansion e dans laquelle les trous sont instanciées en n copies du type qui suit F. La règle d'application d'une telle substitution est donc la suivante :

  $$\mathbf{S}(F\psi) = e[\mathbf{S}(\langle\psi\rangle^{s_1}), \ldots, \mathbf{S}(\langle\psi\rangle^{s_n})]$$

  où les s_i sont les chemins de e. Cette forme de substitution est le pendant de $D(n,T)$ qui réplique n fois (mais de façon linéaire) les structures dont les variables sont dans T.

Une bonne partie de [KW04] est consacrée à définir précisément certaines opérations relatives aux substitutions (notamment une forme non triviale de composition) ainsi que les propriétés qu'elles vérifient.

Contraintes Les contraintes sont des équations notées $\psi \doteq \psi'$ (nous pouvons d'ores et déjà mentionner que les deux membres sont inversés par rapport aux équations de notre algorithme). Les ensembles de contraintes sont notés Δ et le problème de *β-unification* consiste à résoudre ces ensembles.

On note $\vec{F}(\psi \doteq \psi')$ pour désigner $\vec{F}\psi \doteq \vec{F}\psi'$, ainsi que :

$$\vec{F}\Delta = \{\vec{F}\psi \doteq \vec{F}\psi' \mid (\psi \doteq \psi') \in \Delta\}$$

Simplification de contraintes Un algorithme auxiliaire de simplification des contraintes $\mathbf{simplify}(\Delta)$ est défini par :

- $\mathbf{simplify}(\emptyset) = \emptyset$
- $\mathbf{simplify}(\{\psi \doteq \psi'\} \cup \Delta) = \mathbf{simplify}(\psi \doteq \psi') \cup \mathbf{simplify}(\Delta)$
- $$\mathbf{simplify}(\psi \doteq \psi') = \begin{cases} F\ \mathbf{simplify}(\psi_1 \doteq \psi'_1) & \text{si } \psi = F\psi_1 \text{ et } \psi' = F\psi'_1 \\ \mathbf{simplify}(\psi'_1 \doteq \psi_1) \cup \mathbf{simplify}(\psi_2 \doteq \psi'_2) & \text{si } \psi = \psi_1 \to \psi_2 \text{ et } \psi' = \psi'_1 \to \psi'_2 \\ \mathbf{simplify}(\psi_1 \doteq \psi'_1) \cup \mathbf{simplify}(\psi_2 \doteq \psi'_2) & \text{si } \psi = \psi_1 \wedge \psi_2 \text{ et } \psi' = \psi'_1 \wedge \psi'_2 \\ \emptyset & \text{si } \psi = \psi' \\ \{\psi \doteq \psi'\} & \text{sinon} \end{cases}$$

Le travail effectué par cette opération est relativement simple : elle supprime les équations triviales $\psi \doteq \psi$, décompose les équations lorsque les deux membres sont des types flèches ou des conjonctions, et "factorise" les variables d'expansion qui apparaissent en tête des deux membres d'une équation.

Algorithme de β-unification Nous en arrivons au cœur de l'algorithme d'inférence. Les états intermédiaires du système sont décrits par $(\Delta, \mathbf{S}, \mathcal{E})$ où Δ est l'ensemble de contraintes restant à résoudre, $\mathbf{S}$ la substitution la plus générique calculée jusqu'ici, et $\mathcal{E}$ un "environnement" que nous ne décrirons pas ici. L'algorithme suivant $\mathbf{Unify}(\Delta)$ prend un ensemble de contraintes initiales Δ et retourne une substitution :

Etapes effectuées :
- appel initial : $\mathbf{Unify}(\Delta) \Rightarrow \mathbf{Unify}(\mathbf{simplify}(\Delta), \{\![\,]\!\}, \text{E-env}(\Delta))$
- appel final : $\mathbf{Unify}(\emptyset, \mathbf{S}, \mathcal{E}) \Rightarrow \mathbf{S}$
- $\mathbf{Unify}(\Delta_0, \mathbf{S}_0, \mathcal{E}) \Rightarrow \mathbf{Unify}(\Delta_1, \mathbf{S}_1, \mathcal{E})$, lorsque :
  - $\Delta_0 = \Delta \cup \vec{F}\{\rho \doteq \sigma\}$ et $\rho \doteq \sigma \Rightarrow \mathbf{S}$ est une instance de l'une des règles de réécriture ci-dessous
  - $\Delta_1 = \mathbf{simplify}(\mathbf{S}\Delta_0)$ et $\mathbf{S}_1 = \mathbf{S} \otimes_{\mathcal{E}} \mathbf{S}_0$

Règles de réécriture :

$\alpha \doteq \overline{\sigma}$	$\Rightarrow$	$\{\![\,\alpha := \overline{\sigma}\,]\!\}$	**(règle 1)**
$\overline{\rho} \doteq \alpha$	$\Rightarrow$	$\{\![\,\alpha := \overline{\rho}\,]\!\}$	**(règle 2)**
$F\overline{\rho} \doteq e[\overline{\sigma_1}, \ldots, \overline{\sigma_n}]$	$\Rightarrow$	$\{\![\,F := e\,]\!\}$	**(règle 3)**

(Les métavariables ρ, $\overline{\rho}$, σ et $\overline{\sigma}$ désignent des types ψ ou $\overline{\psi}$ qui respectent une polarité imposée. La définition précise de leur sous-grammaire est donnée dans [KW04]. De même, nous ne détaillerons pas ici l'opération E-env(Δ) ni la composition $\otimes_{\mathcal{E}}$.)

Squelettes et système de types Les *squelettes* sont l'équivalent des squelettes de preuve utilisés par notre algorithme. Leur définition est donnée par les règles suivantes, où $A \vdash_? M : \tau$ désigne $A \vdash M : \tau$ ou $A \vdash_e M : \tau$.

$$\frac{}{x : \overline{\psi} \vdash x : \overline{\psi}}\ \textsc{var} \qquad \frac{A_1 \vdash_? M : \psi_1 \qquad A_2 \vdash_? M : \psi_2}{A_1 \wedge A_2 \vdash_e M : \psi_1 \wedge \psi_2}\ \wedge$$

$$\frac{A[x \mapsto \psi] \vdash M : \overline{\psi'}}{A \vdash \lambda x M : \psi \to \overline{\psi'}}\ \textsc{abs-i} \qquad \frac{A \vdash_? M : \psi}{F\ A \vdash_e M : F\ \psi}\ \textsc{f}$$

$$\frac{A \vdash M : \overline{\psi'}}{A \vdash \lambda x M : \overline{\psi} \to \overline{\psi'}}\ \textsc{abs-k} \quad \text{si } x \notin fv(M) \qquad \frac{A_1 \vdash M : \overline{\psi_1} \qquad A_2 \vdash_? N : \psi_2}{A_1 \wedge A_2 \vdash MN : \overline{\psi_3}}\ \textsc{app}$$

Les différences essentielles tiennent à la présence des conjonctions binaires et au rajout de la règle F (qui est définie pour chaque variable d'expansion F). Par ailleurs, on peut noter une légère différence dans le traitement des fonctions qui n'utilisent pas leur argument. Le système $\mathbb{I}$ utilise dans ce cas la règle ABS-K qui autorise un type quelconque $\overline{\psi}$ comme argument (il n'est pas possible d'avoir une conjonction vide ω dans le système $\mathbb{I}$). A notre connaissance, ceci n'a pas d'impact sur l'expressivité du système, mais relève uniquement d'un choix de conception.

Les règles du système de types proprement dit pour lequel sont démontrés tous les théorèmes sont obtenues simplement en remplaçant la règle APP ci-dessus par la règle plus stricte suivante :

$$\frac{A_1 \vdash M : \psi \to \overline{\psi'} \qquad A_2 \vdash N : \psi}{A_1 \wedge A_2 \vdash MN : \overline{\psi'}}\ \textsc{app}$$

et on note $\mathcal{D}$ les *arbres de typage* (ou *dérivations*) obtenus. Une dérivation $\mathcal{D}$ est dite *principale* pour M (au sens de Kfoury et Wells) si c'est une dérivation pour M et si pour toute dérivation $\mathcal{D}'$ de M, il existe une substitution $\mathbf{S}$ telle que $\mathcal{D}' = \mathbf{S}(\mathcal{D})$.

Etat initial Etant donné un λ-terme M, les définitions suivantes produisent un ensemble de contraintes initiales $\Gamma_{\mathbb{I}}(M)$, ainsi que le *squelette* initial $Skel_{\mathbb{I}}(M)$ correspondant :

- si $M = x,$ pour $\alpha \in TVar_b$ fraîche :
  $Typ_{\mathbb{I}}(M) = \alpha$
  $Env_{\mathbb{I}}(M) = \{x \mapsto \alpha\}$
  $\Gamma_{\mathbb{I}}(M) = \emptyset$

  $$Skel_{\mathbb{I}}(M) = \frac{}{Env_{\mathbb{I}}(M) \vdash M : \alpha}\ \textsc{var}$$

- si $M = (N_1 N_2),$ pour $F \in EVar_b$ et $\beta \in TVar_b$ fraîches :
  $Typ_{\mathbb{I}}(M) = \beta$
  $Env_{\mathbb{I}}(M) = Env_{\mathbb{I}}(N_1) \wedge F\ Env_{\mathbb{I}}(N_2)$
  $\Gamma_{\mathbb{I}}(M) = \Gamma_{\mathbb{I}}(N_1) \cup F\ \Gamma_{\mathbb{I}}(N_2) \cup \{Typ_{\mathbb{I}}(N_1) \doteq F\ Typ_{\mathbb{I}}(N_2) \to \beta\}$

  $$Skel_{\mathbb{I}}(M) = \frac{Skel_{\mathbb{I}}(N_1) \qquad \dfrac{Skel_{\mathbb{I}}(N_2)}{F\ Env_{\mathbb{I}}(N_2) \vdash N_2 : F\ Typ_{\mathbb{I}}(N_2)}\ \textsc{f}}{Env_{\mathbb{I}}(M) \vdash M : \beta}\ \textsc{app}$$

• si $M = \lambda x N$, pour $\alpha \in TVar_b$ fraîche :

$$Typ_{\mathbb{I}}(M) = \begin{cases} Env_{\mathbb{I}}(N)(x) \to Typ_{\mathbb{I}}(N) & \text{si } Env_{\mathbb{I}}(N)(x) \text{ est défini} \\ \alpha \to Typ_{\mathbb{I}}(N) & \text{sinon} \end{cases}$$

$$Env_{\mathbb{I}}(M) = Env_{\mathbb{I}}(N) \setminus x$$

$$\Gamma_{\mathbb{I}}(M) = \Gamma_{\mathbb{I}}(N)$$

$$Skel_{\mathbb{I}}(M) = \frac{Skel_{\mathbb{I}}(N)}{Env_{\mathbb{I}}(M) \vdash M : Typ_{\mathbb{I}}(M)}\ {\scriptstyle R}$$

où si $x \in fv(N)$ alors $R =$ ABS-I sinon $R =$ ABS-K

On remarquera que ces fonctions donnent des résultats quasiment équivalents à $\mathcal{E}_0(M^{\mathbf{t}})$ et $\Pi_0(M^{\mathbf{t}})$ avec $M^{\mathbf{t}} = M_0^{\mathbf{t}}(M)$. Plus précisément :

- pour une variable x, il n'y a aucune différence.
- pour une fonction $\lambda x M$, la seule différence tient au traitement spécial lorsque $x \notin fv(M)$: on utilise ici la règle ABS-K et une variable fraîche α pour x là où on aurait utilisé (FUN) et ω.
- pour une application MN, la différence tient à l'introduction d'une variable d'expansion fraîche F. Celle-ci se retrouve placée en tête de l'environnement $F\ Env_{\mathbb{I}}(N_2)$, des contraintes générées par le sous-terme $F\ \Gamma_{\mathbb{I}}(N_2)$, du type $F\ Typ_{\mathbb{I}}(N_2)$ et du squelette $F\ Skel_{\mathbb{I}}(N_2)$. En d'autres termes, la variable d'expansion F préfixe tout ce qui a trait à N_2. Grossièrement parlant, ceci signifie qu'à chaque fois que nous rencontrerons F, nous saurons que la suite est un type, une contrainte, un squelette, etc... qui correspond à N_2 (c'est grâce à cette propriété que l'algorithme d'inférence du système $\mathbb{I}$ peut fonctionner). Dans notre algorithme, la propriété équivalente était assurée par le territoire $T = ftv(N_2^{\mathbf{t}})$ associé au Lemme 8.4. La correspondance entre les deux systèmes se base donc sur le fait que F et $ftv(N_2^{\mathbf{t}})$ véhiculent la même information et jouent des rôles identiques. Plus précisément, il est même possible d'utiliser les définitions de [KW04] pour expliciter la correspondance qui permet de passer d'un monde à l'autre :

$$F_T \quad \longleftrightarrow \quad T = \{v \mid F_T \in \text{E-path}(v, \Gamma_{\mathbb{I}}(M))\}$$

  En ce qui concerne l'équation générée pour ce noeud, il s'agit bien de la même (à une inversion près), puisqu'on a d'un côté $Typ_{\mathbb{I}}(N_1) \doteq F\ Typ_{\mathbb{I}}(N_2) \to \beta$ avec β fraîche, et de l'autre $Typ(N_2^{\mathbf{t}}) \to t \ \doteq\ Typ(N_1^{\mathbf{t}})\ \ [ftv(N_2^{\mathbf{t}})]$ avec t fraîche...

Algorithme d'inférence L'algorithme d'inférence pour système $\mathbb{I}$ est défini comme suit :

$$PT(M) = \begin{cases} \mathbf{S}(Skel_{\mathbb{I}}(M)) & \text{s'il existe une évaluation de } \mathbf{Unify}(\Gamma_{\mathbb{I}}(M)) \text{ qui converge} \\ & \text{et qui retourne } \mathbf{S} \\ \text{indéfini} & \text{si toutes les évaluations de } \mathbf{Unify}(\Gamma_{\mathbb{I}}(M)) \text{ divergent} \end{cases}$$

Le résultat essentiel démontré dans [KW04] prouve la correction de l'algorithme et la principalité du résultat obtenu :

Théorème 11.1 *Si M est typable dans le système $\mathbb{I}$, alors $PT(M)$ retourne une dérivation principale pour M, sinon $PT(M)$ diverge.*

Rang fini Une version à rang fini de l'algorithme d'inférence est également donnée dans [KW04] et le résultat suivant est démontré :

Théorème 11.2 *Soient $r \geq 0$ et M un λ-terme. Le problème de l'existence d'une dérivation de rang r pour M est décidable* [27].

11.2 Exemple et comparaison

Dans cette partie, nous donnons un exemple d'exécution de l'algorithme du système $\mathbb{I}$ (pour la partie contraintes seulement). Le terme choisi est celui de l'Exemple 7.6, $M = I(\lambda y(\Delta y)y)$. De façon intentionnelle, nous suivrons les mêmes étapes de réduction et nous encourageons le lecteur à comparer les évolutions.

En nommant les variables fraîches de façon cohérente avec l'Exemple 7.6, on obtient les contraintes initiales :

$$\Gamma_{\mathbb{I}}(M) = \left\{ \begin{array}{rcl} K(\alpha_1 & \doteq & F\ \alpha_2 \to \alpha_3), \\ K(\alpha_1 \wedge F\ \alpha_2 \to \alpha_3 & \doteq & G\ \alpha_4 \to \alpha_5), \\ K(\alpha_5 & \doteq & H\ \alpha_6 \to \alpha_7), \\ \alpha_0 \to \alpha_0 & \doteq & K\ (G\ \alpha_4 \wedge H\ \alpha_6 \to \alpha_7) \to \alpha_8 \end{array} \right\}$$

Après simplification, on obtient :

$$\Delta_0 = \mathbf{simplify}(\Gamma_{\mathbb{I}}(M)) = \left\{ \begin{array}{rcl} K(\alpha_1 & \doteq & F\ \alpha_2 \to \alpha_3), \\ K(\alpha_3 & \doteq & \alpha_5), \\ K(G\ \alpha_4 & \doteq & \alpha_1 \wedge F\ \alpha_2), \\ K(\alpha_5 & \doteq & H\ \alpha_6 \to \alpha_7), \\ \alpha_0 & \doteq & \alpha_8, \\ K\ (G\ \alpha_4 \wedge H\ \alpha_6 \to \alpha_7) & \doteq & \alpha_0 \end{array} \right\}$$

Toutes les contraintes sont des instances des règles de réécriture, nous avons donc le choix... Nous commençons par la troisième contrainte en utilisant la règle 3. Ceci génère donc la substitution :

$$\mathbf{S}_0 = \{\!\!\{ G := \Box \wedge F\ \Box \}\!\!\}$$

Et le système devient :

$$\Delta_1 = \mathbf{simplify}(\mathbf{S}_0(\Delta_0)) = \left\{ \begin{array}{rcl} K(\alpha_1 & \doteq & F\ \alpha_2 \to \alpha_3), \\ K(\alpha_3 & \doteq & \alpha_5), \\ K(\alpha_4^1 & \doteq & \alpha_1), \\ KF(\alpha_4^2 & \doteq & \alpha_2), \\ K(\alpha_5 & \doteq & H\ \alpha_6 \to \alpha_7), \\ \alpha_0 & \doteq & \alpha_8, \\ K\ ((\alpha_4^1 \wedge F\ \alpha_4^2) \wedge H\ \alpha_6 \to \alpha_7) & \doteq & \alpha_0 \end{array} \right\}$$

(Pour garder la similarité avec l'Exemple 7.6, nous numérotons α_4^1 et α_4^2 au lieu de α_4^0 et α_4^1 comme nous aurions dû le faire.)

Nous utilisons ensuite trois fois la règle 2 pour les deuxième, troisième et quatrième équations, ce qui nous amène à effectuer successivement les substitutions :

$$\mathbf{S}_1 = \{\!\!\{ \alpha_5 := \alpha_3 \}\!\!\}$$
$$\mathbf{S}_2 = \{\!\!\{ \alpha_1 := \alpha_4^1 \}\!\!\}$$
$$\mathbf{S}_3 = \{\!\!\{ \alpha_2 := \alpha_4^2 \}\!\!\}$$

[27] Si l'existence d'une dérivation de rang r est bien décidable, en réalité, les auteurs n'ont pu seulement démontrer qu'il *existait* une évaluation de l'algorithme qui termine pour un rang donné ; ils conjecturent (et nos propres expériences vont aussi dans ce sens) que *toutes* les évaluations terminent, mais sans avoir pu mener la preuve à son terme.

Le système est alors :

$$\Delta_4 = \mathbf{simplify}(\mathbf{S}_3(\mathbf{S}_2(\mathbf{S}_1(\Delta_1)))) = \left\{ \begin{array}{rcl} K(\alpha_4^1 & \doteq & F\ \alpha_4^2 \to \alpha_3), \\ K(\alpha_3 & \doteq & H\ \alpha_6 \to \alpha_7), \\ \alpha_0 & \doteq & \alpha_8, \\ K\ ((\alpha_4^1 \wedge F\ \alpha_4^2) \wedge H\ \alpha_6 \to \alpha_7) & \doteq & \alpha_0 \end{array} \right\}$$

On peut noter que ces contraintes sont celles de $\mathcal{E}_1$ dans l'Exemple 7.6. Résolvons maintenant la quatrième avec la règle 3. On pose donc :

$$\mathbf{S}_4 = \{\!\![\, K := \Box \,]\!\!\}$$

et on obtient :

$$\Delta_5 = \mathbf{simplify}(\mathbf{S}_4(\Delta_4)) = \left\{ \begin{array}{rcl} \alpha_4^1 & \doteq & F\ \alpha_4^2 \to \alpha_3, \\ \alpha_3 & \doteq & H\ \alpha_6 \to \alpha_7, \\ \alpha_0 & \doteq & \alpha_8, \\ (\alpha_4^1 \wedge F\ \alpha_4^2) \wedge H\ \alpha_6 \to \alpha_7 & \doteq & \alpha_0 \end{array} \right\}$$

A l'aide de la règle 2, décomposons la troisième puis la quatrième équation :

$$\begin{array}{l} \mathbf{S}_5 = \{\!\![\, \alpha_8 := \alpha_0 \,]\!\!\} \\ \mathbf{S}_6 = \{\!\![\, \alpha_0 := (\alpha_4^1 \wedge F\ \alpha_4^2) \wedge H\ \alpha_6 \to \alpha_7 \,]\!\!\} \end{array}$$

Il ne reste plus que :

$$\Delta_7 = \mathbf{simplify}(\mathbf{S}_6(\mathbf{S}_5(\Delta_5))) = \left\{ \begin{array}{rcl} \alpha_4^1 & \doteq & F\ \alpha_4^2 \to \alpha_3, \\ \alpha_3 & \doteq & H\ \alpha_6 \to \alpha_7 \end{array} \right\}$$

Enfin, deux utilisations de la règle 1 permettent d'achever la résolution des contraintes :

$$\begin{array}{l} \mathbf{S}_7 = \{\!\![\, \alpha_3 := H\ \alpha_6 \to \alpha_7 \,]\!\!\} \\ \mathbf{S}_8 = \{\!\![\, \alpha_4^1 := F\ \alpha_4^2 \to H\ \alpha_6 \to \alpha_7 \,]\!\!\} \end{array}$$

Si l'on appliquait les substitutions trouvées à $Skel_{\mathbb{I}}(M)$, on vérifierait qu'on trouve bien une dérivation *principale* pour M dans le système $\mathbb{I}$, et que cet arbre de typage est le même que celui trouvé à l'Exemple 7.6, à condition de supprimer les variables d'expansion.

11.3 Correspondance opérationnelle

Dans cette partie, nous justifions de façon générale la correspondance conjecturée sur l'exemple précédent. Plus précisément, nous allons "montrer" le résultat suivant :

Conjecture 11.3 *Soit $M \longrightarrow_{\mathcal{K}} N$ une $\mathcal{K}$-réduction. On sait que $\mathcal{E}_0(M_0^{\mathbf{t}}(M)) \longrightarrow_{\equiv} \mathcal{E}_0(M_0^{\mathbf{t}}(N))$ par la règle (R_n) ou (R_0). Alors, il existe une réduction de $\mathbf{simplify}(\Gamma_{\mathbb{I}}(M))$ en $\mathbf{simplify}(\Gamma_{\mathbb{I}}(N))$ (à un renommage des variables près) en $n+2$ étapes (3 pour (R_0)).*

Règle générale (R_n) Pour simplifier, nous ne considérerons pas la construction $[\_,\_]$ dans la réduction (celle-ci n'a pas d'influence). Explicitons donc le redex envisagé :

$$M = \mathcal{C}[(\lambda x N_1) N_2] \quad \longrightarrow_{\mathcal{K}} \quad N = \mathcal{C}[N_1\{x \leftarrow N_2\}]$$

où $\mathcal{C}$ désigne un contexte et $x \in fv(N_1)$. L'ensemble des contraintes initiales s'écrit alors sous la forme :

$$\Gamma_{\mathbb{I}}(M) = \left\{ \ \vec{F}(Env_{\mathbb{I}}(N_1)(x) \to Typ_{\mathbb{I}}(N_1) \ \doteq \ G\ Typ_{\mathbb{I}}(N_2) \to \beta) \ \right\} \cup \Delta$$

où G et β sont des variables fraîches, $\vec{F}$ les variables d'expansions introduites par le contexte $\mathcal{C}$, et Δ les contraintes correspondant aux autres noeuds-applications de M. Après simplification, ce système devient :

$$\Delta_0 = \mathbf{simplify}(\Gamma_{\mathbb{I}}(M)) = \left\{ \begin{array}{rcl} \vec{F}(Typ_{\mathbb{I}}(N_1) & \doteq & \beta), \\ \vec{F}(G\ Typ_{\mathbb{I}}(N_2) & \doteq & Env_{\mathbb{I}}(N_1)(x)) \end{array} \right\} \cup \mathbf{simplify}(\Delta)$$

En effet, comme G est fraîche, elle ne peut apparaître en tête de $Env_{\mathbb{I}}(N_1)(x)$ et donc on ne peut simplifier plus loin. De plus, étant donné la définition de la fonction $Env_{\mathbb{I}}$ et le fait que $x \in fv(N_1)$, on peut affirmer qu'il existe une expansion e telle que :

$$Env_{\mathbb{I}}(N_1)(x) = e[\alpha_1, \ldots, \alpha_n]$$

où n est le nombre d'occurrence de x dans N_1 et les α_i sont les variables fraîches qui leur ont été attribuées. La deuxième contrainte de Δ_0 est donc une instance de la règle de réécriture 3. Ceci génère la substitution :

$$\mathbf{S}_0 = \{\!\!\{ G := e \}\!\!\}$$

Pour des raisons analogues au Lemme 8.4, G ne peut apparaître ni dans $Typ_{\mathbb{I}}(N_1)$, ni dans $Typ_{\mathbb{I}}(N_2)$ ni dans $\vec{F}$. Le système évolue alors donc en :

$$\Delta_1 = \mathbf{simplify}(\mathbf{S}_0(\Delta_0)) = \left\{ \begin{array}{rcl} \vec{F}(Typ_{\mathbb{I}}(N_1) & \doteq & \beta), \\ \vec{F}(\langle Typ_{\mathbb{I}}(N_2)\rangle^{s_1} & \doteq & \alpha_1), \\ & \cdots & \\ \vec{F}(\langle Typ_{\mathbb{I}}(N_2)\rangle^{s_n} & \doteq & \alpha_n) \end{array} \right\} \cup \mathbf{simplify}(\mathbf{S}_0(\mathbf{simplify}(\Delta)))$$

où les s_i sont les chemins de e. On peut ensuite utiliser la règle 2 pour la première contrainte et poser :

$$\mathbf{S}_1 = \{\!\!\{ \beta := Typ_{\mathbb{I}}(N_1) \}\!\!\}$$

Comme β était fraîche, elle n'apparaît pas dans $Typ_{\mathbb{I}}(N_2)$ et est différente des α_i, donc :

$$\begin{array}{rcl} \Delta_2 & = & \mathbf{simplify}(\mathbf{S}_1(\Delta_1)) \\ & = & \left\{ \begin{array}{rcl} \vec{F}(\langle Typ_{\mathbb{I}}(N_2)\rangle^{s_1} & \doteq & \alpha_1), \\ & \cdots & \\ \vec{F}(\langle Typ_{\mathbb{I}}(N_2)\rangle^{s_n} & \doteq & \alpha_n) \end{array} \right\} \ \cup \ \mathbf{simplify}(\mathbf{S}_1(\mathbf{simplify}(\mathbf{S}_0(\mathbf{simplify}(\Delta))))) \end{array}$$

Enfin, il reste à utiliser n fois la règle 2 pour résoudre les n contraintes restantes avec :

$$\mathbf{S}_{i+1} = \{\!\!\{ \alpha_i := \langle Typ_{\mathbb{I}}(N_2)\rangle^{s_i} \}\!\!\} \quad \text{pour } 1 \leq i \leq n$$

Il reste alors :

$$\Delta_{n+2} = \mathbf{simplify}(\mathbf{S}_{n+1}(\cdots(\mathbf{S}_0(\mathbf{simplify}(\Delta)))\cdots))$$

Nous avons vu plus haut que $\mathbf{simplify}(\Gamma_{\mathbb{I}}(M))$ correspondait à $\mathcal{E}_0(M_0^{\mathbf{t}}(M))$. Puisqu'on sait que $\mathcal{E}_0(M_0^{\mathbf{t}}(M)) \longrightarrow_{\equiv} \mathcal{E}_0(M_0^{\mathbf{t}}(N))$ en générant $n+2$ substitutions équivalentes à celles que nous venons de réaliser, en reprenant les preuves du Chapitre 8, nous pouvons affirmer que Δ_{n+2} et $\mathcal{E}_0(M_0^{\mathbf{t}}(N))$ se correspondent (à un renommage près). Or, comme $\mathcal{E}_0(M_0^{\mathbf{t}}(N))$ correspond aussi à $\mathbf{simplify}(\Gamma_{\mathbb{I}}(N))$, les ensembles de contraintes Δ_{n+2} et $\mathbf{simplify}(\Gamma_{\mathbb{I}}(N))$ sont donc égaux à un renommage près.

Règle (R_0) On considère maintenant le redex

$$M = \mathcal{C}[(\lambda x N_1) N_2] \quad \longrightarrow_{\mathcal{K}} \quad N = \mathcal{C}[\ [N_1, N_2]\ ]$$

avec $x \notin fv(N_1)$. L'ensemble des contraintes initiales en est légèrement modifié :

$$\Gamma_{\mathbb{I}}(M) = \left\{\ \vec{F}(\alpha \to Typ_{\mathbb{I}}(N_1) \ \doteq\ G\ Typ_{\mathbb{I}}(N_2) \to \beta)\ \right\} \cup \Delta$$

où α est une variable fraîche. Après simplification, on a donc :

$$\Delta_0 = \mathbf{simplify}(\Gamma_{\mathbb{I}}(M)) = \left\{ \begin{array}{rcl} \vec{F}(Typ_{\mathbb{I}}(N_1) & \doteq & \beta), \\ \vec{F}(G\ Typ_{\mathbb{I}}(N_2) & \doteq & \alpha) \end{array} \right\} \cup \mathbf{simplify}(\Delta)$$

Il suffit alors d'utiliser successivement une fois la règle 3 et deux fois la règle 2 pour résoudre ces contraintes, avec :

$$\mathbf{S}_0 = \{\!\!\{ G := \Box \}\!\!\}$$

$$\mathbf{S}_1 = \{\!\!\{ \beta := Typ_{\mathbb{I}}(N_1) \}\!\!\}$$

$$\mathbf{S}_2 = \{\!\!\{ \alpha := Typ_{\mathbb{I}}(N_2) \}\!\!\}$$

(l'ordre est important, car α peut apparaître dans $Typ_{\mathbb{I}}(N_1)$). Il reste alors les autres contraintes :

$$\Delta_3 = \mathbf{simplify}(\mathbf{S}_2(\mathbf{simplify}(\mathbf{S}_1(\mathbf{simplify}(\mathbf{S}_0(\mathbf{simplify}(\Delta)))))))$$

On conclut alors de la même façon (il faut bien sûr étendre les définitions pour traiter $\Gamma_{\mathbb{I}}([N_1, N_2])$).

Règle finale (R_f) Lorsque M est en forme normale, on montre facilement que toutes les contraintes de $\Gamma_{\mathbb{I}}(M)$ sont de la forme $\alpha \doteq \overline{\sigma}$. Seule la règle 1 peut alors être utilisée, ce qui génère des substitutions de la forme $\{\!\!\{ \alpha := \overline{\sigma} \}\!\!\}$. Ceci équivaut à utiliser la règle finale (R_f) dans notre algorithme, et il n'est pas difficile de vérifier que les deux systèmes évoluent bien de la même façon jusqu'à éliminer toutes les contraintes.

Conclusion On constate donc que pour un terme M et une évolution de $Syst_0(M)$ donnés, il existe une évaluation de $\mathbf{Unify}(\Gamma_{\mathbb{I}}(M))$ qui suit exactement la même évolution et qui retourne une substitution équivalente en cas de convergence. On montrerait sans peine à l'aide de la correspondance entre variables d'expansion et territoires que l'application de cette substitution à $Skel_{\mathbb{I}}(M)$ et $\Pi_0(M_0^{\mathbf{t}}(M))$ fournit des arbres de typage équivalents (aux différences de conception près, et en supprimant les variables d'expansion dans le premier). On en conclut que $Syst_0(M)$ et $PT(M)$ calculent des résultats équivalents et décrivent en fait le même algorithme.

Où est alors la différence entre les deux systèmes ? Elle tient essentiellement au fait que **Unify** est beaucoup plus permissif sur l'ordre dans lequel les substitutions sont effectuées. Au prix d'une plus grande complexité dans les définitions de l'application et de la composition des substitutions, l'algorithme du système $\mathbb{I}$ autorise l'interversion de nombreuses micro-étapes, là où notre algorithme impose un ordre précis pour l'exécution de ces micro-étapes.

11.4 Nouveaux résultats

Forts de cette correspondance opérationnelle et suite à de nombreux tests exhaustifs sur le prototype (grâce à la commande `[A]ll`), nous sommes en mesure d'énoncer les deux conjectures "fortes" suivantes pour notre système :

Conjecture 11.4 *Le résultat donné par $Syst_0(M)$ pour un terme typable M est un arbre de typage principal.*

(Etant donné qu'il y a aussi une correspondance entre les substitutions **S** et $\overline{S}$, nous pouvons légitimement penser qu'il y a équivalence entre les deux notions de principalité, celles de Kfoury et Wells et celle des arbres de typage donnée en Définition 7.4.)

Conjecture 11.5 *Pour tout rang $r \geq 0$, l'algorithme de rang fini termine toujours et permet de décider quels termes sont typables au rang r.*

Chapitre 12

Ajout des références

Dans ce chapitre, nous allons considérer un langage étendu avec des références. Comme c'est le cas en ML, où les références obligent à restreindre la règle d'introduction du polymorphisme, cette modification va nous amener à restreindre les duplications. Nous montrerons également comment adapter l'inférence de types à ce cas.

12.1 Le problème posé par la duplication

En ML [MTHM97], la règle d'introduction du polymorphisme permet de généraliser le type des expressions apparaissant dans une liaison introduite par un `let`. Tant que l'on reste dans le cadre du λ-calcul pur, ceci ne pose pas de problème ; en revanche, avec les références (et plus généralement toute structure de données mutable), il faut limiter cette règle, comme le montre l'exemple suivant :

$$\texttt{let } x = \texttt{ ref [ ] in } (x := [\texttt{"chaîne"}] \texttt{ ; hd}(!\ x) + 1)$$

Dans cet exemple, si l'on suivait la règle sans restriction, on donnerait à `ref [ ]` le type $\forall\alpha.\alpha\ list\ ref$, où α est une variable de type généralisée. Les occurrences de x dans la suite auraient alors respectivement les types instanciés $string\ list\ ref$ et $int\ list\ ref$, et le corps du `let` serait bien typé. Or, bien entendu, ce programme provoque une erreur à l'exécution.

La solution simple généralement employée dans les langages actuels basés sur ML consiste à restreindre l'introduction du polymorphisme aux seuls cas où le corps de la liaison est *syntaxiquement* une valeur. Dans notre exemple, étant donné que `ref [ ]` est une application, le type $\alpha\ list\ ref$ reste donc monomorphe ; la variable α est alors instanciée une fois en *string*, puis provoque une erreur de typage lorsqu'on tente de l'unifier avec *int*. Des travaux antérieurs [Tof90, Wri95] ont montré que cette modification apportée à l'introduction du polymorphisme rendait les programmes typables sûrs.

Dans le cadre des systèmes de types avec intersection, un problème tout à fait similaire se pose (après tout, un type avec intersection n'est rien d'autre qu'une forme de polymorphisme limitée à un nombre défini de variantes) ; la seule différence étant que la "généralisation" a potentiellement lieu au niveau de chaque application et non pas du `let`. Examinons l'exemple suivant dans notre langage (que nous supposons enrichi de références, listes, chaînes et entiers) :

$$(\lambda r\ (r := [\texttt{"chaîne"}] \texttt{ ; hd}(!\ r) + 1))\ (\texttt{ref [ ]})$$

L'application de l'algorithme de typage nous conduit à donner à la partie gauche de l'application (i.e. la fonction) le type $string\ list\ ref, int\ list\ ref\ \rightarrow int$. Il faut donc typer `ref [ ]` deux fois, une

fois avec *string list ref*, et une fois avec *int list ref*. Ceci ne pose pas de problème particulier et le terme ci-dessus est donc bien typé. Or, comme précédemment, il provoque une erreur d'exécution puisqu'on aboutit à `"chaîne"` + 1...

Cette question fut traitée récemment par Davies et Pfenning [DP00] : ils ont proposé un système de types avec intersection pour un langage avec références et montré qu'il était sûr vis-à-vis de l'évaluation des termes. La solution adoptée est similaire à celle de ML : n'autoriser l'introduction de la conjonction dans les types d'une application que lorsque l'argument de l'application est *syntaxiquement* une valeur. Plus précisément, les deux règles importantes de leur système de types sont les suivantes :

$$\frac{\Delta;\Gamma \vdash V : A \qquad \Delta;\Gamma \vdash V : B}{\Delta;\Gamma \vdash V : A \wedge B} \qquad\qquad \frac{\Delta;\Gamma \vdash M : A \to B \qquad \Delta;\Gamma \vdash N : A}{\Delta;\Gamma \vdash MN : B}$$

où V désigne une valeur et M un terme quelconque (nous renvoyons le lecteur à [DP00] pour les notations et les preuves de correction). Ces auteurs ont également proposé un algorithme de *vérification* de types pour leur langage. Il ne s'agit que d'un algorithme de semi-inférence, car il nécessite de donner explicitement un grand nombre d'annotations de types supplémentaires. Nous nous proposons de montrer dans la suite de ce chapitre comment adapter notre algorithme d'inférence au cas des références afin d'éviter ces annotations de types.

12.2 Un langage avec références et constantes

Tout d'abord, nous allons étendre le langage considéré, par des primitives de création, de consultation et de modification de références, ainsi que par quelques constantes. Pour des raisons qui sont maintenant claires, nous distinguons la classe des *valeurs* V parmi les termes. On obtient la classe des *réponses* A (i.e. les résultats possibles d'une évaluation) en l'enrichissant par les adresses mémoires l (car celles-ci, bien qu'elles soient le résultat d'une évaluation, ne doivent pas être dupliquées, tout comme `ref` M).

$$V ::= \lambda x M \mid x \mid \texttt{ref} \mid ! \mid := \mid n \mid s \mid T \mid F \mid () \mid [V, M]$$

$$A ::= V \mid l \mid [A, M]$$

$$M, N ::= A \mid MN \mid [M, N]$$

où :

- $l \in Loc$ désigne une adresse mémoire
- `ref`, ! et := sont respectivement les primitives de création, consultation et de modification d'une adresse mémoire
- $n \in \mathbb{Z}$ est un entier, $s \in String$ une chaîne de caractères, T et F les booléens "vrai" et "faux", et () une constante "unit" (cette liste n'est bien entendu pas exhaustive ; on pourrait également rajouter sans difficulté des structures de données ayant un type générique comme les listes).

Les *contextes d'évaluation* sont donnés par la syntaxe suivante :

$$\mathbf{E} ::= [] \mid M\mathbf{E} \mid \mathbf{E}A \mid [M, \mathbf{E}] \mid [\mathbf{E}, M]$$

La *mémoire*, notée R ou S, est une fonction partielle qui associe une réponse aux adresses mémoires. Un *état* est un couple M, R tel que $floc(M) \cup floc(im(R)) \subseteq dom(R)$ (i.e. les adresses mémoires qui apparaissent libres dans le terme M ou dans une des cases mémoires de R ont toutes

une valeur associée dans R). On pourra vérifier que cette propriété est un invariant pour les règles de réduction.

Enfin, voici les règles de réduction pour ce langage étendu :

$$\begin{cases} (\lambda x M)\ A\ ,\ R \longrightarrow_{\mathcal{K}} \begin{cases} M\{x \mapsto A\}\ ,\ R & \text{si } x \in fv(M) \\ [M, A]\ , R & \text{sinon} \end{cases} \\ \texttt{ref}\ A\ ,\ R \longrightarrow_{\mathcal{K}} l\ ,\ R\{l \mapsto A\} \quad \text{où } l \text{ est frais (i.e. } l \notin dom(R)) \\ !l\ ,\ R \longrightarrow_{\mathcal{K}} R(l)\ ,\ R \\ := l\ A\ ,\ R \longrightarrow_{\mathcal{K}} ()\ ,\ R\{l \mapsto A\} \\ [A_1, M]\ A_2\ ,\ R \longrightarrow_{\mathcal{K}} [A_1\ A_2, M]\ ,\ R \\ M\ ,\ R \longrightarrow_{\mathcal{K}} N\ ,\ S \implies \mathbf{E}[M]\ ,\ R \longrightarrow_{\mathcal{K}} \mathbf{E}[N]\ ,\ S \end{cases}$$

Le premier axiome décrit la β-réduction classique. Les trois axiomes suivants régissent respectivement la création, consultation et modification d'adresses mémoires. Dans le style du calcul de Klop originel, le cinquième axiome permet de commuter applications et constructeur $[\_, \_]$ (sans lui, il aurait fallu donner des règles du style :

$$[\texttt{ref}, M_1, \ldots, M_n]\ A\ ,\ R \longrightarrow_{\mathcal{K}} [l, M_1, \ldots, M_n]\ ,\ R\{l \mapsto A\}$$

pour chacun des axiomes, comme nous l'avions fait jusqu'à présent). Enfin, la dernière règle permet d'utiliser ces redex dans n'importe quel contexte d'évaluation.

12.3 Inférence de types

Suivant les résultats de [DP00], la technique utilisée va consister à interdire la duplication lorsque l'argument d'une application n'est pas une valeur. Ceci va cependant poser quelques difficultés supplémentaires. Tout d'abord, parce que ne pas dupliquer risque de briser certaines propriétés d'invariance importantes (à savoir au plus une occurrence positive et une occurrence négative pour chaque variable de type). Egalement, parce que l'ajout de types construits comme les références complique la tâche de résolution des équations. Enfin, il faut pouvoir décider quand on est en présence d'une valeur ou non. Ceci peut se faire simplement sur le terme de départ en "marquant" les équations correspondant à des noeuds-applications dont l'argument n'est pas une valeur. Au moment de décomposer ces équations, il suffit alors de ne pas faire de duplication.

Cependant, il est également possible de savoir *au moment de la décomposition* si l'argument est une valeur ou non. En effet, à une petite modification près, il suffit de consulter son type pour obtenir cette information.

Types

A cette fin, nous séparons les variables de types t en deux sous-catégories : celles correspondant aux termes-variables, qu'on notera t_v, et celles correspondant aux applications, qu'on notera $t_@$. La syntaxe des types premiers simples est alors la suivante :

$$t_b ::= t_v \mid t_b\ ref \mid cte \mid t_b\ list$$

$$\tau, \sigma ::= t_v \mid \tau\ ref \mid cte \mid \tau\ list \mid t_@ \mid t_b, \ldots, t_b \to \tau$$

où :

- t_b désigne les *types de base* pouvant apparaître à gauche d'une flèche (dans la version d'origine, cette grammaire se réduisait à t_v seul).

- $\tau\ ref$ est intuitivement le type d'une référence contenant une valeur de type τ.
- cte désigne des types constants, par exemple int ou $string$, et $\tau\ list$ un exemple de type construit, ici les listes ; ces deux catégories servent essentiellement à montrer comment enrichir le langage et adapter l'algorithme à de nouveaux types, si besoin est.

La classification des types suivant qu'ils correspondent à une valeur ou non est alors donnée par le prédicat $ValueType$ suivant :

$$\begin{array}{l} ValueType(t_v) = true \\ ValueType(\tau\ ref) = false \\ ValueType(cte) = true \\ ValueType(\tau\ list) = ValueType(\tau) \\ ValueType(t_@) = false \\ ValueType(t_{b_1}, \ldots, t_{b_n} \to \tau) = true \end{array}$$

En ce qui concerne le typage des primitives sur les références, on leur attribue leurs types respectifs usuels par :

$$\begin{array}{ll} M_0^{\mathbf{t}}(\texttt{ref}) = t_v \to t_v\ ref & \text{où } t_v \text{ est fraîche} \\ M_0^{\mathbf{t}}(!) = t_v\ ref \to t_v & \text{où } t_v \text{ est fraîche} \\ M_0^{\mathbf{t}}(:=) = t_v\ ref \to t_v \to unit & \text{où } t_v \text{ est fraîche} \end{array}$$

et de même pour les primitives sur les listes, les entiers, etc...

Règles de décomposition

Nous ne détaillons pas les définitions concernant les substitutions, leur application ni la construction de l'état initial du système ; celles-ci se déduisent sans difficulté. Les équations sont maintenant de la forme : $\tau \to t_@ \ \stackrel{\perp}{=}\ \sigma\ \ [T]$. Si à un moment quelconque de l'algorithme, le système d'équations contient une contrainte dont le membre droit est de la forme $\sigma'\ ref$, ou $\sigma'\ list$, ou une constante cte, l'algorithme s'arrête et conclut à l'impossibilité du typage. Les seuls membres droits autorisés sont donc de la forme t_v, ou $t_@$, ou $t_{b_1}, \ldots, t_{b_n} \to \tau$. Et par conséquent, les seules équations décomposables sont de la forme :

$$\tau \to t_@ \ \stackrel{\perp}{=}\ t_{b_1}, \ldots, t_{b_n} \to \sigma\ \ [T]$$

Pour cela, on utilise les deux nouvelles règles suivantes (R_0) et (R_n). Lorsque $n = 0$:

$$\boxed{\begin{array}{c} (\{\tau \to t_@ \ \stackrel{\perp}{=}\ \omega \to \sigma\ \ [T]\} \cup \mathcal{E},\ \Pi) \ \longrightarrow\ (S(\mathcal{E}),\ \overline{S}(\Pi)) \\ \text{avec } S = \{t_@ \mapsto \sigma, \emptyset\} \end{array}} \quad (R_0)$$

Cette règle est similaire à celle d'origine et n'appelle pas de commentaires. Plus importante est la règle pour $n \geq 1$:

$$\boxed{\begin{array}{c} (\{\tau \to t_@ \ \stackrel{\perp}{=}\ t_{b_1}, \ldots, t_{b_n} \to \sigma\ \ [T]\} \cup \mathcal{E},\ \Pi) \ \longrightarrow\ (S(\mathcal{E}),\ \overline{S}(\Pi)) \\ \text{avec } S = \begin{cases} mgu(t_{b_i}, \langle\tau\rangle^i, \langle T\rangle^i)_{1\leq i\leq n} :: \{t_@ \mapsto \sigma, \emptyset\} :: D(n,T) & \text{si } ValueType(\tau) \\ mgu(t_{b_i}, \tau, T)_{1\leq i\leq n} :: \{t_@ \mapsto \sigma, \emptyset\} & \text{sinon} \end{cases} \end{array}} \quad (R_n)$$

Comme annoncé, lorsque l'argument est une valeur, on effectue la duplication $D(n,T)$, puis dans un deuxième temps, on unifie chaque type de base t_{b_i} avec la i^{e} projection de τ. Sinon, la duplication n'est pas effectuée, et chaque type de base t_{b_i} est unifié avec τ lui-même.

Ces règles font appel à une opération d'*unification* portant sur les t_{b_i} car ceux-ci peuvent être plus compliqués que de simples variables :

$$\begin{cases} mgu(t_v, \tau, T) = \{t_v \mapsto \tau, T\} \\ mgu(t_b\ ref, \tau\ ref, T) = mgu(t_b, \tau, T) \\ mgu(t_b\ list, \tau\ list, T) = mgu(t_b, \tau, T) \\ mgu(cte, cte, T) = \{\} \\ mgu(t_b, t_v, T) = \{t_v \mapsto t_b, ftv(t_b)\} \\ \textbf{fail} \text{ dans les autres cas} \end{cases}$$

Dans le cas d'une variable de type t_v, on procède comme précédemment avec la substitution $\{t_v \mapsto \tau, T\}$. Dans le cas d'un type construit ou d'une constante, il faut que τ soit de la même forme ou la même constante, et l'unification continue sur les sous-types. Ou alors, dans le cas où τ est lui-même une variable de type, c'est celle-ci qui est substituée (ceci s'avère nécessaire par exemple pour la primitive ! qui a le type $t_v\ ref \to t_v$). Dans tous les autres cas, l'unification échoue, ainsi que l'algorithme de typage.

Exemple 12.1 *Considérons l'exemple suivant :* $M = \Delta\ ((\lambda r \lambda x\ !\ r)\ (\texttt{ref}\ [\,]))$. *Sa version annotée est :*

$$\begin{aligned} M^{\mathbf{t}} \quad = \quad & ((\lambda x\ (x : t_{v_0}\ x : t_{v_1}) : t_{@_2}) \\ & \quad ((\lambda r \lambda x\ (! : (t_{v_3}\ ref \to t_{v_3})\ r : t_{v_4}) : t_{@_5}) \\ & \quad\quad (\texttt{ref} : (t_{v_6} \to t_{v_6}\ ref)\ \ [\,] : t_{v_7}\ list) : t_{@_8}) : t_{@_9}) : t_{@_{10}} \end{aligned}$$

et le système d'équations correspondant :

$$\mathcal{E}_0 = \left\{ \begin{array}{rcll} t_{v_1} \to t_{@_2} & \doteq & t_{v_0} & [t_{v_1}], \\ t_{v_4} \to t_{@_5} & \doteq & t_{v_3}\ ref \to t_{v_3} & [t_{v_4}], \\ t_{v_7}\ list \to t_{@_8} & \doteq & t_{v_6} \to t_{v_6}\ ref & [t_{v_7}], \\ t_{@_8} \to t_{@_9} & \doteq & t_{v_4} \to \omega \to t_{@_5} & [t_{v_6}, t_{v_7}, t_{@_8}], \\ t_{@_9} \to t_{@_{10}} & \doteq & t_{v_0}, t_{v_1} \to t_{@_2} & [t_{v_3}, t_{v_4}, t_{@_5}, t_{v_6}, t_{v_7}, t_{@_8}, t_{@_9}] \end{array} \right\}$$

Décomposons la deuxième contrainte ; ceci génère la substitution (nous utilisons bien évidemment ici une version adaptée de la règle (R_1) *afin d'éviter une duplication unaire inutile) :*

$$S_0 = \{t_{v_4} \mapsto t_{v_3}\ ref, \{t_{v_3}\}\} :: \{t_{@_5} \mapsto t_{v_3}, \emptyset\}$$

On notera que cette décomposition fait appel à l'unification particulière $mgu(t_{v_3}\ ref, t_{v_4}, \{t_{v_4}\})$. *On obtient alors le système :*

$$\mathcal{E}_1 = \left\{ \begin{array}{rcll} t_{v_1} \to t_{@_2} & \doteq & t_{v_0} & [t_{v_1}], \\ t_{v_7}\ list \to t_{@_8} & \doteq & t_{v_6} \to t_{v_6}\ ref & [t_{v_7}], \\ t_{@_8} \to t_{@_9} & \doteq & t_{v_3}\ ref \to \omega \to t_{v_3} & [t_{v_6}, t_{v_7}, t_{@_8}], \\ t_{@_9} \to t_{@_{10}} & \doteq & t_{v_0}, t_{v_1} \to t_{@_2} & [t_{v_3}, t_{v_6}, t_{v_7}, t_{@_8}, t_{@_9}] \end{array} \right\}$$

Décomposons maintenant la dernière équation. Comme $ValueType(t_{@_9})$ *est faux, il n'y a pas duplication. La substitution à effectuer est donc :*

$$S_1 = \{t_{v_0} \mapsto t_{@_9}, \{t_{v_3}, t_{v_6}, t_{v_7}, t_{@_8}, t_{@_9}\}\} :: \{t_{v_1} \mapsto t_{@_9}, \{t_{v_3}, t_{v_6}, t_{v_7}, t_{@_8}, t_{@_9}\}\} :: \{t_{@_{10}} \mapsto t_{@_2}, \emptyset\}$$

d'où le nouveau système :

$$\mathcal{E}_2 = \left\{ \begin{array}{rcll} t_{@_9} \to t_{@_2} & \stackrel{\perp}{=} & t_{@_9} & [t_{v_3}, t_{v_6}, t_{v_7}, t_{@_8}, t_{@_9}], \\ t_{v_7}\ list \to t_{@_8} & \stackrel{\perp}{=} & t_{v_6} \to t_{v_6}\ ref & [t_{v_7}], \\ t_{@_8} \to t_{@_9} & \stackrel{\perp}{=} & t_{v_3}\ ref \to \omega \to t_{v_3} & [t_{v_6}, t_{v_7}, t_{@_8}] \end{array} \right\}$$

Décomposons la deuxième équation de $\mathcal{E}_2$, ce qui nous donne :

$$S_2 = \{t_{v_6} \mapsto t_{v_7}\ list, \{t_{v_7}\}\} :: \{t_{@_8} \mapsto t_{v_6}\ ref, \emptyset\}$$

et le nouveau système :

$$\mathcal{E}_3 = \left\{ \begin{array}{rcll} t_{@_9} \to t_{@_2} & \stackrel{\perp}{=} & t_{@_9} & [t_{v_3}, t_{v_7}, t_{@_9}], \\ t_{v_7}\ list\ ref \to t_{@_9} & \stackrel{\perp}{=} & t_{v_3}\ ref \to \omega \to t_{v_3} & [t_{v_7}] \end{array} \right\}$$

A nouveau, c'est la deuxième équation qui se décompose, pour laquelle nous avons à procéder à l'unification non triviale $mgu(t_{v_3}\ ref, t_{v_7}\ list\ ref, \{t_{v_7}\})$. On obtient :

$$S_3 = \{t_{v_3} \mapsto t_{v_7}\ list, \{t_{v_7}\}\} :: \{t_{@_9} \mapsto (\omega \to t_{v_3}), \emptyset\}$$

Il reste alors l'unique équation :

$$\mathcal{E}_4 = \left\{ \begin{array}{rcll} (\omega \to t_{v_7}\ list) \to t_{@_2} & \stackrel{\perp}{=} & \omega \to t_{v_7}\ list & [t_{v_7}] \end{array} \right\}$$

que l'on peut décomposer alors par la règle (R_0) avec :

$$S_4 = \{t_{@_2} \mapsto t_{v_7}\ list, \emptyset\}$$

Si on applique la substitution globale au squelette initial $\Pi_0(M^{\mathbf{t}})$, on trouve :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \omega \to t_{v_7}\ list \vdash x : \omega \to t_{v_7}\ list}\,(ID) \qquad \dfrac{}{x : \omega \to t_{v_7}\ list \vdash x : \omega \to t_{v_7}\ list}\,(ID)}{x : \omega \to t_{v_7}\ list, x : \omega \to t_{v_7}\ list \vdash xx : t_{v_7}\ list}\,(APPL)}{\vdash \Delta : (\omega \to t_{v_7}\ list), (\omega \to t_{v_7}\ list) \to t_{v_7}\ list}\,(FUN) \qquad \Pi}{\vdash M : t_{v_7}\ list}\,(APPL)$$

$$avec \quad \Pi = \dfrac{\Pi' \qquad \dfrac{\vdash \mathtt{ref} : t_{v_7}\ list \to t_{v_7}\ list\ ref \qquad \vdash [\,] : t_{v_7}\ list}{\vdash \mathtt{ref}\ [\,] : t_{v_7}\ list\ ref}\,(APPL)}{\vdash (\lambda r \lambda x\ !\ r)\ (\mathtt{ref}\ [\,]) : \omega \to t_{v_7}\ list}\,(APPL)$$

$$et \quad \Pi' = \dfrac{\dfrac{\dfrac{\vdash ! : t_{v_7}\ list\ ref \to t_{v_7}\ list \qquad \dfrac{}{r : t_{v_7}\ list\ ref \vdash r : t_{v_7}\ list\ ref}\,(ID)}{r : t_{v_7}\ list\ ref \vdash ! \ r : t_{v_7}\ list}\,(APPL)}{r : t_{v_7}\ list\ ref \vdash \lambda x\ !\ r : \omega \to t_{v_7}\ list}\,(FUN)}{\vdash \lambda r \lambda x\ !\ r : t_{v_7}\ list\ ref \to \omega \to t_{v_7}\ list}\,(FUN)$$

On pourra noter que cet arbre n'est pas tout à fait un arbre de typage valide : en effet, le noeud-racine (APPL) n'est pas correctement formé, et ceci en raison de la non-duplication que nous avons effectuée. Cependant, il suffirait soit de doubler le sous-arbre Π, soit d'autoriser la contraction dans le typage de Δ pour obtenir un arbre valide.

Exemple 12.2 *Considérons le terme* $M = (\lambda f\ \Delta(ff))\ I$. *Sa version annotée est :*

$$M^{\mathbf{t}} = ((\lambda f\ ((\lambda x\ (x : t_{v_0}\ x : t_{v_1}) : t_{@_2})\ (f : t_{v_3}\ f : t_{v_4}) : t_{@_5}) : t_{@_6})\ (\lambda x\ x : t_{v_7})) : t_{@_8}$$

et le système d'équations correspondant par :

$$\mathcal{E}_0 = \left\{ \begin{array}{rcll} t_{v_1} \to t_{@_2} & \stackrel{\perp}{=} & t_{v_0} & [t_{v_1}], \\ t_{v_4} \to t_{@_5} & \stackrel{\perp}{=} & t_{v_3} & [t_{v_4}], \\ t_{@_5} \to t_{@_6} & \stackrel{\perp}{=} & t_{v_0}, t_{v_1} \to t_{@_2} & [t_{v_3}, t_{v_4}, t_{@_5}], \\ (t_{v_7} \to t_{v_7}) \to t_{@_8} & \stackrel{\perp}{=} & t_{v_3}, t_{v_4} \to t_{@_6} & [t_{v_7}] \end{array} \right\}$$

Si on choisit de décomposer d'abord la dernière équation, comme $ValueType(t_{v_7} \to t_{v_7})$ *est vraie, il y a bien duplication :*

$$S_0 = \{t_{v_3} \mapsto (t^1_{v_7} \to t^1_{v_7}), \{t^1_{v_7}\}\} :: \{t_{v_4} \mapsto (t^2_{v_7} \to t^2_{v_7}), \{t^2_{v_7}\}\} :: \{t_{@_8} \mapsto t_{@_6}, \emptyset\} :: D(2, \{t_{v_7}\})$$

et le système devient :

$$\mathcal{E}_1 = \left\{ \begin{array}{rcll} t_{v_1} \to t_{@_2} & \stackrel{\perp}{=} & t_{v_0} & [t_{v_1}], \\ (t^2_{v_7} \to t^2_{v_7}) \to t_{@_5} & \stackrel{\perp}{=} & t^1_{v_7} \to t^1_{v_7} & [t^2_{v_7}], \\ t_{@_5} \to t_{@_6} & \stackrel{\perp}{=} & t_{v_0}, t_{v_1} \to t_{@_2} & [t^1_{v_7}, t^2_{v_7}, t_{@_5}] \end{array} \right\}$$

Si on décompose alors la deuxième équation, on obtient :

$$S_1 = \{t^1_{v_7} \mapsto (t^2_{v_7} \to t^2_{v_7}), \{t^2_{v_7}\}\} :: \{t_{@_5} \mapsto t^1_{v_7}, \emptyset\}$$

et :

$$\mathcal{E}_2 = \left\{ \begin{array}{rcll} t_{v_1} \to t_{@_2} & \stackrel{\perp}{=} & t_{v_0} & [t_{v_1}], \\ (t^2_{v_7} \to t^2_{v_7}) \to t_{@_6} & \stackrel{\perp}{=} & t_{v_0}, t_{v_1} \to t_{@_2} & [t^2_{v_7}] \end{array} \right\}$$

A cet instant, puisque $ValueType(t^2_{v_7} \to t^2_{v_7})$, *la deuxième équation donne lieu à duplication, et on termine l'algorithme de typage sans encombre par :*

$$S_2 = \{t_{v_0} \mapsto (t^{21}_{v_7} \to t^{21}_{v_7}), \{t^{21}_{v_7}\}\} :: \{t_{v_1} \mapsto (t^{22}_{v_7} \to t^{22}_{v_7}), \{t^{22}_{v_7}\}\} :: \{t_{@_6} \mapsto t_{@_2}, \emptyset\} :: D(2, \{t^2_{v_7}\})$$

puis :

$$S_3 = \{t^{21}_{v_7} \mapsto (t^{22}_{v_7} \to t^{22}_{v_7}), \{t^{22}_{v_7}\}\} :: \{t_{@_2} \mapsto t^{21}_{v_7}, \emptyset\}$$

On note que l'équation dont le noeud est $t_{@_6}$ *n'a reçu une valeur dans son membre gauche qu'à partir de* $\mathcal{E}_2$. *Dans* $\mathcal{E}_0$ *et* $\mathcal{E}_1$, *elle avait encore* $t_{@_5}$ *à cette place, qui n'est pas une valeur. Que se serait-il donc passé si nous avions décidé de commencer par décomposer d'abord cette équation ?*

Dans ce cas, il n'y a pas duplication et on a :

$$S'_0 = \{t_{v_0} \mapsto t_{@_5}, \{t_{v_3}, t_{v_4}, t_{@_5}\}\} :: \{t_{v_1} \mapsto t_{@_5}, \{t_{v_3}, t_{v_4}, t_{@_5}\}\} :: \{t_{@_6} \mapsto t_{@_2}, \emptyset\}$$

ce qui nous donne :

$$\mathcal{E}'_1 = \left\{ \begin{array}{rcll} t_{@_5} \to t_{@_2} & \stackrel{\perp}{=} & t_{@_5} & [t_{v_3}, t_{v_4}, t_{@_5}], \\ t_{v_4} \to t_{@_5} & \stackrel{\perp}{=} & t_{v_3} & [t_{v_4}], \\ (t_{v_7} \to t_{v_7}) \to t_{@_8} & \stackrel{\perp}{=} & t_{v_3}, t_{v_4} \to t_{@_2} & [t_{v_7}] \end{array} \right\}$$

(ces équations correspondent au terme $(\lambda f\ (ff)\ (ff))\ I$, où les deux occurrences de (ff) sont "partagées"). Décomposons maintenant la troisième équation comme précédemment :

$$S'_1 = \{t_{v_3} \mapsto (t^1_{v_7} \to t^1_{v_7}), \{t^1_{v_7}\}\} :: \{t_{v_4} \mapsto (t^2_{v_7} \to t^2_{v_7}), \{t^2_{v_7}\}\} :: \{t_{@_8} \mapsto t_{@_2}, \emptyset\} :: D(2, \{t_{v_7}\})$$

On obtient le système :

$$\mathcal{E}'_2 = \left\{ \begin{array}{rcll} t_{@_5} \to t_{@_2} & \stackrel{\perp}{=} & t_{@_5} & [t^1_{v_7}, t^2_{v_7}, t_{@_5}], \\ (t^2_{v_7} \to t^2_{v_7}) \to t_{@_5} & \stackrel{\perp}{=} & t^1_{v_7} \to t^1_{v_7} & [t^2_{v_7}] \end{array} \right\}$$

(ce qui correspond au terme $(II)\ (II)$, où les deux occurrences de (II) sont partagées). La décomposition suivante donne :

$$S'_2 = \{t^1_{v_7} \mapsto (t^2_{v_7} \to t^2_{v_7}), \{t^2_{v_7}\}\} :: \{t_{@_5} \mapsto t^1_{v_7}, \emptyset\}$$

ce qui nous laisse avec l'unique équation :

$$\mathcal{E}'_3 = \{\ (t^2_{v_7} \to t^2_{v_7}) \to t_{@_2} \quad \stackrel{\perp}{=} \quad t^2_{v_7} \to t^2_{v_7} \quad [t^2_{v_7}]\ \}$$

(autrement dit le terme II où les deux occurrences de I sont partagées). Ceci nous conduit à la substitution :

$$S'_3 = \{t^2_{v_7} \mapsto (t^2_{v_7} \to t^2_{v_7}), \{t^2_{v_7}\}\} :: \{t_{@_2} \mapsto t^2_{v_7}, \emptyset\}$$

ce qui est interdit car cette dernière est cyclique ! L'algorithme d'inférence échoue donc dans ce cas...

En conclusion, on constate que suivant l'ordre dans lequel les équations sont décomposées, le système peut ou non avoir une solution...

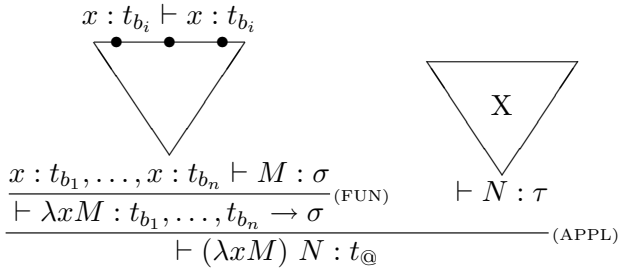
12.4 Ordre de résolution des contraintes

Examinons plus précisément l'origine du problème sur l'exemple précédent. Pour cela, considérons une équation générique qui ne duplique pas :

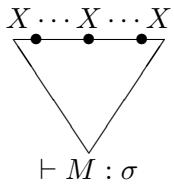
$$(\{\tau \to t_@ \ \stackrel{\perp}{=}\ t_{b_1}, \ldots, t_{b_n} \to \sigma \ \ [T]\} \cup \mathcal{E},\ \Pi) \ \longrightarrow\ (S(\mathcal{E}),\ \overline{S}(\Pi))$$

$$\text{avec } S = mgu(t_{b_i}, \tau, T)_{1 \le i \le n} :: \{t_@ \mapsto \sigma, \emptyset\} \text{ et } ValueType(\tau) = false$$

On sait que cette réduction correspond à un redex $(\lambda x M)\ N \longrightarrow_{\mathcal{K}} M\{x \mapsto N\}$ (on sait ici que $x \in fv(M)$. Considérons également le changement produit sur les arbres syntaxiques correspondants, décorés par quelques annotations de types ; celui de $(\lambda x M)\ N$ a la forme suivante :



tandis que celui de $M\{x \mapsto N\}$ aura la forme suivante :



où X désigne toujours le *même* arbre, celui de N ci-dessus (dans le cas d'une duplication, on aurait ici n copies $\langle X \rangle^i$ différentes de cet arbre).

Le problème vient du fait que le type τ à la racine de X ($t_{@_5}$ dans l'exemple 12.2) apparaît maintenant plusieurs fois dans les équations en position positive, puisqu'il peut être utilisé dans plusieurs applications . Puisque $ValueType(\tau)$ est faux par hypothèse, on sait que τ est de la forme $\tau'\ ref$ ou $t'_{@}$. Dans le premier cas, il n'y a rien à faire puisqu'on souhaite de toute façon que toutes les utilisations de $\tau'\ ref$ partagent le même type. Dans le deuxième cas, tant que l'équation dont le noeud est $t'_{@}$ ne réduit pas, aucune substitution n'est faite sur cette variable (puisque ce noeud en porte la seule occurrence négative susceptible de générer une substitution), et donc il n'y a pas de risque de générer une substitution cyclique dans une autre équation (sauf éventuellement au moment de la règle (R_f), mais il s'agit alors d'une véritable erreur de typage qui ne peut être évitée).

Il s'agit donc de s'assurer que le noeud racine de l'argument N dans un redex qui ne duplique pas ne sera plus jamais réduit dans le futur. Pour cela, les deux conditions suivantes sont suffisantes :

- que cet argument soit en forme normale de tête,
- et que ce redex ne soit pas dans un contexte $(\lambda y\ [\ ])\ N'$ qui lie la variable de tête y de N dans un autre redex.

Dans l'Exemple 12.2, c'est ce deuxième point qui n'était pas vérifié, puisque nous avons réduit le terme M par :

$$M = (\lambda f\ \Delta\ (f f))\ I \quad \longrightarrow_{\mathcal{K}} \quad (\lambda f\ (f f)\ (f f))\ I$$

alors que la variable de tête f de $f f$ était liée dans le redex $(\lambda f \ldots)\ I$. Notons également qu'à l'Exemple 12.1, où nous ne nous étions pas préoccupés de l'ordre d'évaluation, il y avait aussi potentiellement un danger de substitution cyclique, bien visible sur la dernière équation :

$$\mathcal{E}_4 = \left\{\ (\omega \to t_{v_7}\ list) \to t_{@_2} \quad \doteq \quad \omega \to t_{v_7}\ list \quad [t_{v_7}]\ \right\}$$

Heureusement pour nous, c'était la règle (R_0) qui s'appliquait ; mais si nous avions eu $t_{b_1}, \ldots, t_{b_n}$ à la place de ω, nous aurions essuyé un échec de l'algorithme.

Comment satisfaire les deux conditions évoquées ci-dessus ? Une méthode simple consiste à suivre la stratégie de réduction des équations données par une évaluation en *appel par valeur*, excepté éventuellement en ce qui concerne les membres droits des opérateurs $[\_, \_]$ qui sont de toute façon indépendants. C'est la stratégie que nous avons suivie, avec succès, dans la première réduction de l'Exemple 12.2. C'est également celle correspondant à la relation de réduction présentée à la Section 12.2. Il est logique que nous soyons obligés d'imposer un ordre de décomposition des équations : en effet, dans le $\Lambda_{\mathcal{K}}$-calcul sans références, comme les termes typables sont aussi fortement normalisants, l'ordre de réduction est sans importance. En revanche, avec des références, ceci n'est plus vrai : dans $(\lambda r\ !r)$ (`ref` 0), il serait absurde de chercher à réduire d'abord $!r$ sous le λ avant d'avoir alloué la référence en mémoire...

Il ne reste plus qu'à montrer comment, à partir de l'ensemble des équations, on peut retrouver celles qui correspondent à cette stratégie et peuvent être décomposées sans danger. Nous allons

pour cela définir un algorithme $Next(\mathcal{E})$ retournant ce sous-ensemble d'équations de $\mathcal{E}$. Dans la suite, on notera $eq(t_@, \mathcal{E})$ la fonction qui retourne l'équation de $\mathcal{E}$ dont le noeud est $t_@$.

La fonction suivante $NF_\mathcal{E}(\tau)$ est un prédicat qui retourne vrai si et seulement si le terme dont le type est τ ne contient pas de redex (toujours sans tenir compte des membres droits des opérateurs $[\_,\_]$) :

$$
\begin{array}{l}
NF_\mathcal{E}(t_{b_1}, \ldots, t_{b_n} \to \tau) = NF_\mathcal{E}(\tau) \\
NF_\mathcal{E}(t_v) = true \\
NF_\mathcal{E}(cte) = true \\
NF_\mathcal{E}(\tau\ ref) = NF_\mathcal{E}(\tau) \\
NF_\mathcal{E}(t_@) = NF_\mathcal{E}(\tau) \quad \wedge \quad NF_\mathcal{E}(\sigma) \quad \wedge \quad \sigma \neq t_{b_1}, \ldots, t_{b_n} \to \sigma' \quad \text{où } (\tau \to t_@ \ \stackrel{\perp}{=}\ \sigma \ \ [T]) = eq(t_@, \mathcal{E})
\end{array}
$$

On rappelle que $rgt(\sigma)$ désigne le type le plus à droite des flèches dans σ. Le prédicat suivant vérifie que nous ne tentons pas de réduire un redex qui apparaît dans la partie gauche d'un autre redex :

$$NC_\mathcal{E}(t_@) = \forall(\tau' \to t'_@ \ \stackrel{\perp}{=}\ \sigma' \ \ [T']) \in \mathcal{E},\ rgt(\sigma') = t_@ \Longrightarrow (\sigma' = t_@ \ \wedge\ NC_\mathcal{E}(t'_@))$$

Plus précisément, partant d'un noeud-application $t_@$, on cherche si ce noeud apparaît en partie gauche d'une autre application, éventuellement sous des λs (la condition $rgt(\sigma') = t_@$ ne peut être vérifiée que par au plus une équation de $\mathcal{E}$). Si c'est le cas, on vérifie qu'il n'y a en fait pas de λ (par $\sigma' = t_@$), donc pas de redex, et on continue récursivement la remontée dans l'arbre. Celle-ci s'arrête lorsqu'on atteint la racine du terme, ou lorsque le noeud $t_@$ apparaît en partie droite d'une application, auquel cas le prédicat est trivialement vrai car aucune équation ne vérifie $rgt(\sigma') = t_@$.

Il ne reste plus qu'à combiner ces définitions : une équation de $\mathcal{E}$ est décomposable si et seulement si l'argument est en forme normale, s'il n'y a pas de redex englobant à gauche, et bien entendu si l'équation correspond à un redex :

$$Next(\mathcal{E}) = \{(\tau \to t_@ \ \stackrel{\perp}{=}\ \sigma \ \ [T]) \in \mathcal{E} \ /\ NF_\mathcal{E}(\tau) \ \wedge\ NC_\mathcal{E}(t_@) \ \wedge\ \sigma = t_{b_1}, \ldots, t_{b_n} \to \sigma'\}$$

On peut montrer que cet algorithme retourne le sous-ensemble de $\mathcal{E}$ correspondant à l'ordre d'évaluation recherché, et que les propriétés suivantes sont vérifiées :

- $Next(\mathcal{E})$ contient exactement une équation par sous-arbre syntaxique réductible, à savoir une pour l'arbre racine s'il contient un redex, et une par membre droit d'un $[\_,\_]$ qui contient un redex.
- $Next(\mathcal{E}) = \emptyset$ si et seulement si $\mathcal{E}$ ne peut plus réduire, autrement dit si le terme analysé est en forme normale.

Chapitre 13

Ajout de la récursion

Dans ce chapitre final, nous nous intéressons à l'ajout d'un opérateur de point fixe au langage et nous étendons l'algorithme d'inférence. La technique utilisée va consister dans un premier temps à appliquer l'algorithme comme s'il n'y avait pas de point fixe, puis à résoudre un certain nombre d'équations induites par la récursion, grâce à de l'unification simple associée à de la contraction. Il s'agit d'une méthode "minimaliste" dans le sens où elle va conduire à recontracter des duplications introduites par l'inférence, et qu'elle va déclarer non typables des termes qui pourraient peut-être l'être si l'on procédait à une analyse plus fine... Ceci dit, on sait par ailleurs que l'inférence de types polymorphes pour la récursion en ML est un problème indécidable ; il n'est donc pas surprenant que nous nous heurtions également à des difficultés dans le cadre des types avec intersection.

13.1 Point fixe

13.1.1 Extension des définitions

Nous commençons par étendre le langage avec un opérateur de point fixe, similaire à celui déjà rencontré au Chapitre 5, et dont la sémantique est évidente :

$$M ::= \ \ldots \mid \mu x \ M$$

Concernant les squelettes de preuve, tout comme pour les applications, ils ne seront pas forcément des instances valides d'un arbre de typage. Plus précisément, on donnera à $\mu x \ M$ le même type qu'à M et on supprimera dans l'environnement les liaisons portant sur la variable x :

$$\Pi ::= \ \ldots \mid \ \frac{\Pi}{Env(\Pi) \setminus x \vdash \mu x \ Term(\Pi) : Typ(\Pi)}^{(\text{REC})}$$

Les termes annotés sont également étendus de façon triviale, la récursion ne demandant pas d'annotation particulière :

$$M^{\mathbf{t}} ::= \ \ldots \mid \mu x \ M^{\mathbf{t}}$$

Avec ces définitions, il est possible d'étendre les fonctions du Chapitre 7 qui permettent de construire l'état initial du système :

$$Typ(\mu x\ M^{\mathbf{t}}) = Typ(M^{\mathbf{t}})$$

$$VarTyp(x, \mu x\ M^{\mathbf{t}}) = \emptyset$$
$$VarTyp(y, \mu x\ M^{\mathbf{t}}) = VarTyp(y, M^{\mathbf{t}}) \quad \text{pour } x \neq y$$

$$\mathcal{E}_0(\mu x\ M^{\mathbf{t}}) = \mathcal{E}_0(M^{\mathbf{t}})$$

$$Env(\mu x\ M^{\mathbf{t}}) = Env(M^{\mathbf{t}}) \setminus x$$
$$\Pi_0(\mu x\ M^{\mathbf{t}}) = \frac{\Pi_0(M^{\mathbf{t}})}{Env(\mu x\ M^{\mathbf{t}}) \vdash UnTyp(\mu x\ M^{\mathbf{t}}) : Typ(\mu x\ M^{\mathbf{t}})}\,(\text{REC})$$

13.1.2 Extension de l'algorithme

Grossièrement, $\mu x\ M$ se comporte comme $\lambda x\ M$ pour ce qui est de la gestion des variables, et comme M pour ce qui est de son type, ce qui le rend totalement transparent par rapport au reste du terme et de l'algorithme d'inférence. On peut donc sans danger appliquer ce dernier (que ce soit la version d'origine du Chapitre 7 ou une des variantes présentées dans la suite) aux termes contenant un ou plusieurs points fixes.

Une fois celui-ci terminé, le résultat retourné est un arbre de typage valide, sauf en ce qui concerne les noeuds récursions. Si on lui applique une substitution quelconque, on aura toujours un arbre de typage valide (sauf pour les noeuds récursions) : en effet, l'arbre retourné étant un typage principal, lui appliquer une substitution revient à en prendre une instance particulière. Pour résoudre les récursions, la méthode va donc consister à trouver une substitution la plus générale possible qui unifie les types inférés pour la variable récursive dans l'environnement avec le type inféré pour le corps de la récursion, et à l'appliquer à l'arbre tout entier.

Plus précisément, si un noeud récursif non résolu de l'arbre après l'inférence est donné par :

$$\frac{\Gamma, x : \overline{\sigma_1}, \ldots, x : \overline{\sigma_n} \vdash M : \overline{\tau}}{\Gamma \vdash \mu x\ M : \overline{\tau}}\,(\text{REC})$$

alors nous aurons à résoudre les n équations suivantes :

$$\Theta = \{\overline{\sigma_i} = \overline{\tau} \mid 1 \leq i \leq n\}$$

(Etant donné que les $\overline{\sigma_i}$ et $\overline{\tau}$ sont obtenus à partir du squelette initial, ces équations équivalent à résoudre pour chaque $\mu x\ M^{\mathbf{t}}$:

$$\overline{S}(VarTyp(x, M^{\mathbf{t}})) = \overline{S}(Typ(M^{\mathbf{t}}))$$

si l'on note $\overline{S}$ la substitution globale générée par l'algorithme et que l'on duplique les équations lorsque le membre gauche est une conjonction.)

Pour résoudre ces équations, nous utiliserons un algorithme d'unification standard, avec contraction pour traiter les conjonctions, dont voici les règles :

$$\begin{array}{lll}
\{t = t\} \cup \Theta & \Longrightarrow & \Theta \\
\{t = \overline{\tau}\} \cup \Theta \quad \text{avec } \overline{\tau} \neq t & \Longrightarrow & \{t \mapsto \overline{\tau}\}(\Theta) \ \text{ si } t \notin ftv(\overline{\tau}) \ \text{ sinon } \mathbf{fail} \\
\{\overline{\tau} = t\} \cup \Theta \quad \text{avec } \overline{\tau} \neq t & \Longrightarrow & \{t \mapsto \overline{\tau}\}(\Theta) \ \text{ si } t \notin ftv(\overline{\tau}) \ \text{ sinon } \mathbf{fail} \\
\{\overline{\tau_1}, \ldots, \overline{\tau_n} \to \overline{\tau'} = \overline{\sigma_1}, \ldots, \overline{\sigma_p} \to \overline{\sigma'}\} \cup \Theta & \Longrightarrow & \\
\multicolumn{3}{r}{\{\overline{\tau_1} = \overline{\tau_i} \mid 2 \leq i \leq n\} \cup \{\overline{\tau_1} = \overline{\sigma_j} \mid 1 \leq j \leq p\} \cup \{\overline{\tau'} = \overline{\sigma'}\} \cup \Theta}
\end{array}$$

Les équations portant sur des variables génèrent des substitutions simples (qui sont immédiatement appliquées au reste des équations ainsi qu'au squelette), les types fonctionnels sont décomposés, et les conjonctions sont remplacées par une série d'équations pour toutes les unifier [28]. Dans le cas où la résolution conduirait à appliquer une substitution récursive (i.e. $\{t \mapsto \overline{\tau}\}$ avec $t \in ftv(\overline{\tau})$), l'unification échoue et par conséquent l'algorithme d'inférence également.

13.1.3 Modification du système de types

Avant de pouvoir affirmer que le squelette final obtenu est un arbre de typage valide, il nous reste à préciser quelle règle nous adoptons pour le système de types. En raison de la règle de contraction, l'algorithme d'unification ci-dessus ne génère pas forcément une substitution $\overline{S}$ telle que $\forall i\ \overline{S}(\overline{\sigma_i}) = \overline{S}(\overline{\tau})$ pour chacun des noeuds récursions. En effet, on pourra vérifier que cette égalité ne tient que modulo une certaine relation $\equiv$, cette relation étant la plus petite congruence induite par les lois de commutativité et de *contraction* (encore appelée loi d'*idempotence*) sur les types premiers simples [29] :

$$\ldots, \overline{\tau_1}, \overline{\tau_2}, \ldots \to \overline{\sigma} \quad \equiv \quad \ldots, \overline{\tau_2}, \overline{\tau_1}, \ldots \to \overline{\sigma}$$

$$\ldots, \overline{\tau}, \overline{\tau}, \ldots \to \overline{\sigma} \quad \equiv \quad \ldots, \overline{\tau}, \ldots \to \overline{\sigma}$$

On a alors bien $\forall i\ \overline{S}(\overline{\sigma_i}) \equiv \overline{S}(\overline{\tau})$ pour chaque noeud récursion. Ceci nous conduit donc à adopter la règle appropriée suivante pour le système de types [30] :

$$\frac{\Gamma, x : \overline{\sigma_1}, \ldots, x : \overline{\sigma_n} \vdash M : \overline{\tau}}{\Gamma \vdash \mu x\ M : \overline{\tau}}(\text{REC}) \quad \text{avec } \forall i\ \overline{\sigma_i} \equiv \overline{\tau}$$

et on pourra vérifier que les noeuds récursions dans l'arbre retourné par l'inférence lorsque celle-ci termine sont bien des instances de cette règle.

13.2 Exemples

Exemple 13.1 *En guise d'application, nous allons inférer un typage pour le λ-terme récursif M suivant :*

$$M = \mu f \lambda x\ (\Delta\ (\lambda z\ fx))$$

[28] D'autres alternatives sont envisageables, par exemple tester toutes les partitions possibles de $\{\overline{\tau_1}, \ldots, \overline{\tau_n}\}$ contre toutes les partitions de même cardinalité de $\{\overline{\sigma_1}, \ldots, \overline{\sigma_p}\}$, mais de telles solutions ont une complexité très grande et rien ne dit qu'elles donnent de meilleurs résultats dans le cas général.

[29] En réalité, les substitutions générées par l'algorithme d'unification donnent des types modulo une congruence plus stricte, induite par la loi de contraction forte suivante :

$$\overline{\tau}, \ldots, \overline{\tau} \to \overline{\sigma} \quad \equiv \quad \overline{\tau} \to \overline{\sigma}$$

[30] On pourrait aussi envisager d'autoriser directement les contractions au niveau de l'axiome pour les variables, et ainsi d'avoir une règle plus simple pour la récursion, comme suit :

$$\frac{\overline{\sigma} \equiv \overline{\sigma'}}{x : \overline{\sigma} \vdash x : \overline{\sigma'}}(\text{ID}) \qquad \text{et} \qquad \frac{\Gamma, x : \overline{\sigma}, \ldots, x : \overline{\sigma} \vdash M : \overline{\sigma}}{\Gamma \vdash \mu x\ M : \overline{\sigma}}(\text{REC})$$

mais on autoriserait du coup des types différents au niveau des λs (par exemple, on pourrait typer l'identité $\lambda x x$ avec $(t, t \to t) \to (t \to t)$), ce qui n'est pas possible dans le système d'origine, et ce qui n'est pas forcément souhaitable. Pour éviter ce problème, il suffirait cependant de différencier au niveau de la syntaxe deux classes de variables x et X, l'une pour les variables fonctionnelles et l'autre pour les variables récursives.

où $\Delta = \lambda x\ (xx)$. *Sa version annotée est la suivante :*

$$M^{\mathbf{t}} = M_0^{\mathbf{t}}(M) = \mu f \lambda x\ ((\lambda y\ (y : t_0\ y : t_1) : t_2)\ (\lambda z\ (f : t_3\ x : t_4) : t_5)) : t_6$$

d'où l'on déduit l'état initial $(\mathcal{E}_0, \Pi_0)$ *du système :*

$$\mathcal{E}_0 = \left\{ \begin{array}{rcll} t_1 \to t_2 & \doteq & t_0 & [t_1], \\ t_4 \to t_5 & \doteq & t_3 & [t_4], \\ (\omega \to t_5) \to t_6 & \doteq & t_0, t_1 \to t_2 & [t_3, t_4, t_5] \end{array} \right\}$$

$$\Pi_0 = \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y : t_0 \vdash y : t_0}\,(ID) \quad \dfrac{}{y : t_1 \vdash y : t_1}\,(ID)}{y : t_0, y : t_1 \vdash y\ y : t_2}\,(APPL)}{\vdash \Delta : t_0, t_1 \to t_2}\,(FUN) \quad \Pi_0'}{f : t_3, x : t_4 \vdash \Delta\ (\lambda z\ fx) : t_6}\,(APPL)}{f : t_3 \vdash \lambda x\ (\Delta\ (\lambda z\ fx)) : t_4 \to t_6}\,(FUN)}{\vdash \mu f \lambda x\ (\Delta\ (\lambda z\ fx)) : t_4 \to t_6}\,(REC)$$

$$\textit{avec} \quad \Pi_0' = \dfrac{\dfrac{\dfrac{}{f : t_3 \vdash f : t_3}\,(ID) \quad \dfrac{}{x : t_4 \vdash x : t_4}\,(ID)}{f : t_3, x : t_4 \vdash fx : t_5}\,(APPL)}{f : t_3, x : t_4 \vdash \lambda z\ fx : \omega \to t_5}\,(FUN)$$

En appliquant l'algorithme d'inférence, on génère les substitutions suivantes (dans l'ordre) :

$$S_0 = \{t_0 \mapsto (\omega \to t_5^1), \{t_3^1, t_4^1, t_5^1\}\} :: \{t_1 \mapsto (\omega \to t_5^2), \{t_3^2, t_4^2, t_5^2\}\} :: \{t_6 \mapsto t_2, \emptyset\} :: D(2, \{t_3, t_4, t_5\})$$

$$S_1 = \{t_2 \mapsto t_5^1, \emptyset\}$$

$$\overline{S}_2 = \{t_3^1 \mapsto t_4^1 \to t_5^1\} :: \{t_3^2 \mapsto t_4^2 \to t_5^2\}$$

Si on note $\overline{S} = \overline{S_2} :: \overline{S_1} :: \overline{S_0}$ *la substitution globale effectuée, le squelette de preuve* $\overline{S}(\Pi_0)$ *à la fin de l'algorithme est le suivant :*

$$\Pi_3 = \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{f : t_4^1 \to t_5^1, f : t_4^2 \to t_5^2, x : t_4^1, x : t_4^2 \vdash \Delta\ (\lambda z\ fx) : t_5^1}\,(APPL)}{f : t_4^1 \to t_5^1, f : t_4^2 \to t_5^2 \vdash \lambda x\ (\Delta\ (\lambda z\ fx)) : t_4^1, t_4^2 \to t_5^1}\,(FUN)}{\vdash \mu f \lambda x\ (\Delta\ (\lambda z\ fx)) : t_4^1, t_4^2 \to t_5^1}\,(REC)} \quad \Pi_3' \quad \Pi_3'' \quad \Pi_3'''$$

$$\textit{avec} \quad \Pi_3' = \dfrac{\dfrac{\dfrac{}{y : \omega \to t_5^1 \vdash y : \omega \to t_5^1}\,(ID) \quad \dfrac{}{y : \omega \to t_5^2 \vdash y : \omega \to t_5^2}\,(ID)}{y : \omega \to t_5^1, y : \omega \to t_5^2 \vdash y\ y : t_5^1}\,(APPL)}{\vdash \Delta : (\omega \to t_5^1), (\omega \to t_5^2) \to t_5^1}\,(FUN)$$

$$\textit{et} \quad \Pi_3'' = \dfrac{\dfrac{\dfrac{}{f : t_4^1 \to t_5^1 \vdash f : t_4^1 \to t_5^1}\,(ID) \quad \dfrac{}{x : t_4^1 \vdash x : t_4^1}\,(ID)}{f : t_4^1 \to t_5^1, x : t_4^1 \vdash fx : t_5^1}\,(APPL)}{f : t_4^1 \to t_5^1, x : t_4^1 \vdash \lambda z\ fx : \omega \to t_5^1}\,(FUN)$$

$$et \quad \Pi_3''' = \dfrac{\dfrac{\overline{f : t_4^2 \to t_5^2 \vdash f : t_4^2 \to t_5^2}^{(ID)} \qquad \overline{x : t_4^2 \vdash x : t_4^2}^{(ID)}}{f : t_4^2 \to t_5^2, x : t_4^2 \vdash fx : t_5^2}^{(APPL)}}{f : t_4^2 \to t_5^2, x : t_4^2 \vdash \lambda z\ fx : \omega \to t_5^2}^{(FUN)}$$

Il ne reste plus qu'à résoudre les contraintes correspondant au noeud (REC), autrement dit les deux équations :

$$\Theta = \left\{ \begin{array}{lcl} t_4^1 \to t_5^1 & = & t_4^1, t_4^2 \to t_5^1, \\ t_4^2 \to t_5^2 & = & t_4^1, t_4^2 \to t_5^1 \end{array} \right\}$$

Ceci se résout sans difficulté par l'algorithme d'unification avec (par exemple) :

$$\overline{S}_3 = \{t_5^2 \mapsto t_5^1\} :: \{t_4^2 \mapsto t_4^1\}$$

et l'arbre de typage final $\overline{S}_3(\Pi_3)$ devient :

$$\Pi_4 = \dfrac{\dfrac{\dfrac{\Pi_4' \qquad \Pi_4'' \qquad \Pi_4''}{f : t_4^1 \to t_5^1, f : t_4^1 \to t_5^1, x : t_4^1, x : t_4^1 \vdash \Delta\ (\lambda z\ fx) : t_5^1}^{(APPL)}}{f : t_4^1 \to t_5^1, f : t_4^1 \to t_5^1 \vdash \lambda x\ (\Delta\ (\lambda z\ fx)) : t_4^1, t_4^1 \to t_5^1}^{(FUN)}}{\vdash \mu f \lambda x\ (\Delta\ (\lambda z\ fx)) : t_4^1, t_4^1 \to t_5^1}^{(REC)}$$

$$avec \quad \Pi_4' = \dfrac{\dfrac{\overline{y : \omega \to t_5^1 \vdash y : \omega \to t_5^1}^{(ID)} \qquad \overline{y : \omega \to t_5^1 \vdash y : \omega \to t_5^1}^{(ID)}}{y : \omega \to t_5^1, y : \omega \to t_5^1 \vdash y\ y : t_5^1}^{(APPL)}}{\vdash \Delta : (\omega \to t_5^1), (\omega \to t_5^1) \to t_5^1}^{(FUN)}$$

$$et \quad \Pi_4'' = \dfrac{\dfrac{\overline{f : t_4^1 \to t_5^1 \vdash f : t_4^1 \to t_5^1}^{(ID)} \qquad \overline{x : t_4^1 \vdash x : t_4^1}^{(ID)}}{f : t_4^1 \to t_5^1, x : t_4^1 \vdash fx : t_5^1}^{(APPL)}}{f : t_4^1 \to t_5^1, x : t_4^1 \vdash \lambda z\ fx : \omega \to t_5^1}^{(FUN)}$$

On remarquera que Π_3'' et Π_3''' sont maintenant devenus des sous-arbres identiques Π_4''. On pourra aussi noter qu'au niveau du noeud (REC), la loi d'idempotence $\equiv$ est bien nécessaire.

En guise de conclusion, on pourrait également envisager d'autoriser la congruence $\equiv$ partout dans l'arbre, et même de contracter les liaisons dans l'environnement ainsi que les sous-arbres identiques. Appliqué à notre exemple, ceci nous permettrait d'écrire l'arbre simplifié :

$$\Pi_5 = \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{y : \omega \to t_5^1 \vdash y : \omega \to t_5^1}^{(ID)} \qquad \overline{y : \omega \to t_5^1 \vdash y : \omega \to t_5^1}^{(ID)}}{y : \omega \to t_5^1 \vdash y\ y : t_5^1}^{(APPL)}}{\vdash \Delta : (\omega \to t_5^1) \to t_5^1}^{(FUN)} \qquad \Pi_4''}{f : t_4^1 \to t_5^1, x : t_4^1 \vdash \Delta\ (\lambda z\ fx) : t_5^1}^{(APPL)}}{f : t_4^1 \to t_5^1 \vdash \lambda x\ (\Delta\ (\lambda z\ fx)) : t_4^1 \to t_5^1}^{(FUN)}}{\vdash \mu f \lambda x\ (\Delta\ (\lambda z\ fx)) : t_4^1 \to t_5^1}^{(REC)}$$

Exemple 13.2 *Voici un exemple plus réaliste :*

$$M = \mu f\ \lambda l\ (g\ l\ (f\ (h\ l)))$$

que l'on peut interpréter comme la forme générale d'un récurseur sur les listes, suivant la façon dont sont choisies les fonctions g et h. Par exemple, avec $h = tl$ (ou cdr) et

$$g = \lambda l\ \lambda n\ \mathit{if}\ \mathit{null}\ l\ \mathit{then}\ 0\ \mathit{else}\ n+1$$

(et en prenant une sémantique d'évaluation paresseuse pour le if), le terme M ci-dessus est une fonction qui retourne la longueur d'une liste.

En appliquant les algorithmes d'inférence et d'unification, on trouve le typage suivant pour M :

$$g : t_{list} \to t \to t,\ \ h : t_{list} \to t_{list} \vdash M : t_{list}, t_{list} \to t$$

ce qui est bien le typage attendu pour un récurseur sur les listes (à la contraction près).

Exemple 13.3 *Pour terminer, nous allons considérer un exemple qui ne se comporte pas comme nous pourrions l'espérer : $M = \mu f\ [\Delta, f\ y]$ avec $\Delta = \lambda x\ (xx)$* [31]*. Ce terme étant en forme normale, la première partie de l'algorithme d'inférence est immédiate, et retourne le squelette (nous ne détaillons pas le typage de Δ) :*

$$\Pi = \dfrac{\dfrac{\vdash \Delta : (t_0 \to t_1), t_0 \to t_1 \qquad \dfrac{\dfrac{}{f : t_2 \to t_3 \vdash f : t_2 \to t_3}(ID) \qquad \dfrac{}{y : t_2 \vdash y : t_2}(ID)}{f : t_2 \to t_3, y : t_2 \vdash f\ y : t_3}(APPL)}{f : t_2 \to t_3, y : t_2 \vdash [\Delta, f\ y] : (t_0 \to t_1), t_0 \to t_1}(FORGET)}{y : t_2 \vdash \mu f\ [\Delta, f\ y] : (t_0 \to t_1), t_0 \to t_1}(REC)$$

Dans la deuxième phase, nous sommes donc amenés à résoudre l'unique équation de récursion :

$$t_2 \to t_3 = (t_0 \to t_1), t_0 \to t_1$$

qui conduit à $t_2 = t_0 \to t_1 = t_0$ et donc à un échec puisqu'on est face à une substitution récursive. Le terme M n'est donc pas typable par cet algorithme. Pourtant, il est tout à fait typable si on duplique le sous-arbre correspondant à y et qu'on effectue les substitutions nécessaires. Plus précisément, la substitution à appliquer est :

$$\overline{S} = \{t_2^1 \mapsto t_0 \to t_1\} :: \{t_2^2 \mapsto t_0\} :: \{t_3 \mapsto t_1\} :: D(2, \{t_2\})$$

et l'on obtient alors l'arbre final $\overline{S}(\Pi)$ suivant (en abrégeant $\overline{\tau} = (t_0 \to t_1), t_0 \to t_1$) :

$$\dfrac{\dfrac{\vdash \Delta : \overline{\tau} \qquad \dfrac{\dfrac{}{f : \overline{\tau} \vdash f : \overline{\tau}}(ID) \qquad \dfrac{}{y : t_0 \to t_1 \vdash y : t_0 \to t_1}(ID) \qquad \dfrac{}{y : t_0 \vdash y : t_0}(ID)}{f : \overline{\tau}, y : t_0 \to t_1, y : t_0 \vdash f\ y : t_1}(APPL)}{f : \overline{\tau}, y : t_0 \to t_1, y : t_0 \vdash [\Delta, f\ y] : \overline{\tau}}(FORGET)}{y : t_0 \to t_1, y : t_0 \vdash \mu f\ [\Delta, f\ y] : \overline{\tau}}(REC)$$

qui est bien un arbre de typage valide pour M.

Ce dernier exemple montre que l'algorithme présenté dans ce chapitre donne des résultats dans certains cas, mais qu'il n'est pas complet. Malgré nos efforts, nous n'avons pas (encore) trouvé une méthode générale simple qui permettrait de typer tous les termes récursifs typables.

[31] Pour un exemple équivalent en λ-calcul sans $[\_, \_]$, on pourra considérer $\mu f\ ((\lambda z\ \Delta)(f\ y))$.

Annexe A

Mini-manuel de référence de MlObj

L'interpréteur est téléchargeable à l'adresse suivante :

`http://www-sop.inria.fr/mimosa/Pascal.Zimmer/mlobj.html`

A.1 Ligne de commande

La ligne de commande de MlObj prend la forme suivante :

`mlobj <options> [fichier.mlobj] ...`

Les options possibles sont les suivantes :

- `-v` : donne davantage d'informations en cas d'erreur de typage
- `-no_typing` : désactive la phase de typage ; l'interpréteur devient un simple évaluateur (aucune garantie de sûreté n'est plus donnée bien entendu)
- `-help` ou `---help` : affiche les informations d'usage pour la ligne de commande

Les fichiers donnés en ligne de commande sont chargés au lancement de l'interpréteur, dans l'ordre où ils sont donnés, comme s'ils avaient été tapés interactivement. En cas d'erreur de syntaxe ou de typage, la lecture du fichier est stoppée et l'on passe au suivant. Lorsque tous ont été lus, l'interpréteur continue en mode interactif normal.

A.2 Analyse lexicale

Identifieurs Un identifieur est une suite non vide de caractères, commençant par une lettre ou un underscore _, suivi de lettres, chiffres, underscores ou apostrophes `'`.

Par exception, les mots-clés suivants sont réservés et ne peuvent pas être utilisés comme identifieurs : `let`, `rec`, `and`, `type`, `unsafe`, `use`, `in`, `fun`, `if`, `then`, `else`, `mixin`, `end`, `var`, `cst`, `meth`, `inherit`, `as`, `without`, `rename`.

Labels Les labels ont la même syntaxe que les identifieurs.

Entiers Un entier est une suite non vide de chiffres, éventuellement précédée du caractère `-`.

Chaîne de caractères Une chaîne de caractères (*string*) est une suite de caractères commençant et terminant par des guillemets `"`. Les caractères intermédiaires peuvent être quelconques, excepté les guillemets.

Variables de type et de rangée Une variable de type commence par une apostrophe `'`, suivie d'une lettre minuscule, puis d'une suite quelconque de lettres, chiffres, underscores et apostrophes. Une variable de rangée commence par une apostrophe `'`, suivie d'une lettre majuscule, puis d'une suite quelconque de lettres, chiffres, underscores et apostrophes.

Espaces, etc... Les caractères espaces, retour à la ligne et tabulations sont ignorés.

Commentaires On utilise une syntaxe à la C. Les caractères suivant `//` sont ignorés jusqu'au prochain retour à la ligne. Les caractères entre `/*` et `*/` sont également ignorés (attention à ne pas utiliser ces séquences dans une chaîne de caractères).

A.3 Analyse syntaxique

Cette section décrit la grammaire utilisée par MLObj.

Commandes et instructions Une commande à l'interpréteur se termine toujours par `;;`. C'est seulement lorsque ce marqueur de fin est lu que l'évaluation de l'instruction est lancée. Si une expression seule est donnée, elle est typée, puis évaluée et son résultat est imprimé. Les instructions introduites par `let` et `let rec` sont traitées de la même manière, mais leur(s) résultat(s) est(sont) stocké(s) dans l'environnement global et peuvent être accédé(s) par la suite. Les instructions `#type` et `#unsafe` ne réalisent respectivement que le typage et l'évaluation de l'expression donnée. Ces commandes sont fournies dans un but de mise au point et peuvent bien évidemment conduire au plantage de l'interpréteur en ce qui concerne `#unsafe`. On notera qu'afin de garantir au maximum l'intégrité des évaluations suivantes, seules des expressions peuvent être évaluées par `#type` et `#unsafe` (pas de stockage possible, sauf à contourner la restriction en effectuant une affectation à une référence bien entendu). Enfin, l'instruction `#use` suivie d'un nom de fichier permet de charger un fichier source dans l'interpréteur.

command ::= *instr* `;;`
instr ::= `let` *let_binding*
| `let rec` *let_binding*
| *expr*
| `#type` *expr*
| `#unsafe` *expr*
| `#use` *string*

Liaisons A la suite d'un `let` ou `let rec`, on trouve des liaisons (au moins une) de la forme `f x1 x2 ... = e` séparées par le mot-clé `and`. `f` désigne le nom de la variable pour la liaison, `x1`, `x2`... sont les éventuels arguments de la fonction définie (voir plus loin les remarques sur le sucre syntaxique). Ces arguments sont soit des variables simples, soit des variables avec une information de type (qui sera unifiée avec le type inféré lors du typage).

let_binding ::= *ident params* `=` *expr*
| *ident params* `=` *expr* `and` *let_binding*
params ::= *typable_id params*
|
typable_id ::= *ident*
| `(` *ident* `:` *typexpr* `)`

Expressions La grammaire des expressions ci-dessous ne demande pas de commentaires particuliers ; elle est identique ou similaire à celle de la plupart des langages fonctionnels. De même que pour les liaisons, les arguments d'une fonction `fun` peuvent contenir des informations de typage. Les mixins (voir plus bas) sont introduits par les mots-clés `mixin` et `end`.

$$
\begin{array}{rcl}
\textit{expr} & ::= & \texttt{let}\ \textit{let\_binding}\ \texttt{in}\ \textit{expr} \\
& | & \texttt{let rec}\ \textit{let\_binding}\ \texttt{in}\ \textit{expr} \\
& | & \texttt{fun}\ \textit{typable\_id}\ \textit{params}\ \texttt{->}\ \textit{expr} \\
& | & \texttt{if}\ \textit{expr}\ \texttt{then}\ \textit{expr}\ \texttt{else}\ \textit{expr} \\
& | & \texttt{mixin}\ \textit{mixin}\ \texttt{end} \\
& | & \textit{expr}\ \textit{infix\_op}\ \textit{expr} \\
& | & \textit{appl} \\
\textit{appl} & ::= & \textit{appl}\ \textit{term} \\
& | & \textit{term}
\end{array}
$$

Termes De même, la grammaire des termes n'appelle que peu de remarques. Les enregistrements sont introduits par des accolades. Il est possible de donner une information de type sous la forme `( e : tau )`. Le type inféré sera alors unifié avec le type donné ; ceci permet dans certains cas de limiter le type inféré automatiquement lorsqu'il est trop générique par rapport à l'usage que l'on souhaite faire de l'expression. Les opérateurs infixes, ainsi que `!`, peuvent être accédés en tant que fonction préfixe classique (par exemple, `(+)` pour la fonction addition). Les listes sont introduites par des crochets et leurs éléments sont séparés par des virgules. Enfin, on retrouve les opérateurs de sélection et de restriction d'un champ pour les enregistrements et les mixins.

$$
\begin{array}{rcl}
\textit{term} & ::= & \textit{ident} \\
& | & \textit{entier} \\
& | & \textit{string} \\
& | & \{\ \textit{record}\ \} \\
& | & \texttt{(}\ \textit{expr}\ \texttt{)} \\
& | & \texttt{(}\ \textit{expr}\ \texttt{:}\ \textit{typexpr}\ \texttt{)} \\
& | & \texttt{(}\ \textit{op}\ \texttt{)} \\
& | & \texttt{[}\ \textit{list}\ \texttt{]} \\
& | & \textit{term}\ \texttt{.}\ \textit{label} \\
& | & \textit{term}\ \texttt{\#}\ \textit{label} \\
& | & \textit{term} \setminus \textit{label} \\
& | & \texttt{!}\ \textit{term} \\
\textit{list} & ::= & \textit{expr}\ \texttt{,}\ \textit{list} \\
& | & \textit{expr} \\
& | &
\end{array}
$$

Opérateurs et relations infixes La grammaire comprend une batterie d'opérateurs classiques pour un langage de programmation. Ces opérateurs sont détaillés à la Section A.6.

$$
\begin{array}{rcl}
\textit{infix\_op} & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{:=} \mid \texttt{==} \mid \texttt{<>} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{::} \mid \texttt{;} \mid \texttt{@} \mid \texttt{\^{}} \\
\textit{op} & ::= & \textit{infix\_op} \mid \texttt{!} \mid
\end{array}
$$

Enregistrements Les enregistrements sont une suite de définitions de champs de la forme `label x1 x2 ... = e` ou de modifications de champs de la forme `label x1 x2 ... <- e`. Optionnellement, en tête, peut se trouver une expression qui désigne l'enregistrement dont on part pour réaliser

les extensions et modifications (sinon on part implicitement d'un enregistrement vide).

record ::= *label params* `=` *expr*
| *expr* `,` *label params* `=` *expr*
| *record* `,` *label params* `=` *expr*
| *expr* `,` *label params* `<-` *expr*
| *record* `,` *label params* `<-` *expr*
|

Mixins Les mixins sont une suite d'instructions introduites par un jeu de mots-clés différents. Le mot-clé `inherit` introduit une expression s'évaluant en un mixin dont on va hériter tous les champs. Le mot-clé `var` introduit une nouvelle variable modifiable. Le mot-clé `cst` introduit un champ non modifiable. La séquence `meth label(super,self) ...` désigne une méthode de nom `label` ; les arguments `super` et `self` désigne respectivement le mixin avant modification et le mixin courant et sont destinés à être utilisés dans le corps de la méthode. Cette méthode est soit déclarée nouvelle (par `=`), soit comme remplaçant d'une méthode existante de même nom (par `<-`). Le mot-clé `without` supprime un champ. Enfin, le mot-clé `rename` permet le renommage d'un champ existant.

mixin ::= *mixin* `inherit` *appl*
| *mixin* `var` *label params* `=` *expr*
| *mixin* `cst` *label params* `=` *expr*
| *mixin* `meth` *label* `(` *typable_id* `,` *typable_id* `)` *params* `=` *expr*
| *mixin* `meth` *label* `(` *typable_id* `,` *typable_id* `)` *params* `<-` *expr*
| *mixin* `without` *label*
| *mixin* `rename` *label* `as` *label*
|

Expressions de type Une expression de type est soit une constante, soit une variable de type, soit un type fonction, soit le type d'un enregistrement, soit un type liste, soit un type référence. Le type d'un enregistrement commence optionnellement par une variable de rangée et est suivi des noms de champs avec leurs types respectifs. On notera que cette grammaire ne fait pas mention des degrés pour les fonctions et qu'il n'est donc pas possible en l'état de l'interpréteur de "forcer" ce paramètre (pour chaque type fonctionnel entré, l'analyseur syntaxique introduit implicitement une variable de degré fraîche ; nous n'avons pas trouvé d'utilisation pratique qui justifierait de proposer cette fonctionnalité).

typexpr ::= `int` | `bool` | `unit` | `string`
| *typ_ident*
| *typexpr* `->` *typexpr*
| `(` *typexpr* `)`
| `{` *typrecord* `}`
| `{` *row_ident* `,` *typrecord* `}`
| *typexpr* `ref`
| *typexpr* `list`
typrecord ::= *label* `:` *typexpr*
| *label* `:` *typexpr* `,` *typrecord*
|

A.4 Sucre syntaxique

Comme c'est le cas en Objective Caml, il est possible de définir des fonctions de manière "compacte". Ainsi, les instructions

```
fun x1 ... xn -> expr
```

```
let (rec) f x1 ... xn = expr in ...
```

```
{ ... , f x1 ... xn = expr , ... }
```

sont respectivement équivalentes à

```
fun x1 -> ... -> fun xn -> expr
```

```
let (rec) f = fun x1 -> ... -> fun xn -> expr in ...
```

```
{ ... , f = fun x1 -> ... fun xn -> expr , ... }
```

Ce "sucre syntaxique" est admis devant tous les opérateurs `=`, `<-` et `->`, autrement dit dans les définitions globales et locales introduites par `let (rec)`, dans les fonctions introduites par `fun`, dans les champs d'enregistrements extensibles, dans les champs modifiables et non modifiables de mixins, et dans leurs définitions de méthodes.

Par ailleurs, la redéfinition de champs pour un enregistrement est directement convertie en son équivalent restriction, puis extension. Par exemple :

```
{ expr, foo x <- x * x }
```

est transformé en :

```
{ expr \ foo, foo x = x * x }
```

Enfin, un autre ensemble de transformations syntaxiques a lieu en ce qui concerne les mixins, afin de les traduire dans le sous-langage composé seulement du cœur fonctionnel, des références et des enregistrements. Cette transformation est décrite à la Section 4.2.

A.5 Priorité des opérateurs

Cette table montre la priorité relative et l'associativité pour les opérateurs du langage, en commençant par la priorité la plus élevée.

Construction ou opérateur	Associativité
\	gauche
`. #`	gauche
`!`	droite
`* / %`	gauche
`+ -`	gauche
`::`	droite
`@ ^`	droite
`== <> < <= > >=`	aucune
`&&`	gauche
`||`	gauche
`:=`	droite
`else`	aucune
`;`	droite
`->`	droite
`in`	aucune

A.6 Librairie prédéfinie

Voici la liste des fonctions et des constantes prédéfinies dans le système, avec leurs types. Tous les opérateurs binaires sont accessibles en tant que fonction sous leur forme préfixée en les notant entre parenthèses (par exemple, `(+)` pour la fonction addition).

A.6.1 Unit

`() : unit`
La valeur unit (void).

A.6.2 Arithmétique

```
(+) : int ^a-> int ^0-> int
(-) : int ^a-> int ^0-> int
(*) : int ^a-> int ^0-> int
(/) : int ^a-> int ^0-> int
(%) : int ^a-> int ^0-> int
```

Les opérateurs standards pour l'addition, la soustraction, la multiplication, la division et le modulo.

```
(<) : int ^a-> int ^0-> bool
(<=) : int ^a-> int ^0-> bool
(>) : int ^a-> int ^0-> bool
(>=) : int ^a-> int ^0-> bool
(==) : int ^a-> int ^0-> bool
(<>) : int ^a-> int ^0-> bool
```

Les relations de comparaison entre entiers.

A.6.3 Booléens

`true : bool`
`false : bool`
Les constantes booléennes.

`(&&) : bool ^a-> bool ^0-> bool`
`(||) : bool ^a-> bool ^0-> bool`
`not : bool ^0-> bool`
Les opérateurs booléens standards (conjonction, disjonction et négation).

A.6.4 Références

`ref : 'a ^b-> 'a ref`
Crée une nouvelle référence.

`(!) : 'a ref ^0-> 'a`
Donne la valeur d'une référence.

`(:=) : 'a ref ^b-> 'a ^c-> unit`
Change la valeur d'une référence.

A.6.5 Listes

`[] : 'a list`
La liste vide.

`(::) : 'a ^b-> 'a list ^0-> 'a list`
L'opérateur cons.

`hd : 'a list ^0-> 'a`
`tl : 'a list ^0-> 'a list`
Les opérateurs tête et queue.

`null : 'a list ^0-> bool`
Teste si une liste est vide.

`(@) : 'a list ^b-> 'a list ^0-> 'a list`
Concaténation de listes.

A.6.6 Autres

`(^) : string ^a-> string ^0-> string`
Concaténation de chaînes de caractères.

`(;) : unit ^b-> 'a ^0-> 'a`
Séquence.

`print_int : int ^0-> unit`

`print_string : string ^0-> unit`

`print_newline : unit ^0-> unit`

Ecriture d'un entier, d'une chaîne de caractères ou d'un retour à la ligne sur la sortie standard.

`quit : unit ^0-> 'a`

Quitte la session.

A.7 Sémantique big-step d'origine

Pour mémoire et pour information, nous reproduisons ici la sémantique "big-step" utilisée dans les premières versions de l'interpréteur. D'un point de vue performance, celle-ci n'est pas plus lente que la machine abstraite du Chapitre 5, car elle peut être implémentée directement de façon récursive terminale...

A.7.1 Syntaxe

La syntaxe est à peu de choses près la même que celle utilisée dans la version actuelle de MLOBJ (il manque les listes et les liaisons multiples).

$$
\begin{array}{rcl}
M, N & ::= & MN \\
& | & \texttt{let x =}\ M\ \texttt{in}\ N \\
& | & \texttt{let rec x =}\ M\ \texttt{in}\ N \\
& | & \texttt{fun x}\ \rightarrow M \\
& | & \texttt{x} \\
& | & i \in Int \\
& | & s \in String \\
& | & \texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ N' \\
& | & \{\ \} \\
& | & \{\ M\texttt{, l =}\ N\ \} \\
& | & M\texttt{.l} \\
& | & M\backslash\texttt{l}
\end{array}
$$

A.7.2 Valeurs à l'exécution

On suppose qu'on a un ensemble *Location* de *cases mémoires*. Celles-ci serviront aussi bien pour les références créées par `ref` que pour les valeurs récursives. L'ensemble des valeurs retournées

par l'évaluation d'un terme est alors (où $(\mathtt{x}, M, E)$ désigne une clôture) :

$$\begin{array}{rcl} V & ::= & (\mathtt{x}, M, E) \\ & | & u \in \mathit{Location} \\ & | & \{l_1 = V_1, \ldots, l_n = V_n\} \text{ sans ordre et } l_i \neq l_j \\ & | & \mathit{rec}(u) \\ & | & \mathtt{()} \in \mathit{Unit} \\ & | & i \in \mathit{Int} \\ & | & b \in \mathit{Bool} \\ & | & s \in \mathit{String} \\ & | & \mathit{prim}(f) \text{ avec } f : V \times S \to \mathit{Res} \times S \\ S & ::= & \mathit{Location} \to V \cup \{\bullet\} \\ E & ::= & \emptyset \\ & | & E, \mathtt{x} = V \\ \mathit{Res} & ::= & V \\ & | & \mathtt{err} \end{array}$$

Une *mémoire* S est une application partielle de domaine fini, d'un ensemble infini de cases mémoires vers l'ensemble des valeurs (ou la valeur spéciale $\bullet$ (pointeur null), utilisée seulement pour `let rec`). Il est donc toujours possible de trouver une case mémoire $u \notin \mathit{dom}(S)$ (on dira que u est *fraîche*). La mise à jour d'une mémoire est définie par :

$$S\{^u/_V\} = S' \quad \text{où} \quad \begin{cases} S'(v) = S(v) & \text{pour } v \neq u \\ S'(u) = V \end{cases}$$

avec $\mathit{dom}(S') = \mathit{dom}(S)$ si $u \in \mathit{dom}(S)$ et $\mathit{dom}(S') = \mathit{dom}(S) \cup \{u\}$ sinon.

Un environnement E est une suite de liaisons variables-valeurs. Enfin, un résultat Res est soit une valeur, soit `err` pour signifier une erreur d'exécution.

A.7.3 Sémantique big-step

La forme générale est :

$$E; S \vdash M \Rightarrow \mathit{Res}, S'$$

i.e. l'évaluation du terme M dans l'environnement E et la mémoire S, produit le résultat Res pour une mémoire modifiée S'.

Les règles de réduction sont listées ci-dessous ; nous ne les détaillerons pas.

Recherche dans l'environnement

$$\frac{}{E, \mathtt{x} = V; S \vdash \mathtt{x} \Rightarrow V, S}$$

$$\frac{E; S \vdash \mathtt{x} \Rightarrow V, S' \qquad \mathtt{x} \neq \mathtt{y}}{E, \mathtt{y} = V'; S \vdash \mathtt{x} \Rightarrow V, S'}$$

$$\frac{E; S \vdash \mathtt{x} \Rightarrow \mathit{rec}(u), S' \qquad S'(u) \neq \bullet}{E; S \vdash \mathtt{x} \Rightarrow S'(u), S'}$$

Règles principales

$$\frac{E; S \vdash M \Rightarrow (\mathtt{x}, M', E'), S' \qquad E; S' \vdash N \Rightarrow V, S'' \qquad E', \mathtt{x} = V; S'' \vdash M' \Rightarrow V', S'''}{E; S \vdash MN \Rightarrow V', S'''}$$

$$\frac{E; S \vdash M \Rightarrow prim(f), S' \qquad E; S' \vdash N \Rightarrow V, S''}{E; S \vdash MN \Rightarrow f(V, S'')} \quad \text{lorsque } f(V, S'') = (V', S''')$$

$$\frac{E; S \vdash M \Rightarrow V, S' \qquad E, \mathtt{x} = V; S' \vdash N \Rightarrow V', S''}{E; S \vdash \mathtt{let\ x\ =}\ M\ \mathtt{in}\ N \Rightarrow V', S''}$$

$$\frac{u \text{ fraîche pour } S \qquad E, \mathtt{x} = rec(u); S\{^u/_\bullet\} \vdash M \Rightarrow V, S' \qquad E, \mathtt{x} = rec(u); S'\{^u/_V\} \vdash N \Rightarrow V', S''}{E; S \vdash \mathtt{let\ rec\ x\ =}\ M\ \mathtt{in}\ N \Rightarrow V', S''}$$

$$\frac{}{E; S \vdash \mathtt{fun\ x}\ \rightarrow M \Rightarrow (\mathtt{x}, M, E), S}$$

$$\frac{i \in Int}{E; S \vdash i \Rightarrow i, S}$$

$$\frac{s \in String}{E; S \vdash s \Rightarrow s, S}$$

$$\frac{E; S \vdash M \Rightarrow true, S' \qquad E; S' \vdash N \Rightarrow V, S''}{E; S \vdash \mathtt{if}\ M\ \mathtt{then}\ N\ \mathtt{else}\ N' \Rightarrow V, S''}$$

$$\frac{E; S \vdash M \Rightarrow false, S' \qquad E; S' \vdash N' \Rightarrow V, S''}{E; S \vdash \mathtt{if}\ M\ \mathtt{then}\ N\ \mathtt{else}\ N' \Rightarrow V, S''}$$

$$\frac{}{E; S \vdash \{\ \} \Rightarrow \{\}, S}$$

$$\frac{E; S \vdash M \Rightarrow \{l_i = V_i\}, S' \qquad E; S' \vdash N \Rightarrow V, S'' \qquad l \neq l_i}{E; S \vdash \{\ M\mathtt{,\ l\ =}\ N\} \Rightarrow \{l_i = V_i, l = V\}, S''}$$

$$\frac{E; S \vdash M \Rightarrow \{l_i = V_i\}, S'}{E; S' \vdash M\mathtt{.l}_i \Rightarrow V_i, S'}$$

$$\frac{E; S \vdash M \Rightarrow \{l_i = V_i\}, S'}{E; S \vdash M\backslash\mathtt{l}_i \Rightarrow \{l_1 = V_1, \ldots, l_{i-1} = V_{i-1}, l_{i+1} = V_{i+1}, \ldots, l_n = V_n\}, S'}$$

A.7.4 Erreurs à l'exécution

Axiomes non capturés par le système de type

Identificateur non défini (pourrait être détecté par une analyse statique) :

$$\frac{}{\emptyset; S \vdash \mathtt{x} \Rightarrow \mathtt{err}, S}$$

Vraie erreur à l'exécution (par exemple 1/0) :

$$\frac{E; S \vdash M \Rightarrow prim(f), S' \qquad E; S' \vdash N \Rightarrow V, S''}{E; S \vdash MN \Rightarrow f(V, S'')} \quad \text{lorsque } f(V, S'') = (\mathtt{err}, S''')$$

Axiomes capturés par le système de type

$$\frac{E;S \vdash M \Rightarrow V, S' \qquad V \neq (\mathtt{x}, M', E') \qquad V \neq prim(f)}{E;S \vdash MN \Rightarrow \mathtt{err}, S'}$$

$$\frac{E;S \vdash M \Rightarrow V, S' \qquad V \notin Bool}{E;S \vdash \mathtt{if}\ M\ \mathtt{then}\ N\ \mathtt{else}\ N' \Rightarrow \mathtt{err}, S'}$$

$$\frac{E;S \vdash M \Rightarrow V, S' \qquad V \neq \{l_i = V_i\}}{E;S \vdash \{M\mathtt{,\ l\ =}\ N\} \Rightarrow \mathtt{err}, S'}$$

$$\frac{E;S \vdash M \Rightarrow \{l_i = v_i\}, S' \qquad E;S' \vdash N \Rightarrow V, S'' \qquad l = l_i}{E;S \vdash \{M\mathtt{,\ l\ =}\ N\} \Rightarrow \mathtt{err}, S''}$$

$$\frac{E;S \vdash M \Rightarrow V, S' \qquad V \neq \{l_i = V_i\}}{E;S \vdash M\mathtt{.l} \Rightarrow \mathtt{err}, S'}$$

$$\frac{E;S \vdash M \Rightarrow \{l_i = V_i\}, S' \qquad l \neq l_i}{E;S \vdash M\mathtt{.l} \Rightarrow \mathtt{err}, S'}$$

$$\frac{E;S \vdash M \Rightarrow V, S' \qquad V \neq \{l_i = V_i\}}{E;S \vdash M\backslash\mathtt{l} \Rightarrow \mathtt{err}, S'}$$

$$\frac{E;S \vdash M \Rightarrow \{l_i = V_i\}, S' \qquad l \neq l_i}{E;S \vdash M\backslash\mathtt{l} \Rightarrow \mathtt{err}, S'}$$

Propagation des erreurs

$$\frac{E;S \vdash \mathtt{x} \Rightarrow \mathtt{err}, S' \qquad \mathtt{x} \neq \mathtt{y}}{E, \mathtt{y} = V'; S \vdash \mathtt{x} \Rightarrow \mathtt{err}, S'}$$

$$\frac{E;S \vdash M \Rightarrow \mathtt{err}, S'}{E;S \vdash MN \Rightarrow \mathtt{err}, S'}$$

$$\frac{E;S \vdash M \Rightarrow (\mathtt{x}, M', E'), S' \qquad E;S' \vdash N \Rightarrow \mathtt{err}, S''}{E;S \vdash MN \Rightarrow \mathtt{err}, S''}$$

$$\frac{E;S \vdash M \Rightarrow prim(f), S' \qquad E;S' \vdash N \Rightarrow \mathtt{err}, S''}{E;S \vdash MN \Rightarrow \mathtt{err}, S''}$$

$$\frac{E;S \vdash M \Rightarrow (\mathtt{x}, M', E'), S' \qquad E;S' \vdash N \Rightarrow V, S'' \qquad E', \mathtt{x} = V; S'' \vdash M' \Rightarrow \mathtt{err}, S'''}{E;S \vdash MN \Rightarrow \mathtt{err}, S'''}$$

$$\frac{E;S \vdash M \Rightarrow \mathtt{err}, S'}{E;S \vdash \mathtt{let\ x\ =}\ M\ \mathtt{in}\ N \Rightarrow \mathtt{err}, S'}$$

$$\frac{E;S \vdash M \Rightarrow V, S' \qquad E, \mathtt{x} = V; S' \vdash N \Rightarrow \mathtt{err}, S''}{E;S \vdash \mathtt{let\ x\ =}\ M\ \mathtt{in}\ N \Rightarrow \mathtt{err}, S''}$$

$$\frac{u \text{ fraîche pour } S \qquad E, \mathtt{x} = rec(u); S\{^u/_\bullet\} \vdash M \Rightarrow \mathtt{err}, S'}{E;S \vdash \mathtt{let\ rec\ x\ =}\ M\ \mathtt{in}\ N \Rightarrow \mathtt{err}, S'}$$

$$\frac{u \text{ fraîche pour } S \qquad E, \mathtt{x} = rec(u); S\{^u/_\bullet\} \vdash M \Rightarrow V, S' \qquad E, \mathtt{x} = rec(u); S'\{^u/_V\} \vdash N \Rightarrow \mathtt{err}, S''}{E;S \vdash \mathtt{let\ rec\ x\ =}\ M\ \mathtt{in}\ N \Rightarrow \mathtt{err}, S''}$$

$$\frac{E;S \vdash M \Rightarrow \mathtt{err}, S'}{E;S \vdash \mathtt{if}\ M\ \mathtt{then}\ N\ \mathtt{else}\ N' \Rightarrow \mathtt{err}, S'}$$

$$\frac{E; S \vdash M \Rightarrow true, S' \qquad E; S' \vdash N \Rightarrow \texttt{err}, S''}{E; S \vdash \texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ N' \Rightarrow \texttt{err}, S''}$$

$$\frac{E; S \vdash M \Rightarrow false, S' \qquad E; S' \vdash N' \Rightarrow \texttt{err}, S''}{E; S \vdash \texttt{if}\ M\ \texttt{then}\ N\ \texttt{else}\ N' \Rightarrow \texttt{err}, S''}$$

$$\frac{E; S \vdash M \Rightarrow \texttt{err}, S'}{E; S \vdash \{\ M\texttt{, l = }N\} \Rightarrow \texttt{err}, S'}$$

$$\frac{E; S \vdash M \Rightarrow \{l_i = V_i\}, S' \qquad E; S' \vdash N \Rightarrow \texttt{err}, S''}{E; S \vdash \{\ M\texttt{, l = }N\} \Rightarrow \texttt{err}, S''}$$

$$\frac{E; S \vdash M \Rightarrow \texttt{err}, S'}{E; S \vdash M\texttt{.l} \Rightarrow \texttt{err}, S'}$$

$$\frac{E; S \vdash M \Rightarrow \texttt{err}, S'}{E; S \vdash M\backslash\texttt{l} \Rightarrow \texttt{err}, S'}$$

A.7.5 Quelques primitives prédéfinies dans l'environnement initial

Pour créer et manipuler les références :

$$ref(V, S) = (u, S\{^u/_V\}) \quad \text{avec } u \text{ fraîche pour } S$$

$$!(V, S) = \begin{cases} (S(u), S) & \text{si } V = u \in dom(S) \\ (\texttt{err}, S) & \text{sinon} \end{cases}$$

$$:= (V, S) = \begin{cases} (prim(:=_u), S) & \text{si } V = u \in dom(S) \\ (\texttt{err}, S) & \text{sinon} \end{cases}$$

$$\text{où } :=_u (V, S) = (\texttt{()}, S\{^u/_V\})$$

Addition :

$$+(V, S) = \begin{cases} (prim(+_i), S) & \text{si } V = i \in Int \\ (\texttt{err}, S) & \text{sinon} \end{cases}$$

$$\text{où } +_i (V, S) = \begin{cases} (i + j, S) & \text{si } V = j \in Int \\ (\texttt{err}, S) & \text{sinon} \end{cases}$$

et de même pour les autres fonctions arithmétiques, booléennes et de comparaison...

Annexe B

Travaux sur les ambients

Cette Annexe résume brièvement un autre pan de mes travaux de recherche, effectué avant et pendant la thèse, et consacré au modèle des ambients.

Avec la croissance rapide des réseaux informatiques s'est posé le problème de la conception de langages de programmation adaptés à cet environnement et d'un cadre formel permettant de le décrire. En effet, la capacité à *exécuter du code en parallèle*, à *migrer du code*, voire à *migrer le support d'exécution* (i.e. un portable), ne saurait être modélisée par les formalismes habituels (essentiellement à base de λ-calcul), plus aptes à décrire une exécution mono-thread. Les premières réponses furent CCS et le π-calcul [Mil91], inventés par Robin Milner, des langages simples qui permettent l'exécution de différents processus concurrents et la synchronisation ou l'échange d'informations par l'intermédiaire de canaux nommés.

Ce calcul a fait depuis de nombreux émules ; l'un d'eux est le calcul des *ambients*, un modèle inventé par Luca Cardelli en 1998 [CG98], qui ajoute au π-calcul la notion de *mobilité*. Informellement, un ambient est une "boîte" dans laquelle s'exécutent des processus, pouvant contenir d'autres ambients, et capable de migrer dans un sous-ambient ou à l'extérieur de l'ambient englobant. En donnant une interprétation correcte des entités manipulées, on peut montrer comment des notions centrales des réseaux, comme les domaines, les pare-feux et l'envoi de messages, sont modélisées de façon simple dans un formalisme unique. Ceci facilite d'autant leur étude et permet d'obtenir des résultats généraux. Depuis 1998, un grand nombre de travaux et de variantes des ambients ont été proposés par la communauté informatique, auxquels j'ai participé pour certains d'entre eux.

Typage C'est en 1999, à Turin sous la direction de Mariangiola Dezani, que j'ai découvert les ambients. Des systèmes de types pour ce calcul commençaient alors à voir le jour [CG99, CGG99, LS00], dont le but était de garantir certaines propriétés sur les ambients, ceux-ci étant parfois trop génériques pour être réalistes. Entre autres : un ambient a-t-il ou non le droit de se déplacer ? peut-on ouvrir la "boîte" afin de consulter son contenu ? (par exemple, un paquet IP doit pouvoir se déplacer à travers le réseau et être ouvert pour lire ses données ; à l'inverse, un switch dont le travail est de relayer ces paquets doit rester immobile et clos) les informations échangées à l'intérieur d'un même ambient sont-elles toujours de la même nature ? Pouvoir garantir de telles propriétés permet d'étudier plus finement les systèmes.

Les travaux précédents ne proposaient que les systèmes de types seuls, autrement dit la *vérification* d'un arbre de typage complet. Mon travail consista à étendre ces systèmes par une relation de *sous-typage* et de s'en servir pour construire un algorithme d'*inférence de types*.

Publication : FOSSACS'00 [Zim00].

Expressivité Une autre question essentielle concerne l'expressivité du calcul et l'expressivité relative de ses sous-fragments, afin de déterminer quelles sont les primitives véritablement essentielles. Au cours de mon stage de DEA dans l'équipe MIMOSA, nous nous sommes intéressé au calcul des *ambients purs*, obtenu en enlevant les primitives de communication pour ne garder que les primitives de mouvement. Il était déjà connu que ce sous-calcul pouvait simuler une machine de Turing, mais ceci n'est pas significatif : dans un monde distribué par nature, il faut se comparer à d'autres références qu'une machine de Turing mono-thread. J'ai donc pu préciser ce résultat en construisant un encodage compositionnel du π-calcul synchrone dans les ambients purs et en prouvant sa correction par une correspondance opérationnelle.

Publications : EXPRESS'00 [Zim03a], MSCS [Zim03b].

Contrôle de ressources Dans un monde distribué et mobile, il est fréquent que plusieurs agents tentent d'accéder en même temps à certaines ressources disponibles seulement en nombre limité. Certaines situations critiques doivent alors être évitées à tout prix : par exemple, l'utilisation simultanée d'une même ressource par deux agents. D'autres peuvent se révéler très pénalisantes : par exemple, un trop grand nombre de demandes par rapport aux ressources disponibles, conduisant à des situations du type "déni de service". Cette dernière situation peut même devenir fatale pour certaines applications temps réel (imaginons que le bus de données soit accaparé par un sous-système lorsque le centre de guidage doit envoyer des commandes au bras d'un robot-chirurgien ou à l'aiguillage d'une ligne TGV...). Pour toutes ces raisons, il est utile de disposer d'un formalisme permettant de raisonner et de prouver certaines propriétés sur la cohérence d'utilisation des ressources. Dans ce sens, en collaboration avec deux collègues de l'ENS Lyon, nous avons mis au point une variante du calcul des ambients ainsi qu'un nouveau système de types permettant le contrôle de ressources.

Publications : CONCUR'02 [TZH02], IJIS [TZH04].

Implémentation Dès le début, la question de l'implémentation du calcul des ambients de façon efficace s'est révélée un problème délicat, auquel je me suis plusieurs fois attelé dans la pratique. S'il est possible d'écrire facilement une implémentation mono-processus, l'essence même du calcul est d'être distribué sur plusieurs machines reliées par un réseau. Une approche brutale conduit alors à un grand nombre de synchronisations inutiles à travers le réseau, ce qui ralentit énormément l'exécution globale. Ce problème n'a à mon sens pas été résolu de façon satisfaisante à ce jour, mais des progrès ont été accomplis.

En 2000, j'ai collaboré à la mise au point par Davide Sangiorgi et Andrea Valente d'une machine abstraite pour une version des ambients qui garantit (à l'aide d'un système de types *ad hoc*) une exécution mono-thread à l'intérieur de chaque ambient [SV01].

En 2002, j'ai également co-encadré avec Frédéric Boussinot le stage de Damien Pous, étudiant de Licence de l'ENS Lyon. Son travail a débouché sur la première simulation graphique des ambients, basée sur une plate-forme de programmation réactive mise au point au sein de notre projet. L'aspect purement visuel et interactif de l'implémentation en fait un instrument de choix pour l'enseignement, mais aussi un outil simple à utiliser de conception de solutions algorithmiques par ambients. J'ai également été chargé de la rédaction d'une page web présentant ses résultats.

Référence : `http://www-sop.inria.fr/mimosa/ambicobjs/`

Bibliographie

[Aba94] M. Abadi. Baby Modula-3 and a theory of objects. *Journal of Functional Programming*, 4(2) :249–283, 1994.

[AC96] M. Abadi and L. Cardelli. *A theory of objects*. Springer-Verlag, 1996.

[AC98] R.M. Amadio and P.-L. Curien. *Domains and lambda-calculi*. Cambridge University Press, 1998.

[ASU89] A. Aho, R. Sethi, and J. Ullman. *Compilateurs : principes, techniques et outils*. Inter-Editions, 1989.

[AZ98] D. Ancona and E. Zucca. A theory of mixin modules : basic and derived operators. *Mathematical Structures in Computer Science*, 8 :401–446, 1998.

[Bar92] H.P. Barendregt. Lambda-calculi with types. In *Handbook of Logic in Computer Science*, volume 2 : Computational Structures, pages 117–309. Oxford University Press, 1992.

[BC90] G. Bracha and W. Cook. Mixin-based inheritance. In *ECOOP/OOPSLA'90*, pages 303–311, 1990.

[BCC+95] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Objects Group, G.T. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3) :221–242, 1995.

[BKN96] R. Bloo, F. Kamareddine, and R.P. Nederpelt. The Barendregt Cube with definitions and generalised reductions. *Information and Computation*, 126(2) :123–143, 1996.

[Bou03] G. Boudol. On strong normalization in the intersection type discipline. In *TLCA'03*, LNCS 2701, pages 60–74, 2003.

[Bou04] G. Boudol. The recursive record semantics of objects revisited. *Journal of Functional Programming*, 2004. Version préliminaire dans *ESOP'*01, LNCS 2028, pages 260–283, 2001.

[BPSM99a] V. Bono, A. Patel, V. Shmatikov, and J. Mitchell. A core calculus of classes and mixins. In *ECOOP'99*, LNCS 1628, pages 43–66, 1999.

[BPSM99b] V. Bono, A. Patel, V. Shmatikov, and J. Mitchell. A core calculus of classes and objects. In *MFPS'99*, volume 20 of *Electronic Notes in Computer Science*, 1999.

[Bru93] K. Bruce. Safe type checking in a statically-typed object-oriented programming language. In *POPL'93*, pages 285–298, 1993.

[BZ02] G. Boudol and P. Zimmer. Recursion in the call-by-value lambda-calculus. In *Proceedings of FICS 2002 (Fixed Points in Computer Science)*, BRICS Notes Series NS-02-2, 2002.

[Car88] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76, 1988.

[Car02] S. Carlier. Polar type inference with intersection types and ω. *ENTCS*, 70(1), 2002.

[CDC80] M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame Journal of Formal Logic*, 21 :685–693, 1980.

[CDCS79] M. Coppo, M. Dezani-Ciancaglini, and P. Sallé. Functional characterization of some semantic equalities inside lamda-calculus. In *ICALP Graz*, volume 71 of *LNCS*, pages 133–146, 1979.

[CDCV80] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Principal type schemes and λ-calculus semantics. In *Hindley and Seldin [HS80]*, pages 535–560, 1980.

[CG98] L. Cardelli and A.D. Gordon. Mobile Ambients. In *Proceedings FoSSaCS'98*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer Verlag, 1998.

[CG99] L. Cardelli and A. D. Gordon. Types for Mobile Ambients. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 79–92. ACM, January 1999.

[CGG99] L. Cardelli, G. Ghelli, and A. D. Gordon. Mobility Types for Mobile Ambients. In *Proceedings of ICALP'99*, LNCS 1644, pages 230–239, 1999.

[CHC94] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *[GM94]*, pages 497–517, 1994.

[CHP99] K. Crary, R. Harper, and S. Puri. What is a recursive module ? In *PLDI'99*, pages 50–63, 1999.

[CM94] L. Cardelli and J. Mitchell. Operations on records. In *[GM94]*, pages 295–350, 1994.

[CP89] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *OOPSLA'89*, volume 24 (No. 10) of *ACM SIGPLAN Notices*, pages 433–443, 1989.

[CPWK04] S. Carlier, J. Polakow, J.B. Wells, and A.J. Kfoury. System E : expansion variables for flexible typing with linear and non-linear types and intersection types. In *Programming Languages and Systems, 13th Europ. Symp. Programming, ESOP'04*, LNCS, 2004.

[CW04] S. Carlier and J.B. Wells. Type inference with expansion variables and intersection types in system E and an exact correspondence with β-reduction. Technical Report HW-MACS-TR-0012, Heriot Watt University, 2004.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Conf. Rec. 9th Ann. ACM Symp. Princ. of Prog. Lang.*, pages 207–212, 1982.

[DP00] R. Davies and F. Pfenning. Intersection types and computational effects. In *Proc. of ICFP'00*, pages 198–208. ACM Press, 2000.

[Dug94] D. Duggan. Object interfaces, polymorphic methods, and multi-method dispatch for ML-like languages. In *ACM SIGPLAN Workshop on Standard* ML *and its Applications*, 1994.

[EST95] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *OOPSLA'95*, volume 30 (No. 10) of *ACM SIGPLAN Notices*, pages 169–184, 1995.

[FHM93] K. Fisher, F. Honsell, and J. Mitchell. A lambda calculus of objects and method specialization. In *LICS'93*, pages 26–38, 1993.

[Fis96] K. Fisher. *Type systems for object-oriented programming languages*. PhD thesis, Stanford University, 1996.

[FKF98] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *POPL'98*, pages 171–183, 1998.

[FM95] K. Fisher and J. Mitchell. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems*, 1(3) :189–220, 1995.

[Gir86] J.-Y. Girard. The system $\mathcal{F}$ of variable types, fifteen years later. *Theoretical Computer Science*, 45 :159–192, 1986.

[GM94] C. Gunter and J. Mitchell, editors. *Theoretical aspects of object-oriented programming*. The MIT Press, 1994.

[HL02] T. Hirschowitz and X. Leroy. Mixin modules in a call-by-value setting. In *ESOP'02*, LNCS 2305, pages 6–20, 2002.

[HS80] J.R. Hindley and J.P. Seldin, editors. *To H.B. Curry : Essays on combinatory logic, lambda-calculus and formalism*. Academic Press, 1980.

[Jim96] T. Jim. What are principal typings and what are they good for ? In *POPL'96 : 23rd ACM Symp. Princ. of Prog. Langs*, pages 42–53, 1996.

[Jim00] T. Jim. A polar type system, 2000. ITRS'00.

[Kam88] S. Kamin. Inheritance in SMALLTALK-80 : a denotational definition. In *POPL'88*, pages 80–87, 1988.

[Kam00] F. Kamareddine. Postponement, conservation and preservation of strong normalization for generalized reduction. *Journal of Logic and Computation*, 10(5) :721–738, 2000.

[Kfo99] A.J. Kfoury. Beta-reduction as unification. *Logic, Algebra, and Computer Science*, 46 :137–158, 1999.

[Klo80] J.W. Klop. *Combinatory reduction systems*. PhD thesis, Utrecht University, 1980. Mathematical Centre Tracts Vol. 127, Mathematisch Centrum, Amsterdam.

[KMTW99] A.J. Kfoury, H.G. Mairson, F.A. Turbak, and J.B. Wells. Relating typability and expressiveness in finite-rank intersection type systems. In *Proc. 1999 International Conference on Functional Programming*, pages 90–101. ACM Press, 1999.

[KN95] F. Kamareddine and R.P. Nederpelt. Generalising reduction in the λ-calculus. *Journal of Functional Programming*, 5(4) :637–651, 1995.

[Kri90] J.-L. Krivine. *Lambda-calcul : types et modèles*. Masson, 1990.

[KW99] A.J. Kfoury and J.B. Wells. Principality and decidable type inference for finite-rank intersection types. In *Conf. Rec. POPL'99*, pages 161–174, 1999.

[KW04] A.J. Kfoury and J.B. Wells. Principality and type inference for intersection types using expansion variables. *Theoretical Computer Science*, 311(1–3) :1–70, 2004.

[Lan64] P.J. Landin. The mechanical evaluation of expressions. *Computer Journal*, Vol. 6 :308–320, 1964.

[LDG+00] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The objective caml system, release 3.07, 2000. Documentation et manuel d'utilisation, disponibles sur la page `http://caml.inria.fr`.

[Lei86] D. Leivant. Typing and computational properties of lambda-expressions. *Theoretical Computer Science*, 44 :51–68, 1986.

[LS00] F. Levi and D. Sangiorgi. Controlling Interference in Ambients. In *Proceedings of POPL'00*. ACM Press, 2000.

[Mey86] B. Meyer. Genericity versus inheritance. In *OOPSLA'86*, volume 21 (No. 11) of *ACM SIGPLAN Notices*, pages 391–405, 1986.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17 :348–375, 1978.

[Mil91] R. Milner. The Polyadic π-Calculus : a Tutorial. Technical Report ECS-LFCS-91-180, University of Edinburgh, 1991.

[MP89] O.L. Madsen and B. Møller Pedersen. Virtual classes : a powerful mechanism in object-oriented programming. In *OOPSLA'89*, volume 24 (No. 10) of *ACM SIGPLAN Notices*, pages 397–406, 1989.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The definition of Standard* ML *(Revised)*. MIT Press, 1997.

[NM03] P. Møller Neergaard and H.G. Mairson. Rank bounded intersection : types, potency, and idempotency, 2003.

[Pit02] A. Pitts. Operationel semantics and program equivalence. In *Lectures given at the International Summer School on Applied Semantics, APPSEM'00*, LNCS 2395, pages 378–412, 2002.

[Pot80] G. Pottinger. A type assignment for the strongly normalizable λ-terms. In *Hindley and Seldin [HS80]*, pages 561–577, 1980.

[PT94] B.C. Pierce and D. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2) :207–247, 1994.

[Rém94] D. Rémy. Programming with ML-ART : an extension to ML with abstract and record types. In *TACS'94*, LNCS 789, pages 321–346, 1994.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal ACM*, 12(1) :23–41, 1965.

[Roc88] S. Ronchi Della Rocca. Principal type schemes and unification for intersection type discipline. *Theoretical Computer Science*, 59(1–2) :181–209, 1988.

[Rus01] C. Russo. Recursive structures for Standard ML. In *ICFP'01*, pages 50–61, 2001.

[RV84] S. Ronchi Della Rocca and B. Venneri. Principal type schemes for an extended type theory. *Theoretical Computer Science*, 28(1–2) :151–169, 1984.

[RV98] D. Rémy and J. Vouillon. Objective ML : an effective object-oriented extension of ML. *Theory and Practice of Object Systems*, 4(1) :27–50, 1998.

[Sal78] P. Sallé. Une extension de la théorie des types en lambda-calcul. In *ICALP Udine*, volume 62 of *LNCS*, pages 398–410, 1978.

[SM96] E. Sayag and M. Mauny. A new presentation of the intersection type discipline through principal typings of normal forms. Technical report, INRIA Research Report RR-2998, 1996.

[SM97] E. Sayag and M. Mauny. Structural properties of intersection types. In *Proceedings of the 8th International Conference on Logic and Computer Science - Theoretical Foundations of Computing (LIRA)*, pages 167–175, 1997.

[Sny86] A. Snyder. CommonObjects : an overview. *ACM SIGPLAN Notices*, 21(10) :19–28, 1986.

[SV01] D. Sangiorgi and A. Valente. A distributed abstract machine for safe ambients. In *ICALP'01*, LNCS 2076, 2001.

[Tai96] A. Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3) :438–479, 1996.

[Tof90] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1) :1–34, 1990.

[TZH02] D. Teller, P. Zimmer, and D. Hirschkoff. Using ambients to control resources. In *Proceedings of CONCUR 2002*, volume 2421 of *LNCS*, pages 288–303, 2002.

[TZH04] D. Teller, P. Zimmer, and D. Hirschkoff. Using ambients to control resources. *International Journal of Information Security*, 2004.

[vB93] S. van Bakel. Principal type schemes for the strict type assignment system. *Journal of Logic and Computation*, 3(6) :643–670, 1993.

[Wan87] M. Wand. Complete type inference for simple objects. In *LICS'87*, pages 37–44, 1987.

[Wan94] M. Wand. Type inference for objects with instance variables and inheritance. In *[GM94]*, pages 97–120, 1994.

[Wel99] J.B. Wells. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3) :111–156, 1999.

[Wel02] J.B. Wells. The essence of principal typings. In *ICALP'02*, volume 2380 of *LNCS*, pages 913–925. Springer-Verlag, 2002.

[WF94] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1) :38–94, 1994.

[Wri95] A.K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4) :343–355, 1995.

[Zim00] P. Zimmer. Subtyping and typing algorithms for mobile ambients. In *Proceedings of FOSSACS'00*, volume 1784 of *LNCS*, pages 375–390, 2000.

[Zim03a] P. Zimmer. On the expressiveness of pure mobile ambients. In Luca Aceto and Björn Victor, editors, *Electronic Notes in Theoretical Computer Science*, volume 39. Elsevier, 2003.

[Zim03b] P. Zimmer. On the expressiveness of pure safe ambients. *Mathematical Structures in Computer Science*, 13, 2003.

Index