THESE DE DOCTORAT

prsente

A L'UNIVERSITE PARIS 7

Spcialit : Informatique

par

Didier REMY

Sujet de la thse :

Algbres Touffues.

Application au Typage Polymorphe des Objets Enregistrements dans les Langages Fonctionnels.

Soutenue le 19 Fvrier 1990 devant la Commission d'examen compose de

MM.	**Guy COUSINEAU**	**Prsident**
	Franois BANCILHON	**Examinateurs**
	Bruno COURCELLE	
	Grard HUET	
	Claude KIRCHNER	
	Jean-Jacques LEVY	

THESE DE DOCTORAT

prsente

A L'UNIVERSITE PARIS 7

Spcialit : Informatique

par

Didier REMY

Sujet de la thse :

Algbres Touffues.

Application au Typage Polymorphe des Objets Enregistrements dans les Langages Fonctionnels.

Soutenue le 19 Fvrier 1990 devant la Commission d'examen compose de

MM.	**Guy COUSINEAU**	**Prsident**
	Franois BANCILHON	**Examinateurs**
	Bruno COURCELLE	
	Grard HUET	
	Claude KIRCHNER	
	Jean-Jacques LEVY	

Algbres touffues. Application au typage polymorphe des objets enregistrements dans les langages fonctionnels.

(Thse de Doctorat)

Didier Rmy
Universit de Paris 7

26 fvrier 1990

Remerciements

Je tiens à remercier Guy Cousineau qui me fait l'honneur de présider ce Jury. Je remercie également François Bancilhon et Bruno Courcelle pour avoir bien voulu s'intéresser à ce travail. Je remercie Claude Kirchner et Jean-Jacques Lévy pour toute l'attention qu'ils ont portée à ce travail et pour leurs conseils et leurs critiques très précieux.

Les discussions fructueuses avec Thomas Ehrhard et Ascánder Suárez ont motivé mes premiers travaux et j'ai beaucoup apprécié leurs chaleureux encouragements.

Je tiens également à remercier toutes les personnes du projet Formel qui en me demandant d'améliorer le typage du langage $CAML$ ont donné une motivation très concrète à mes travaux. Je leur dois aussi l'environnement agréable de programmation $CAML$, qu'ils ont parfois étendu à ma demande, et sans lequel il m'aurait été difficile de réaliser mes différentes maquettes. Je tiens également à leur dire combien les nombreuses discussions m'ont aidé à progresser.

En partageant mon bureau, María Virginia Aponte a également partagé mes humeurs et je lui suis très reconnaissant pour la patience avec laquelle elle a su écouter et critiquer la plupart de mes ébauches.

Je remercie infiniment Gérard Huet pour avoir dirigé mes travaux. Ses conseils précieux sont une aide considérable et m'ont permis de construire progressivement mon propre chemin.

Résumé

Le langage ML est un langage fonctionnel typé polymorphe avec synthèse de types. Nous proposons une méthode plus qu'une solution, à la fois simple et efficace, permettant la synthèse des types dans une extension de ML avec des objets enregistrements avec héritage. Simplicité, car les règles de typage de ML ne sont pas modifiées, l'héritage étant obtenu simplement par le polymorphisme dans les types enregistrements. Efficacité, car nous montrons comment extraire naturellement à partir des règles de typage un algorithme avec partage et permettant un calcul incrémental de la généralisation.

Les objets enregistrements sont des fonctions partielles de domaines finis d'un ensemble dénombrable d'étiquettes vers l'ensemble des valeurs. Nous leurs associons des types enregistrements qui sont des fonctions partielles de l'ensemble des étiquettes vers l'ensemble des types, que nous rendons totales en définissant des types par défaut pour les étiquettes absentes.

Les types enregistrements sont représentables de manière finie dans une algèbre de termes munie d'une famille d'équations très sévèrement controlées par des sortes, que nous appelons la théorie touffue à brins étiquetés. Dans une touffe, l'extraction des brins est obtenue par substitution pour les variables et par propagation vers la racine par des équations de distributivité, et la permutation des brins est assurée par des équations de commutativité gauche.

Nous montrons que la théorie touffue à brins étiquetés est une théorie syntaxique, et nous en déduisons facilement que l'unification y est décidable et unitaire. Puis nous montrons que l'algèbre des types de ML peut être étendue avec une théorie décidable, unitaire et régulière.

Le langage obtenu est paramétré par le jeu de primitives avec leur types qui réalisent les opérations élémentaires sur les enregistrements. Cette méthode devrait s'étendre à des types récursifs et s'appliquer à d'autres langages que ML, notamment à des langages avec de l'inclusion de types.

Mots Clefs

Algèbres Touffues, Classes, Héritage, Inclusion, Langages Fonctionnels, ML, Objets Enregistrements, Polymorphisme, Programmation Orientée par les Objets, Synthèse de Types, Théories Equationnelles, Théories Syntaxiques, Typage, Unification.

Table des matières

Introduction

Evolution des langages de programmation

De la machine à calculer de Pascal à aujourd'hui, l'informatique est une longue et rapide aventure marquée par l'alternance entre les découvertes théoriques et l'évolution de ses techniques, mais peut être plus encore par son miracle technologique permanent.

La machine universelle de Turing, puis les théories des automates et des langages ont permis de s'abstraire des engrenages, des ampoules ou des transistors composant une machine réelle ainsi que des rubans, cartes perforées ou claviers indispensables pour la commander.

Cela n'empêche pas l'histoire de l'informatique d'être bien décrite par l'évolution continue des langages de programmation qui n'a cessé d'accroître la rapidité et la sécurité de la mise au point des programmes en essayant de conserver l'efficacité des langages machines.

Efficacité d'exécution

Le premier et le plus simple des langages de programmation est le langage machine. Son alphabet est un ensemble d'instructions, c'est-à-dire de tuples de 0 et de 1 directement exécutables par la machine. Un langage machine est difficilement compréhensible par un humain, mais on peut remplacer les instructions par une représentation symbolique: c'est le langage d'assemblage. Plus accessible, il reste cependant entièrement lié aux instructions de la machine considérée.

Le premier langage qui fut entièrement indépendant de la machine est Fortran. Il a été conçu en 1957 par John Backus, et est encore aujourd'hui un des langages les plus répandus. Fortran est un langage *impératif* : sa structure est assez peu différente de celle d'un automate et un programme peut encore être vu comme une suite de commandes, modifiant l'état d'une machine virtuelle.

L'avantage d'un tel langage est d'être très facilement et efficacement *compilé*, c'est-à-dire traduit mécaniquement en langage machine pour être exécuté. D'autres langages impératifs sont ensuite apparus parmi lesquels on peut citer Cobol, C, Pascal qui sont les plus répandus.

Une caractéristique commune à tous les langages impératifs est de distinguer les données des programmes, ce qui les rend peu adaptés à des manipulations symboliques où l'on veut parfois considérer un programme comme une donnée.

Rapidité d'écriture

Le lambda-calcul [1, 26], introduit vers 1930 pour l'étude des fondements de la logique et des mathématiques, s'est révélé plus tard d'un grand intérêt en informatique et a joué un rôle important dans la conception des langages Algol, LISP, puis plus tard ISWIM.

Dans le langage LISP conçut vers 1958 par McCarthy, les programmes et les données sont représentés par une même structure. C'était un premier grand pas vers la programmation *fonctionnelle* qui ne fut réellement atteinte qu'avec les langages Scheme ou ML. Dans ces langages, un programme est vu comme une fonction qui associe un résultat à chaque valeur qui lui est donnée en argument. Les fonctions sont des objets de *première classe* car elles peuvent être communiquées à d'autres fonctions qui se chargeront éventuellement de les appliquer à des arguments.

Sécurité dans l'écriture

La sûreté dans la mise au point des programmes a toujours été un problème majeur. Comment un humain pouvait-il relire un programme écrit en langage machine et constitué d'une longue suite de 0

et de 1 ? Le remplacement des langages machines par des langages plus évolués est donc un premier facteur de sûreté. Mais déjà Fortran introduisait une notion de *type*. En déclarant des variables d'un certain *type*, le programmeur introduisait des contraintes qui devaient être partout vérifiées dans le programme. Le compilateur effectue mécaniquement cette vérification. Le typage de Fortran est *faible* car un typage correct du programme n'exclut pas un comportement aberrant à l'exécution. Le typage est également faible dans le langage C où le programmeur peut tricher en demandant qu'il ne soit pas tenu compte de certaines contraintes. Par contre le typage de Algol 68 est *fort* : un programme bien typé ne peut pas avoir un comportement aberrant à l'exécution.

Preuves

Des langages de preuves ont été développés dans la perspective de vérifier voire de synthétiser des preuves mathématiques. Ces systèmes s'appuient sur des logiques intuitionnistes, dans lesquels il n'y a pas la règle du tiers exclu.

L'isomorphisme de Curry Howard relie ces systèmes aux lambda-calculs typés en considérant les types comme des propositions et les termes du lambda-calcul comme des preuves. Le langage de programmation ML a été conçu dans le projet LCF d'Edinburgh, comme un méta-langage pour un démonstrateur assisté. C'est devenu un langage de programmation à part entière, fonctionnel fortement typé et polymorphe. Le polymorphisme permet à une valeur d'avoir plusieurs types.

D'autres systèmes de représentation des connaisances mathématiques tels Automath, la théorie des types de Martin-Löf ou le calcul des constructions ont permis la réalisation de vérificateurs de preuves. Le calcul des constructions peut également être utilisé pour spécifier des algorithmes, et la preuve de réalisabilité d'un algorithme permet d'en extraire automatiquement un programme. Produits mécaniquement à partir de leurs spécifications, ces programmes sont certainement corrects.

Depuis ML, d'autres langages ont été inspirés de systèmes logiques, notamment les langages FUN ou QUEST qui correspondent à des lambda-calcul de second ordre [25] et d'ordre supérieur étendus avec un mécanisme de sous-typage. Le polymorphisme de ces langages est plus puissant que celui de ML. Cela n'a pas empêché l'apparition de nombreux langages partageant la spécifité de ML de synthétiser automatiquement leurs types : Amber, CAML, Haskell, LML, Miranda, SML, etc.

La programmation avec héritage

Un intérêt croissant

Les langages fonctionnels fortement typés sont très prometteurs, car la facilité de la programmation fonctionnelle et la sécurité d'un langage fortement typé rendent la mise au point des programmes extrêmement rapide.

Mais un autre style de programmation est apparu vers les années 80 : c'est la programmation orientée par les objets. Un objet est une donnée structurée à laquelle sont attachés des champs dont certains dits *privés* sont cachés au reste du monde. A la différence de la programmation autoritaire des langages impératifs, la programmation orientée par les objets s'effectue par envoi de messages déléguant aux objets le choix de l'action à effectuer. Il se trouve que la plupart des langages orientés objets ont en plus un mécanisme d'héritage : les objets appartiennent à des *classes* partiellement ordonnées. On peut définir pour chaque classe un ensemble de procédures ou *méthodes* qui seront automatiquement propagées dans toutes les sous-classes.

Le précurseur de ces langages est Simula, et le plus connu est certainement Smalltalk, mais il existe de nombreux dialectes de langages orientés objets ayant chacun leur spécificité. La plupart des langages impératifs et fonctionnels possèdent une extension objet. Une exception notable est les langages fortement typés, car la programmation par envoi de messages a un comportement décentralisé dont il est difficile de garantir la sécurité.

Luca Cardelli est le premier à avoir dénoncé cette grave lacune des langages fortement typés et à leur avoir ajouté des possibilités d'héritage. Dans le langage Amber [8], il a expérimenté la solution minimale d'un langage fonctionnel explicitement typé, non polymorphe, avec inclusion de types [7, 9]. Les méthodes sont des fonctions et l'héritage est obtenu par le sous-typage. Ce langage possède notamment des objets enregistrements non polymorphes avec inclusion. Cette expérience fut probante, et le langage FUN, conçu en 1985 par Luca Cardelli et Peter Wegner [12] reprend les idées essentielles d'Amber, mais avec un typage explicite au second ordre.

Quelques tentatives en ML

L'intérêt d'une extension conservative de ML avec des objets est grand car les langages de la famille ML sont les seuls à permettre une synthèse automatique des types, ce qui semble encore être un avantage considérable.

En 1987, Mitchell Wand tenta d'introduire le typage des objets enregistrements en codant l'héritage par du polymorphisme [59]. Hélas cette solution est incomplète, et la rendre complète conduit à une explosion combinatoire de l'algorithme de synthèse de types [60]. Elle reste néanmoins intéressante, notamment parce qu'elle introduit une nouvelle opération sur les objets enregistrements permettant de rajouter un champ fixé à un enregistrement arbitraire. Elle peut également être étendue à une opération de collage de deux objets enregistrements arbitraires, mais cette extension retire aux programmes ML la propriété de possèder un type principal, la remplaçant par l'obtention d'ensembles complets de types principaux.

L'année suivante, deux solutions très voisines ont été proposées indépendamment pour éviter l'explosion de la solution complète. Elles consistent toutes deux à restreindre l'utilisation de la règle d'extension à des enregistrements dont on connaît tous les champs. L'une proposée par Peter Buneman et Atsushi Ohori dans [48] est plongée dans un langage avec des opérations propres aux bases de données, et a été implantée dans le langage Machiavelli [50]. L'autre proposée par Lalita A. Jategaonkar et John C. Mitchell dans [30] compense partiellement cette restriction par un mécanisme de sous-typage permettant de récupérer une partie de la puissance perdue par des déclarations de relations d'inclusion.

Le sous-typage dont il s'agit est dit *structurel* car il respecte la structure des types et les coercions de types n'ont ultimement lieu qu'entre types atomiques. Il a été introduit par John C. Mitchell dans [43] puis repris dans [44]. D'autres travaux sur l'ajout du sous-typage dans ML ont également été menés par Y-C. Fuh et Pratreek Mishra dans [23, 24] qui ont repris le sous-typage structurel proposé par John C. Mitchell mais avec une approche différente, mettant à jour des problèmes d'efficacité de l'algorithme de synthèse de types.

Le sous-typage dans les langages d'ordre supérieur

Parallèlement à ces tentatives d'introduction d'opérations sur les enregistrements avec héritage dans ML, Luca Cardelli prolongea leur étude dans des langages d'ordre supérieur [10]. Avec John C. Mitchell, il propose dans [11] une extension des opérations sur les records avec des informations négatives pour corriger certaines faiblesses des systèmes précédents. Pavel Curtis étend dans [17] les contraintes de sous-typages de FUN. Ce système beaucoup plus général contient en particulier tous les systèmes de sous-typages proposés pour ML. Une version restreinte permet d'étendre simplement le langage FUN de façon à bien coder les types récursifs [6].

Ce formalisme a été utilisé par John C. Mitchell dans [45] pour décrire un système de sous-typage spécialisé a des opérations sur les objets. Il y est montré en particulier que le codage de ces opérations par des objets enregistrements n'est pas assez puissant, et qu'il est nécessaire d'introduire des primitives spécifiques aux opérations sur les objets. Les règles de typage sont différentes, mais les mécanismes restent les mêmes.

De nombreux travaux ont également été effectués pour donner des sémantiques au sous-typage et à la programmation orientée par les objets.

De l'avenir de ML

L'extension systématique des langages impératifs traditionnels par des dialectes objets et la multitude des travaux sur des extensions objets avec héritage des langages fonctionnels fortement typés montrent l'intérêt considérable de ces langages pour l'avenir. Mais si ces extensions ont finalement réussi dans les langages d'ordre supérieur explicitement typés, les tentatives dans le langage ML ont échoué partiellement. Il y va de l'avenir des langages de la famille ML de savoir incorporer ces notions tout en conservant leur caractéristique essentielle qui est de permettre la synthèse de types.

Les travaux présentés dans ce document ont été motivés par la volonté de relever ce défi lancé à ML. Nous défendons la thèse que le langage ML peut être étendu très naturellement avec des objets enregistrements permettant de coder les opérations élémentaires d'héritage sur les objets. Plutôt que d'épuiser les dernières ressources de ML, cette thèse lui découvre de nouvelles possibilités.

Un chemin sinueux entre la pratique et la théorie

L'auteur souhaite dévoiler le chemin parfois sinueux qui l'a mené jusqu'à l'écriture de ce document, puisse cela être utile à d'autres voyageurs ! L'ordre chronologique des principales étapes est scrupuleusement respecté, et l'interprétation personnelle réduite autant que possible.

Ce voyage a été entrepris dans l'effervescence décrite ci-dessus, à laquelle il convient d'ajouter un intérêt tout particulier au sein du projet Formel pour une extension du langage CAML par des sommes et des produits étiquetés.

Les premières idées

La découverte de l'incomplétude et la non terminaison de la méthode de Mitchell Wand[1] en collaboration avec Laurence Puel et Guy Cousineau a précisé le premier objectif.

Il est difficile de décrire fidèlement la résolution de l'incomplétude, mais le vecteur principal est sans doute la volonté de capturer la disjonction des cas par des variables supplémentaires. Les premières tentatives échouèrent car l'apparition de variables au cours du mécanisme de simplification, rendait alors la terminaison de l'algorithme difficile à entrevoir. La manipulation d'exemples a montré que les variables apparaissant étaient toujours des copies d'un même modèle sur des étiquettes différentes. L'intuition de la solution a émergé très rapidement dès lors que l'on a accepté de raisonner sur des termes infinis possédant une description des champs pour chaque étiquette.

Une première implantation de l'algorithme dans le langage CAML permit de confirmer expérimentalement cette intuition. Les structures infinies étaient simulées par un codage semblable à celui des structures paresseuses de CAML. Il faut reconnaître que cette implantation comportait quelques anomalies corrigées par la suite.

La formalisation dans le langage CAML posait le problème de codage des structures infinies, mais régulières, dans une structure finie. Différents codages mathématiques très simples ont été envisagés mais il était difficile de les relier à des algorithmes d'unification connus. Aussi la première description formelle s'appuie-t-elle sur l'intuition de structures infinies tout en exigeant que le nombre d'étiquettes soit fini. Le problème est éliminé mais pas résolu. Le cas d'un ensemble infini d'étiquettes n'a jamais posé de problème en pratique, et sa première implantation était déjà très proche de la version finale, mais sa formalisation a été gelée faute d'une bonne représentation algébrique des structures infinies.

Après une révision de l'implantation et l'ajout des types récursifs une comparaison détaillée avec les autres systèmes existants a montré les limitations du codage de l'inclusion par du polymorphisme, posant la question de l'utilisation du même codage dans des systèmes de typage plus puissants.

Des recherches auxiliaires fructueuses

L'investigation des systèmes avec types conjonctifs [14] et des systèmes avec sous-types, débouchant sur deux solutions différentes, a beaucoup aidé à la compréhension des mécanismes d'inclusion et, s'il en reste très peu de traces apparentes dans ce document, elle l'a certainement influencé de façon indirecte. La première solution est la découverte des systèmes de typage[2] ML$^{\mathcal{R}}$ paramétrés par une relation de β-réduction $\mathcal{R}$ confluente et nœthérienne. Le typage d'un terme consistant à effectuer la réduction $\mathcal{R}$ de ce terme et à typer la forme canonique obtenue dans le lambda-calcul pur. Le langage ML est obtenu en prenant pour $\mathcal{R}$ la réduction de tous les radicaux marqués par des *let* dans le programme source. Cette dernière solution a été abandonnée car on ne connaît d'algorithme efficace pour décider du typage dans ces systèmes que pour des relations $\mathcal{R}$ sans intérêt. La seconde solution est l'adaptation du codage dans une extension de ML avec sous-typage structurel. Mais l'inefficacité des algorithmes de sous-typage structurel [23] était inquiétante. Une petite étude de l'inclusion structurelle aboutissait à de nombreuses simplifications des algorithmes existants. Une implantation s'imposait.

Cela conduisit à la recherche d'un algorithme efficace pour le langage ML lui-même. La solution décrite dans ce document est apparue en essayant d'effectuer le calcul de l'unification directement sur des types génériques plutôt que de prendre sans cesse des instances de types génériques que l'on généralise après les calculs. Eviter la copie restait très difficile et demandait au minimum une

[1] Cette étude de l'article de Mitchell Wand est tout à fait indépendante de celle des personnes citées dans [60].
[2] Ces systèmes ont été redécouverts indépendamment par Paris Kanellakis.

gestion complexe de copies locales ; en revanche cette étude a mis en évidence la possibilité de calculer les variables généralisables de façon incrémentale. La formalisation de ce calcul par un problème d'unification classique fut immédiate.

Cette maquette a montré que les calculs de la généralisation pouvaient être très localisés, et d'un coût indépendant du contexte dans lequel ils étaient effectués. C'est justement cette sensibilité au contexte qui rendait particulièrement inefficaces les algorithmes de synthèse en présence de relations de sous-typage. L'exploitation de l'idée précédente a conduit à effectuer localement les simplifications des ensembles de contraintes plutôt que de les retarder. En maintenant une structure de graphe avec pointeurs arrière, les simplifications de complexité linéaire par rapport à la taille du graphe se sont avérées suffisantes pour obtenir des formes canoniques presque minimales. Ce que d'ailleurs l'étude théorique permettait de deviner. Cette maquette s'est révélée être très efficace, puisque d'efficacité comparable en moyenne au cas sans inclusion.

L'ajout des enregistrements à cette maquette révéla un défaut dans le codage des structures infinies. Les modèles, sortes de glaçons à l'intérieur desquels on ne peut entrer qu'en les dégelant, rendaient plus difficile la gestion des pointeurs arrière.

La formulation finale

Ce fut le moment de formaliser définitivement les structures infinies. Il fallait expliquer le clonage du modèle par des opérations algébriques, en essayant de rendre les modèles aussi transparents que possible. Le clonage produit deux copies, l'une étant le type du champ extrait, l'autre devenant le nouveau modèle. L'explication du clonage devint évidente dès lors qu'est apparue la nécessité de le découper en opérations atomiques. En effet, le clonage d'une variable correspond à une substitution. Le clonage d'un terme s'obtient par clonage de ses variables et par séparation des termes clonés par des équations permettant de distribuer la "colle" sur les symboles de types.

La formulation complète se ramène à l'étude de la théorie équationnelle formée des équations de clonage. Le formalisme de standardisation de l'unification dans les théories équationnelles de Claude Kirchner présenté dans [36] est apparu bien adapté. En tant qu'extension de l'algorithme de Martelli et Montanari [42], il conduit à des algorithmes d'unification efficaces. Il fallut néanmoins développer une méthode nouvelle pour montrer que la théorie de clonage était syntaxique.

Ce chemin se termine par le réajustement de l'implantation au formalisme ci-dessus, ce qui a permis de remplacer la gestion particulière des modèles par deux transformations de mutation très simples.

Si l'auteur peut se permettre de juger de ce parcours, l'alternance permanente entre la formulation et l'implantation lui semble avoir été extrêmement fructueuse. Notamment le souci d'avoir une implantation efficace, qui imposent à la théorie d'être simple plutôt que de la compliquer. Il tient également à souligner l'importance des travaux auxiliaires pratiques ou théoriques qui même s'ils n'ont pas directement débouché lui ont semblé améliorer considérablement sa perception des problèmes. En revanche, il semble qu'avoir retardé la formulation du cas d'un ensemble infini d'étiquettes ait été une erreur. Mais il n'est pas possible de rejouer la partie, même fictivement pour vérifier cette hypothèse.

Plan de cet ouvrage

Les résultats principaux

Le résultat principal est la décidabilité et le cas échéant l'existence d'un type principal pour l'extension ΠML du langage ML avec des objets enregistrements, énoncé par le théorème 14 dans le chapitre 4.

Ce résultat découle immédiatement de deux résultats secondaires indépendants et plus généraux. Le premier est la décidabilité du typage dans une extension $\mathcal{K}A$ML de ML où les types sont avec sortes et munis d'une théorie équationnelle régulière et décidable, énoncée par le théorème 7 dans le chapitre 3. Le second est la décidabilité de l'unification dans la théorie touffue à brins étiquetés énoncée par le théorème 9 dans le chapitre 4.

Trois thèmes

Le premier chapitre rappelle les résultats généraux sur les algèbres de termes et présente les principales notations.

Le second chapitre est consacré à l'étude de l'unification dans les théories équationnelles et repose essentiellement sur les travaux de Claude Kirchner présentés dans [36]. Les deux premières parties reprennent l'essentiel du formalisme, mais parfois de façon plus abstraite afin de pouvoir appliquer les résultats à deux cas non standards. Dans la troisième partie, nous étudions un ensemble d'unificandes non standard : nous associons un degré à chaque terme, et nous exigeons que les substitutions diminuent les degrés. Cette étude trouvera son application dans le chapitre 3. La dernière partie est consacrée à l'étude des théories syntaxiques. Nous y reprenons les définitions de Claude Kirchner, et nous développons un outillage d'étude de la résolvance d'un ensemble d'axiomes plus général que celui de [36]. Cet outillage sera utilisé dans le chapitre 4. Nous pensons qu'il peut être utile dans d'autres théories équationnelles, et qu'il peut conduire à une méthode de complétion équationnelle.

Le troisième chapitre décrit la synthèse de types dans le langage ML. Le typage de ML est étendu avec des types munis de sortes et dans une théorie équationnelle régulière et décidable. Le résultat obtenu est la décidabilité du typage, et l'obtention d'un type principal dans le cas d'une théorie pour laquelle l'unification est décidable et unitaire. Cela étend donc le résultat de Damas-Milner [18]. La preuve est en deux étapes. La première étape consiste à transformer progressivement le système d'inférence initial de façon à obtenir un système équivalent dirigé par la syntaxe, et dans lequel la règle de généralisation a été dispersée dans chacune des autres règles. Le système final est ensuite considéré comme l'opération de validation de relations de typage considérées comme des unificandes. Nous définissons un ensemble de transformations des relations de typage qui permet de réduire un problème de typage quelconque en un système de multi-équations. Cette méthode a l'avantage de séparer le contrôle du mécanisme de réduction. En particulier certaines opérations pourraient être menées en parallèles.

Dans le dernier chapitre, nous utilisons les résultats précédents pour étendre ML avec des objets enregistrements permettant des opérations d'héritage. Nous supposons d'abord que l'ensemble des étiquettes est fini. Après une approche intuitive du problème d'héritage et de sa solution, nous donnons une formalisation simple dans une extension de ML avec des sortes mais sans équation. Les idées de cette partie ont été présentées dans [52]. Une analyse intuitive de l'extension à un ensemble infini d'étiquettes permet de motiver l'introduction des algèbres touffues. La plus simple des algèbres touffues permet de représenter de façon finie des tuples infinis ultimement réguliers, mais elle ne sera pas suffisante pour décrire le typage des objets enregistrements. En particulier la nécessité de pouvoir extraire les brins par nom plutôt que par position, conduira à l'algèbre touffue à brins étiquetés. Les termes touffus ont une représentation redondante. Nous définissons des termes canoniques et nous introduisons une opération d'expansion qui permet toujours de se ramener à une forme canonique. Nous montrons que deux termes égaux à un renommage et une expansion près auront le même comportement dans un problème d'unification. Nous donnons finalement une formalisation complète et générale du typage des objets enregistrements.

On trouvera en annexe une extension des algèbres touffues à des algèbres ω-touffues, où une touffe pourra elle-même être touffue, et la description complète de deux maquettes de synthétiseurs de types développées dans le langage CAML [16, 62]. La première détaille sur le noyau de ML le fonctionnement des différents algorithmes. La seconde est une extension pour un langage avec filtrage, déclarations de types et objets enregistrements.

Pour une lecture non linéaire

Les trois chapitres principaux sont présentés dans leur ordre de dépendance. Les résultats du chapitre 2 sont utilisés dans les chapitres 3 et 4 et ceux du chapitre 3 permettent de conclure dans le chapitre 4. Mais chacun de ces chapitres peut être lu séparément.

L'ordre logique est souvent contradictoire avec une approche pédagogique et intuitive. Pour chaque chapitre, un paragraphe en décrit très brièvement son contenu et sa position dans la composition finale. Lorsque cela a semblé utile, nous avons également ajouté une courte présentation informelle, afin de motiver le lecteur et de l'aider à se forger sa propre intuition. Bien souvent, ces explications font référence à des applications ultérieures et il est tout à fait possible de s'y reporter pour préciser ses intuitions.

Le chapitre principal est bien sûr le chapitre 4. Les parties 4.1 et 4.2 donnent une approche progressive de la solution tout en montrant les difficultés, et justifient les études précédentes. La solution finale dans la partie 4.6 est accompagnée d'un petit échantillon d'exemples permettant une double lecture. C'est une bonne introduction pour le lecteur non averti qui y découvrira l'intuition d'une solution d'une simplicité mystérieuse pouvant le motiver à une étude plus approfondie. Le lecteur confirmé assistera à l'assemblage des différentes composantes.

Chapitre 1

Rappels et notations

Ce court chapitre présente les formalismes connus et les notations utilisées dans cette thèse.

1.1 Notations

1.1.1 Ensembles

On note $\mathbb{N}$ l'ensemble des entiers naturels et $\mathbb{N}_+$ l'ensemble des entiers strictement positifs. L'ensemble des entiers de 1 à p est noté $[1, p]$.

Un *p-uplet* est un produit de n éléments noté indifféremment $(x_1, \ldots x_p)$ ou $(x_i)_{i\in[1,p]}$. Lorsque p est précisé par le contexte, on pourra simplement noter (x_i). La i-ème composante d'un p-uplet x, définie pour i inférieur ou égal à p est notée $x_{/i}$. Le 0-uplet est noté ϵ. Un couple est un 2-uplet.

L'ensemble des p-uplets homogènes (ou suites finies) d'éléments d'un ensemble E est noté E^*. En général, nous désignerons les p-uplets de E^* par les lettres u, v et w et les éléments de E par les lettres x, y et z. La concaténation des p-uplets u et v est simplement notée uv.

Un *multi-ensemble* est une application d'un ensemble E dans $\mathbb{N}_+$. Il est *fini* si et seulement E est fini. Lorsque qu'un multi-ensemble est fini, on peut le représenter par énumération comme un ensemble dans lequel l'ordre n'a pas d'importance, mais plusieurs éléments identiques ne sont pas confondus.

Un doublet est un ensemble de cardinal 2.

1.1.2 Fonctions

Soit φ une fonction de E vers F.

- Pour tout sous-ensemble E' de E, on note $\varphi \restriction E'$, la restriction de φ à E'.
- On note $\varphi^{(-1)}$ la fonction de F dans $\mathcal{P}(E)$ qui associe à chaque élément y de F l'ensemble
$$\{x \mid y = \varphi(x)\}.$$
- On note φ^{-1} la relation sur $F \times E$ définie par
$$\varphi^{-1}(y, x) \iff y = \varphi(x)$$
Si φ est injective, cette relation est une fonction.

1.2 Algèbres et termes de premier ordre

1.2.1 Signatures

Définition 1.1 Soit $\mathcal{K}$ un ensemble non vide dénombrable dont les éléments seront appelés *sortes* et notés ι et κ. Les p-uplets de sortes seront désignés par les lettres π et ϖ et notés $\iota_1 \otimes \ldots \iota_p$. Une *signature sur $\mathcal{K}$* est un couple noté $\pi \Rightarrow \iota$ formé d'un p-uplet de sortes π et d'une sorte ι. L'entier p est appelé *arité* de la signature. □

Définition 1.2 Une *présentation* est un triplet Σ composé d'un ensemble de sortes $\mathcal{K}$, d'un ensemble dénombrable de symboles $\mathcal{C}$ et d'une fonction totale ζ de $\mathcal{C}$ dans l'ensemble des signatures sur $\mathcal{K}$. On dit aussi que Σ est une signature de $\mathcal{C}$ sur $\mathcal{K}$. On note $\overline{\varsigma}$ et ς respectivement la première projection et la seconde projection de la fonction ζ.

Si $\pi \Rightarrow \iota$ est l'image d'un symbole f par ζ, on note aussi $f : \pi \Rightarrow \iota$, et on dit que f est de signature $\pi \Rightarrow \iota$ et de la sorte ι. Deux symboles $f : \pi \Rightarrow \iota$ et $g : \varpi \Rightarrow \iota$ sont dits *homogènes*. On note $\varrho(f)$ l'arité de π, aussi appelé arité de f. Un symbole d'arité nulle est dit *constant* et un symbole d'arité un est dit *monadique*. On désignera les symboles d'arité quelconque par les lettres f, g et h et les constantes par les lettres a, b et c.

On dit aussi que l'ensemble des symboles $\mathcal{C}$ est *signé* sur $\mathcal{K}$. Lorsque $\mathcal{K}$ est un singleton, alors la fonction signature est entièrement définie par la fonction arité. La présentation est alors *simplement graduée*. □

Définition 1.3 Soit $\mathcal{K}$ un ensemble de sortes. Un $\mathcal{K}$-*ensemble* est un ensemble A muni d'une application, appelée A-sorte et notée ς_A, de A dans $\mathcal{K}$. Si ι est une sorte, on note A^ι l'ensemble $\varsigma_A^{(-1)}(\iota)$ des éléments de A de la sorte ι, et $A^{(\iota_1,\ldots\iota_p)}$ le produit cartésien $A^{\iota_1} \times \ldots A^{\iota_p}$. On dit que le $\mathcal{K}$-ensemble A a la propriété *toujours* p si pour toute sorte ι, l'ensemble A^ι a la propriété p. Une $\mathcal{K}$-*fonction* est une fonction φ d'un $\mathcal{K}$-ensemble A vers un $\mathcal{K}$-ensemble B vérifiant $\varsigma_A = \varsigma_B \circ \varphi$. □

1.2.2 Σ-algèbres

Soit $\mathcal{K}$ un ensemble de sortes, $\mathcal{C}$ un ensemble de symboles et Σ une présentation de $\mathcal{C}$ sur $\mathcal{K}$.

Définition 1.4 On appelle Σ-*algèbre* un couple (A, Σ_A) où A est un $\mathcal{K}$-ensemble toujours non vide et Σ_A est une fonction qui à chaque élément de f de signature $\pi \Rightarrow \iota$ de $\mathcal{C}$ associe une fonction de A^π dans A^ι noté f_{Σ_A}. □

Définition 1.5 On appelle homomorphisme de l'algèbre (A, Σ_A) sur l'algèbre (B, Σ_B) toute $\mathcal{K}$-application φ de A dans B telle que pour tout symbole f de signature $\iota_1 \otimes \ldots \iota_p \Rightarrow \iota$ de $\mathcal{C}$,

$$\varphi^\iota(f_{\Sigma_A}(a_i)_{i\in[1,p]}) = f_{\Sigma_B}(\varphi^{\iota_i}(a_i))_{i\in[1,p]}$$

□

Définition 1.6 Soit $\mathcal{V}$ un $\mathcal{K}$-ensemble donc les éléments seront appelés variables. Une Σ-algèbre (A, Σ_A) est dite *libre* sur $\mathcal{V}$ si :

1. $\mathcal{V} \subset A$
2. Pour toute Σ-algèbre (B, Σ_B), toute $\mathcal{K}$-fonction de $\mathcal{V}$ dans B s'étend en un homomorphisme de (A, Σ_A) dans (B, Σ_B).

□

Théorème 1 *A un isomorphisme près, il existe une et une seule Σ-algèbre libre sur un $\mathcal{K}$-ensemble $\mathcal{V}$ fixé. Elle est notée $\mathcal{T}(\mathcal{C}, \mathcal{V})$ ou simplement $\mathcal{T}$ lorsque $\mathcal{C}$ et $\mathcal{V}$ sont précisés par le contexte.*

1.2.3 Arbres

Soit Σ une présentation de $\mathcal{C}$ sur $\mathcal{K}$ et $\mathcal{V}$ un $\mathcal{K}$-ensemble de variables.

Définition 1.7 On appelle arbre de signature Σ sur l'ensemble de variables $\mathcal{V}$ une application σ d'un sous-ensemble de $I\!N_+^*$ appelé domaine de σ et noté $dom\,(\sigma)$ dans $\mathcal{C} \cup \mathcal{V}$ vérifiant les propriétés suivantes :

1. $\forall u \in dom\,(\sigma),\ \forall x \in I\!N_+,\ ux \in dom\,(\sigma) \iff u \in dom\,(\sigma) \wedge \sigma(u) \in \mathcal{C} \wedge x \in [1, \varrho(\sigma(u))]$
2. $\forall u \in dom\,(\sigma),\ \forall x \in I\!N_+,\ ux \in dom\,(\sigma) \Longrightarrow \varsigma(\sigma(ux)) = \overline{\varsigma}(\sigma(u))_{/x}$

Un arbre est *fini* si son domaine est fini, *vide* si son domaine est vide. Le domaine d'un arbre non vide contient toujours ϵ. Les éléments du domaine d'un arbre σ sont aussi appelés occurrences de σ. L'ensemble des arbres est noté $\mathcal{A}$. La sorte d'un arbre σ, égale à $\varsigma(\sigma(\epsilon))$ est simplement notée $\varsigma(\sigma)$. Deux arbres de même sorte sont dits *homogènes*.

On note $\mathrm{Top}\,(\sigma)$ le symbole ou la variable à l'occurrence ϵ de l'arbre σ. Si c'est un symbole f, on dit aussi que σ a le symbole f en tête.

Si f est un symbole d'arité p et $(\mathcal{T}_i)_{i\in[1,p]}$ sont des ensembles de termes on note parfois $f(\mathcal{T}_i)$ l'ensemble des termes ayant le symbole f en tête et leurs i-èmes sous-termes directs dans $\mathcal{T}_i$. □

Définition 1.8 Soit σ un arbre et u un élément de $dom\,(\sigma)$. Le sous-arbre de σ à l'occurrence u est l'arbre, noté $\sigma_{/u}$, défini sur l'ensemble des occurrences

$$\{v \mid uv \in dom\,(\sigma)\}$$

par $\sigma_{/u}(v) = \sigma(uv)$. Si i est un entier dans $[1, \varrho(\sigma(\epsilon))]$, $\sigma_{/x}$ est le appelé le i-ème sous-arbre direct de σ. □

Notation Soit σ un arbre et τ un sous-arbre de σ. On note $\mathcal{O}(\tau, \sigma)$ l'ensemble $\sigma^{(-1)}(\tau)$ dit ensemble des occurrences de τ dans σ. Deux occurrences non préfixes l'une de l'autre sont dites *disjointes*. Le domaine d'un arbre σ est parfois noté $\mathcal{O}(\sigma)$. On note $\mathcal{O}_{\mathcal{C}}(\sigma)$ l'ensemble $\sigma^{(-1)}(\mathcal{C})$ des occurrences non variables de σ et $\mathcal{O}_{\mathcal{V}}(\sigma)$ celui des ocuurrences variables.

L'ensemble des variables d'un arbre σ, noté $\mathcal{V}(\sigma)$, est égal à $\sigma(\mathcal{O}(\sigma)) \cap \mathcal{V}$. Par extension, si U est un ensemble

$$\mathcal{V}(U) = \bigcup_{x\in U} \mathcal{V}(x).$$

On notera $\overline{\mathcal{V}}(W)$ l'ensemble $\mathcal{V} \setminus \mathcal{V}(W)$ des variables *en dehors* de W, et parfois même simplement $\overline{W}$.

Définition 1.9 Soit σ et τ sont deux arbres et u une occurrence de σ. On note $\sigma[\tau/u]$ l'arbre défini par

$$\begin{cases} \sigma[\tau/u](w) = \tau(v) & \text{si } u \text{ est un préfixe de } w \text{ et } w = uv, \\ \sigma[\tau/u](w) = \tau(w) & \text{si } u \text{ et } w \text{ sont disjointes.} \end{cases}$$

□

Proposition 1.1 *L'ensemble des arbres finis non vides sur $\mathcal{C} \cup \mathcal{V}$ est isomorphe à $\mathcal{T}(\mathcal{C}, \mathcal{V})$.*

Soit σ un arbre fini non vide.

- Si $\mathrm{Top}\,(\sigma)$ est une variable α, alors $dom\,(\sigma) = \{\epsilon\}$ et on note $\langle\alpha\rangle$ l'arbre élémentaire σ.
- Sinon $\mathrm{Top}\,(\sigma)$ est un symbole f et le domaine de σ est égal à :

  $$dom\,(\sigma) = \{\epsilon\} \cup \{xu \mid x \in [1, \varrho(\mathrm{Top}\,(\sigma))] \wedge u \in dom\ \sigma_{/i}\}$$

  On note $\langle f(\sigma_{/i})_{i\in[1,\varrho(f)]}\rangle$ l'arbre σ.

A un arbre fini non vide σ correspond donc un terme φ récursivement défini par :

- $\varphi(\langle\alpha\rangle) = \alpha$
- $\varphi(\langle f(\sigma_{/i})_{i\in[1,p]}\rangle) = f(\varphi(\sigma_{/i}))_{i\in[1,p]}$

Nous désignerons les variables par les lettres α, β, γ et δ et les termes ou arbres par les lettres σ, τ, ρ et π.

Désormais, nous ne considérerons que des arbres finis ou termes.

Définition 1.10 Une pondération de l'ensemble de symboles $\mathcal{C}$ est une application de $\mathcal{C}$ dans $\mathbb{N}$ telle que tout symbole d'arité non nulle ait un poids non nul. Soit p une pondération. La *taille* pour la pondération p est l'application Θ de $\mathcal{T}(\mathcal{C}, \mathcal{V})$ dans $\mathbb{N}$, récursivement définie par :

1. $\Theta(\mathcal{V}) = 0$
2. $\Theta(f(\sigma_i)_{i\in[1,\varrho(f)]}) = p(f) + \sum_{i\in[1,\varrho(f)]} \Theta(\sigma_i)$

□

La taille d'un sous-arbre est toujours strictement plus petite que la taille de l'arbre entier. Par défaut, la pondération est la fonction constante 1.

1.2.4 Substitutions

Des résultats généraux sur les substitutions peuvent être trouvés dans [21].

Définition 1.11 Une substitution est une $\mathcal{K}$-application totale de $\mathcal{V}$ dans le $\mathcal{K}$-ensemble $\mathcal{T}$ égale à l'identité presque partout. Une substitution s'étend donc en un homomorphisme dans $\mathcal{T}$. Les substitutions seront désignées par les lettres μ, ν et ξ. L'ensemble des substitutions est noté $\mathcal{S}$. La composition des substitutions μ et ν est notée $\nu \circ \mu$ ou de façon concise $\mu\nu$. L'application de la substitution μ au σ est notée $\mu(\sigma)$ ou de façon concise $\sigma\mu$. Avec les notations concises, on a

$$(\sigma\nu)\xi = \sigma(\nu\xi),$$

et l'on omettra les parenthèses.

Le *domaine* d'une substitution μ égal à

$$\{\alpha \in \mathcal{V} \mid \alpha\mu \neq \alpha\},$$

est noté $D(\mu)$. La substitution de domaine vide est notée id. On note $(\alpha \mapsto \sigma)$ la substitution élémentaire de domaine $\{\alpha\}$ qui associe le terme σ à α. L'ensemble des *variables introduites* égal à $\mathcal{V}(D(\mu)\mu)$ est noté $I(\mu)$. On notera aussi

$$\mu : V \to W \iff D(\mu) \subset V \wedge \mu(V) \subset W.$$

La restriction d'une substitution μ à un ensemble de variables W est la substitution égale à μ sur $D(\mu) \cap W$ et l'identité partout ailleurs.

Un *renommage* est une substitution dont l'image est incluse dans $\mathcal{V}$ et injective sur son domaine. Les renommages seront représentés par θ et η. La réciproque d'un renommage θ est égale à θ^{-1} sur $I(\mu)$ et l'identité partout ailleurs. Par abus on la notera simplement θ^{-1}. On a seulement l'égalité

$$\theta^{-1} \circ \theta \restriction D(\theta) = id \qquad \theta \circ \theta^{-1} \restriction I(\theta) = id$$

La *somme* de deux substitutions μ et ν de domaines disjoints est la substitution notée $\mu + \nu$ définie par :

$$\alpha \mapsto \begin{cases} \alpha\mu & \text{si } \alpha \in dom\,(\mu) \\ \alpha\nu & \text{si } \alpha \in dom\,(\nu) \\ \alpha & \text{sinon} \end{cases}$$

□

1.2.5 Algèbres de termes ordo-sortée

Pour une classification des algèbres multi-sortées on pourra se reporter à [53, 54]. Des résultats généraux sur les algèbres ordo-sortées sont présentés dans [57, 54]. Un formalisme plus proche de celui que nous utiliserons est décrit dans [38].

Les présentations que nous avons considérées précédemment sont uni-sortées. On peut étendre les définitions à des présentations multi-sortées, en remplaçant la fonction de signature par une relation quelconque, mais on demandera souvent que l'arité reste une fonction. Les algèbres obtenues sont dites multi-sortées.

Dans la suite on mentionnera dans des remarques ou exemples les algèbres ordo-sortées. Une présentation ordo-sortée est une présentation multi-sortée munie d'un ordre partiel $\leq$ sur les sortes.

Informellement, si $\mathcal{K}$ est un ensemble de sortes, Σ une signature ordo-sortée sur $\mathcal{K}$, et $\mathcal{V}$ un $\mathcal{K}$-ensemble de variables, on construit les Σ-termes en respectant les sortes comme pour construire les arbres finis, mais on a maintenant pour tout terme σ la règle

$$\frac{\sigma : \iota \quad \iota \leq \kappa}{\sigma : \kappa}$$

Une Σ-substitution μ est encore définie comme une fonction totale de $\mathcal{V}$ dans $\mathcal{T}$ respectant les sortes, c'est-à-dire pour toute variable α,

$$\alpha : \iota \Longrightarrow \alpha\mu : \iota$$

en tenant compte bien sûr de la règle précédente.

1.2.6 Théories équationnelles et unification

Des résultats généraux sur les théories équationnelles et l'unification peuvent être trouvés dans [22, 56] et [40].

Définition 1.12 Soit A un ensemble de couples de termes homogènes, notés $\sigma \doteq \tau$ et appelés axiomes. La théorie équationnelle A est la plus petite congruence sur $\mathcal{T}$, notée $=_A$ et appelée A-égalité, contenant toutes les paires $\sigma\mu \doteq \tau\mu$, où $\sigma \doteq \tau$ parcourt A et μ parcourt $\mathcal{S}$. On dit également que A est une *présentation* de la théorie équationnelle A.

La théorie A est *régulière* si deux termes A-égaux ont toujours mêmes ensembles de variables. Elle est *potente* s'il existe un terme variable A-égal à un terme non variable. Elle est *triviale* si tous les termes sont A-égaux. □

La A-égalité est étendue aux substitutions par :

$$\mu =_A \nu \iff \forall \alpha \in \mathcal{V}, \alpha\mu =_A \alpha\nu$$

On écrit aussi pour tout ensemble de variables W,

$$\mu =_A^W \nu \iff \forall \alpha \in W, \alpha\mu =_A \alpha\nu$$

Une substitution μ est A-plus petite qu'une substitution ν sur l'ensemble de variables W,

$$\mu <_A^W \nu \iff \exists \xi \in \mathcal{S}, \mu\xi =_A^W \nu$$

Par défaut W est l'ensemble $\mathcal{V}$ tout entier et est omis.

Théorie vide

Deux termes σ et τ sont dits *α-équivalents*, et on note $\sigma \equiv \tau$ si chacun est plus petit que l'autre.

Deux termes σ et τ sont dits *unifiables* s'il existe une substitution μ, appellée un *unificateur* de σ et τ telle que $\sigma\mu$ et $\tau\mu$ soient égaux. Pour tous termes σ et τ unifiables, il existe un plus petit unificateur de σ et τ appelé *unificateur principal*.

Nous donnerons une définition plus générale valable également dans les théories équationelles dans le chapitre suivant.

1.2.7 Systèmes de réécriture

Les propriétés générales des systèmes de réécriture et leur utilisation dans des théories équationnelles sont rassemblées dans [29, 19]

Définition 1.13 Une *règle de réécriture* est un couple homogène de termes, noté $g \to r$. Un *système de réécriture* est un ensemble $\mathcal{R}$ de règles de réécritures. □

Définition 1.14 Soit $\mathcal{R}$ un système de réécriture. Un terme σ se $\mathcal{R}$-réduit en une étape sur un terme τ si

$$\exists g \to d \in \mathcal{R},\ \exists \mu \in \mathcal{S},\ \exists u \in \mathcal{O}(\sigma),\ \sigma_{/u} = g\mu \wedge \tau = \sigma[d\mu/u].$$

et on note $\sigma \xrightarrow{\mathcal{R}} \tau$. On note $\xrightarrow{\mathcal{R}\,*}$ la fermeture transitive de $\xrightarrow{\mathcal{R}}$. □

Définition 1.15 Soit $\mathcal{R}$ une relation sur un ensemble E et $\mathcal{R}^*$ sa fermeture transitive.

1. $\mathcal{R}$ est *noethrienne* s'il n'existe pas de chaîne infinie $x\,\mathcal{R}\,y\,\mathcal{R}\ldots$.
2. $\mathcal{R}$ est *localement confluente* si
$$\forall (x,y,z) \in E^3,\ x\,\mathcal{R}\,y \wedge x\,\mathcal{R}\,z \Longrightarrow (\exists t \in E,\ y\,\mathcal{R}^*\,t \wedge z\,\mathcal{R}^*\,t)$$
3. $\mathcal{R}$ est *confluente* si
$$\forall (x,y,z) \in E^3,\ x\,\mathcal{R}^*\,y \wedge x\,\mathcal{R}^*\,z \Longrightarrow (\exists t \in E,\ y\,\mathcal{R}^*\,t \wedge z\,\mathcal{R}^*\,t)$$

□

Proposition 1.2 Théorème de Newmann *Une relation localement confluente et nœthérienne est confluente.*

L'étude de la confluence locale d'une relation de réécriture peut se ramener à l'étude de la relation sur un ensemble restreint de termes.

Définition 1.16 Soit deux règles $g \to d$ et $g' \to d'$. Supposons qu'il existe une occurrence u de g telle que $g_{/u}$ ne soit pas une variable et $g_{/u}$ et g' soient unifiables. Soit μ un unificateur principal de $g_{/u}$ et g'. Ces deux règles déterminent la paire critique $(g[d'\mu/u]\mu, d\mu)$. □

Proposition 1.3 *Une relation de réécriture $\mathcal{R}$ est localement confluente si, quelle que soit la paire critique (σ,τ) entre deux règles de $\mathcal{R}$, il existe un terme ρ tel que*
$$\sigma \xrightarrow{\mathcal{R}\,*} \rho \qquad \wedge \qquad \tau \xrightarrow{\mathcal{R}\,*} \rho.$$

Définition 1.17 Un système de réécriture est dit *canonique* si il est confluent et nœthérien. □

Chapitre 2

Unification dans les théories équationnelles

Ce chapitre présente le formalisme qui sera utilisé tout au long de cette thèse pour extraire des algorithmes d'unification. C'est essentiellement une reformulation des résultats de C. Kirchner sur la standardisation de l'unification équationnelle [36, 37], mais ceux-ci sont présentés dans un cadre un peu plus abstrait, puis appliqués à des cas particuliers.

Ce chapitre se compose de cinq parties. Dans la première partie nous introduisons une notion d'unificande plus abstraite que celle habituellement utilisée. Notre but n'est pas d'étudier les propriétés abstraites de ces unificandes comme dans [55], mais simplement de pouvoir considérer comme unificandes différentes sortes d'objets et les mélanger harmonieusement. Dans la seconde partie nous décrivons des propriétés très générales sur ces unificandes, indépendantes de la théorie équationnelle considérée.

Dans les deux parties suivantes, nous considérons des unificandes particuliers. Nous retrouvons d'abord les résultats usuels pour l'ensemble des multi-équations de termes d'une Σ-algèbre, puis nous étudions les unificandes hiérarchisés qui sont un cas très particulier d'algèbre ordo-sortée. Les unificandes hiérarchisés contraints permettront un calcul incrémental sur les sortes, ce qui sera à la base de l'algorithme efficace de synthèse de types pour ML présenté dans le chapitre suivant.

Dans la cinquième partie nous étudions les théories syntaxiques, en reprenant les définitions de [36], parfois de façon plus atomique, mais l'ajout essentiel est l'introduction d'un formalisme d'étude de conditions suffisantes pour qu'une présentation équationnelle soit résolvante ; nous appliquons cette méthode dans un cas particulier qui sera utilisé dans le chapitre 4.

La présentation étant devenue très différente de celle de [36], nous regrettons de ne pouvoir rappeler les résultats de C. Kirchner sans changer les notations, ce qui risquerait trop de perdre le lecteur.

Hypothèses Dans tout ce chapitre, nous considérons une présentation Σ de $\mathcal{C}$ sur $\mathcal{K}$ et la Σ-algèbre $\mathcal{T}(\mathcal{C}, \mathcal{V})$. L'ensemble de toutes les substitutions est noté $\mathcal{S}$.

2.1 Unificandes

2.1.1 Substitutions admissibles

Les algèbres avec sortes ne permettent pas toujours de contrôler de façon suffisamment précise l'ensemble des termes que l'on peut construire. Les algèbres ordo-sortées permettent de combler cette lacune. Dans toute cette partie nous ne voulons pas tant contrôler la formation des termes que l'ensemble des substitutions que l'on veut autoriser. La définition suivante plus abstraite que celle d'algèbre ordo-sortée ne cherche pas à définir un nouveau concept, mais simplement à préciser les conditions nécessaires pour les preuves qui suivent.

Définition 2.1 Un sous-ensemble $\mathcal{S}_a$ de $\mathcal{S}$ est dit *admissible* si :

1. $\forall \mu \in \mathcal{S}_a,\ \forall \nu \in \mathcal{S}_a,\ \mu\nu \in \mathcal{S}_a$

2. $\forall \mu \in \mathcal{S}_a,\ \forall W \subset \mathcal{V},\ \mu \restriction W \in \mathcal{S}_a$

3. Pour toute variable α il existe une infinité de variables β telles que $(\alpha \mapsto \beta, \beta \mapsto \alpha)$ soient dans $\mathcal{S}_a$.

Les éléments de $\mathcal{S}_a$ sont alors appelés *substitutions admissibles.* □

La dernière condition est une condition technique permettant d'assurer qu'il y a suffisamment de renommages admissibles :

Définition 2.2 On appelle *renommage admissible* tout renommage θ de $\mathcal{S}_a$ tel que θ^{-1} soit aussi dans $\mathcal{S}_a$. □

Définition 2.3 On appelle *copie* d'un ensemble de variables V tout ensemble $\mathcal{K}$-isomorphe à V par un $\mathcal{K}$-isomorphisme φ dont toutes les restrictions à des parties finies sont des renommages admissibles. On appelle *protection* d'un ensemble fini de variables W tout ensemble fini de variables contenant W. □

La dernière condition de la définition précédente assure que tout ensemble de variables W_0 puisse être copié en dehors d'un ensemble fini W. En effet, comme $\mathcal{V}$ est dénombrable les variables de W_0 peuvent être numérotées à partir de 0. Pour chaque variable α_i, il existe une variable β_i prise en dehors de W et distincte de toutes les variables α_j et β_j pour j strictement plus petit que i, telle que $(\beta_i \mapsto \alpha_i, \alpha_i \mapsto \beta_i)$ soit admissible. La fonction $\varphi : \alpha_i \mapsto \beta_i$ copie W_0 en dehors de W.

Proposition 2.1 *Lorsqu'elle est définie la somme de deux substitutions admissibles est admissible.*

Démonstration Soit μ et ν deux substitutions de domaines disjoints. Soit W une protection de $D(\nu)$, $I(\mu)$ et $I(\nu)$. L'ensemble $I(\mu) \cap D(\nu)$ peut être copié en dehors de W par une substitution ξ. On a

$$\mu + \nu = (\mu\xi \restriction D(\mu))\nu\xi^{-1} \restriction D(\mu) \cup D(\nu)$$

■

Exemple 2.1 Si Σ est une présentation de $\mathcal{C}$ sur $\mathcal{K}$, et $\mathcal{V}$ un $\mathcal{K}$-ensemble de variables, l'ensemble des Σ-substitutions dans l'ensemble $\mathcal{T}(\mathcal{C}, \mathcal{V})$ peut être considéré comme un ensemble de substitutions admissibles dans l'algèbre non sortée associée. C'est encore vrai pour une signature Σ ordo-sortée. La réciproque est fausse.

En effet, soit $\mathcal{V}$ un ensemble infini dénombrable de variables et $\mathcal{C}$ l'ensemble composé de deux symboles monadiques f et g. Privilégions une variable α et deux substitutions μ et ν respectivement égales à $(\alpha \mapsto f(f(\alpha)))$ et $(\alpha \mapsto f(g(\alpha)))$. La fermeture par composition et restriction de l'ensemble des deux substitutions μ et ν et de tous les renommages de $\mathcal{S}$ est un ensemble admissible. Il n'existe aucune signature ordo-sortée Σ pour laquelle l'ensemble des Σ-substitutions serait égal à l'ensemble admissible précédent.

On peut cependant considérer l'algèbre $\mathcal{T}(\mathcal{C}', \mathcal{V})$ où $\mathcal{C}'$ est composé de deux symboles monadiques f' et g' et la fonction φ égale à l'identité sur $\mathcal{V}$ et étendue à une fonction de $\mathcal{T}'$ dans $\mathcal{T}$ par

$$\varphi(f'(\sigma)) = f(f(\varphi(\sigma))), \qquad \varphi(g'(\sigma)) = f(g'(\varphi(\sigma))).$$

L'ensemble admissible précédent est alors égal à l'ensemble des substitutions $\varphi \circ \mu'$ où μ parcourt l'ensemble des substitutions de $\mathcal{T}'$.

Dans ce qui suit, $\mathcal{S}_a$ désignera un sous-ensemble admissible de $\mathcal{S}$. Nous considérons également un ensemble d'axiomes A. La théorie équationnelle $\mathcal{S}_a$-A sera la plus petite congruence sur $\mathcal{T}$, contenant toutes les paires $\sigma\mu \doteq \tau\mu$, où $\sigma \doteq \tau$ parcourt A et μ parcourt $\mathcal{S}_a$. Nous garderons les mêmes notations que précédemment, mais en ajoutant le préfixe $\mathcal{S}_a$ lorsque nous voudrons rappeler que $\mathcal{S}_a$ est restreint aux substitutions admissibles. La $\mathcal{S}_a$-A-égalité est en générale une relation plus restrictive que la $\mathcal{S}$-égalité. Aussi, on écrit pour tout ensemble de variables W,

$$\mu <^W_{\mathcal{S}_a, A} \nu \iff \exists \xi \in \mathcal{S}_a, \mu\xi =^W_A \nu.$$

2.1.2 Unificandes

Hypothèses Dans cette partie, nous nous plaçons par défaut dans une $\mathcal{S}_a$-A-théorie, toutes les substitutions et les opérations sur ces substitutions seront donc implicitement $\mathcal{S}_a$-admissibles.

Un unificande est habituellement défini comme une paire de termes encore appelée équation d'unification et notée $\sigma \doteq \tau$. Un problème d'unification $\sigma \doteq \tau$ consiste à trouver toutes les substitutions μ qui vérifient $\sigma\mu =_A \tau\mu$. Nous utiliserons une définition plus abstraite qui permettra dans les exemples ci-dessous de considérer comme problème d'unification la recherche d'un ensemble de substitutions permettant de valider un prédicat défini sur un ensemble d'objets plus généraux.

Définition 2.4 Pour tout ensemble non vide E muni d'une application var de E dans l'ensemble des parties finies de $\mathcal{V}$, d'une application sub de $E \times \mathcal{S}_a$ dans E et d'un prédicat val sur E appelée *validation*, on dit que (E, var, sub, val) est un *ensemble d'unificandes* si, pour tout élément U de E et toutes substitutions μ et ν de $\mathcal{S}_a$,

1. $var(U) \cap D(\mu) = \emptyset \Longrightarrow sub(U, \mu) = U$,
2. $var(sub(U, \mu)) = \mathcal{V}(var(U)\mu)$,
3. $sub(sub(U, \mu), \nu) = sub(U, \mu\nu)$,
4. $val(U) \Longrightarrow val(sub(U, \mu))$,
5. $\mu =_A \nu \Longrightarrow val(sub(U, \mu)) = val(sub(U, \nu))$.

Si (E, var, sub, val) est un ensemble d'unificandes élémentaires, on note $\mathcal{U}_A^{\mathcal{S}_a}(U)$ l'ensemble des A-solutions de l'unificande U égal à

$$\{\mu \in \mathcal{S}_a \mid val(sub(U, \mu))\}$$

□

Exemple 2.2 On retrouve les équations d'unification dans la théorie vide en considérant l'ensemble $\mathcal{T} \times \mathcal{T}$ dont les éléments seront notés $\sigma \doteq \tau$, qui muni des opérations

1. $var(\sigma \doteq \tau) = \mathcal{V}(\sigma) \cup \mathcal{V}(\tau)$,
2. $sub((\sigma \doteq \tau), \mu) = (\sigma\mu \doteq \tau\mu)$,
3. val est l'égalité sur $\mathcal{T}$.

est un ensemble d'unificandes.

Exemple 2.3 Si $\mathcal{P}_f(\mathcal{V})$ est l'ensemble des parties finies de $\mathcal{V}$, l'ensemble $\mathcal{P}_f(\mathcal{V}) \times \mathcal{T} \times \mathcal{T}$ muni des opérations

1. $var(W, \sigma \doteq \tau) = (\mathcal{V}(\sigma) \cup \mathcal{V}(\tau)) \setminus W$
2. $sub((W, \sigma \doteq \tau), \mu) = (W, \sigma(\mu \restriction \overline{W}) \doteq \tau(\mu \restriction \overline{W}))$
3. $val(W, \sigma \doteq \tau) = (\sigma = \tau)$.

est également un ensemble d'unificandes.

Exemple 2.4 Dans le chapitre suivant, nous définirons pour le langage ML des relations de typage $\Gamma \vdash M : \sigma$, qui sont des triplets formés d'un contexte Γ, d'un terme M du langage et d'un type σ. Nous définirons des opérations var, sub et une opération val. Nous montrerons que l'ensemble des relations de typage muni de ces opérations est un ensemble d'unificandes. Cette application est une des motivations essentielles pour les définitions générales qui précèdent.

Remarque 2.5 Une présentation abstraite de l'unification est également faite dans [55] par M. Schmidt-Schauß et J. H. Siekmann. Une d'*algèbre d'unification* est un triplet (V, OBJ, MAP) formé d'un ensemble de variables V, d'un ensemble d'objets OBJ et d'un ensemble de substitutions MAP vérifiant certains axiomes[1]. Cette abstraction se détache de la structure des termes tout en permettant de retrouver la plupart des propriétés générales sur les unificandes classiques, mais considère toujours un problème d'unification comme la recherche pour un ensemble fixé d'objets, de l'ensemble des substitutions rendant tous les objets égaux.

Notre définition d'unificande abstrait le prédicat *val* permettant de tester la satisfiabilité d'un unificande. L'exemple 2.4 ne peut pas être exprimé dans le formalisme des algèbres d'unification. Les conditions dans les définitions d'ensembles d'unificandes et d'ensembles de substitutions admissibles sont très voisines des axiomes des algèbres d'unifications. Il serait intéressant d'ajouter aux algèbres d'unification un prédicat de validation ; nos unificandes en seraient alors une instance. Cependant notre but est plus pragmatique. Nous voulons simplement fournir un outil nous permettant de traiter de façon uniforme les différents problèmes d'unification de cette thèse.

Définition 2.5 On appelle *conjonction* [respectivement *disjonction*] des ensembles d'unificandes (E, var, sub, val) et (E', var', sub', val'), l'ensemble d'unificandes $(E'', var'', sub'', val'')$ défini par :

1. $E'' = E \times E'$,
2. $var''(U, U') = var(U) \cup var'(U')$,
3. $sub''((U, U'), \mu) = (sub(U, \mu), sub'(U, \mu))$,
4. $val''(U, U') = val(U) \wedge val(U')$,
   [respectivement $val''(U, U') = val(U) \vee val(U')$].

On appelle auto-conjonction [respectivement auto-disjonction] de (E, var, sub, val), la conjonction [respectivement disjonction] de (E, var, sub, val) avec lui-même. □

Il est immédiat de vérifier que la conjonction et la disjonction d'unificandes satisfont les propriétés de la définition 2.4.

Notation Lorsque E et E' sont disjoints, ou lorsque (E, var, sub, val) et (E', var', sub', val') sont égaux, il n'y a pas ambiguïté sur le choix des fonctions var, sub ou du prédicat val et on notera $U \sqcap U'$ un couple de $E \times E'$ considéré comme conjonction d'ensemble d'unificandes. On dira que $U \sqcap U'$ est la conjonction des unificandes U et U'. Similairement on notera $U \sqcup U'$ la disjonction des unificandes U et U'. On écrira aussi $\mathcal{V}(U)$ au lieu de $var(U)$, et $U\mu$ au lieu de $sub(U, \mu)$.

Définition 2.6 Deux unificandes U et U' sont *A-équivalents* si et seulement ils ont le même ensemble de A-solutions. On note alors $U \Longleftrightarrow_A U'$. □

Proposition 2.2 *L'A-équivalence est une relation d'équivalence.*

Proposition 2.3 *Pour tous unificandes U, U' et U'', les propriétés*

- $U \sqcap U' \Longleftrightarrow_A U' \sqcap U$
- $(U \sqcap U') \sqcap U'' \Longleftrightarrow_A U \sqcap (U' \sqcap U'')$
- $U \sqcap (U' \sqcup U'') \Longleftrightarrow_A (U \sqcup U') \sqcap (U \sqcap U'')$

ainsi que les trois propriétés obtenues en inversant les connecteurs $\sqcap$ et $\sqcup$ sont vérifiées.

Démonstration La démonstration est immédiate à partir des définitions. ■

On considérera parfois les équivalences précédentes comme des égalités, ce qui n'est pas gênant. Elles permettent de ramener tout unificande à une disjonction de conjonctions. On parlera alors d'unificandes *composés* , et d'unificandes *élémentaires* pour ceux qui ne comportent pas de conjonctions ni de disjonctions.

[1] Pour plus de détail, on pourra se reporter à [55].

Les transformations A-équivalentes ne sont pas suffisantes pour simplifier les unificandes, parce qu'il est souvent nécessaire d'introduire de nouvelles variables. L'essentiel est de pouvoir reconstruire l'ensemble des A-solutions d'un unificande à partir de son unificande transformé. La notion essentielle est celle d'ensemble complet d'unificateurs.

Définition 2.7 Soit U un unificande protégé par W. Σ est un *ensemble complet d'unificateurs* de U en dehors de W si:

1. $\forall \mu \in \Sigma,\ (D(\mu) \subset \mathcal{V}(U)) \wedge (I(\mu) \cap W = \emptyset)$
2. $\Sigma \subset \mathcal{U}_A^{\mathcal{S}_a}(U)$
3. $\forall \nu \in \mathcal{U}_A^{\mathcal{S}_a}(U),\ \exists \mu \in \Sigma,\ (\mu <_{\mathcal{S}_a,A}^{\mathcal{V}(U)} \nu)$
4. De plus, Σ est dit *minimal* si

   $\forall \mu,\ \nu \in \Sigma, (\mu <_{\mathcal{S}_a,A}^{\mathcal{V}(U)} \nu \Longrightarrow \mu = \nu)$

Par défaut W est égal à $\mathcal{V}(U)$. Si $\{\mu\}$ est un ensemble complet d'unificateurs de U en dehors de W, on dit aussi que μ est un *unificateur principal* de U en dehors de W. Une théorie est *unitaire* si tout unificande admet un ensemble complet et minimal d'unificateurs possédant au plus un élément. □

Proposition 2.4 *Si Σ et Θ sont deux ensembles complets d'unificateurs en dehors de W, pour les unificandes respectifs U et U' protégés par W, alors $\Sigma \cup \Theta$ est un ensemble complet d'unificateurs de $U \sqcup U'$ en dehors de W.*

Démonstration La démonstration est immédiate. Il n'y a bien sûr pas de résultat analogue pour la conjonction. ■

Nous introduisons une transformation plus générale que l'A-équivalence, qui préserve les ensembles complets d'unificateurs, à une restriction près.

Définition 2.8 On dit qu'un unificande U' *A-étend* un unificande U et on note $U' \Rrightarrow_A U$ si pour tout ensemble complet Σ de A-solutions de U' en dehors de $\mathcal{V}(U)$ et $\mathcal{V}(U')$, l'ensemble $\Sigma \restriction \mathcal{V}(U)$ des restrictions des éléments de Σ à $\mathcal{V}(U)$ est un ensemble complet de A-solutions de U. Si de plus W est une protection de U telle que:

$$(\mathcal{V}(U') \setminus \mathcal{V}(U)) \cap W = \emptyset,$$

on dit que U' A-étend U en dehors de W et on note $U' \Rrightarrow_A^W U$. □

La recherche d'un ensemble complet d'unificateur d'un unificande U pourra se faire par la recherche d'un unificande U', A-étendant U et plus simple que U, sur lequel on saura immédiatement reconnaître un ensemble complet d'unificateurs Θ. On en déduira ensuite un ensemble complet d'unificateur Σ pour U, simplement par restriction des domaines des substitutions de Θ.

Remarque 2.6 Si U' et U ont même ensemble de variables, alors un ensemble complet d'unificateurs de U est un ensemble complet d'unificateurs de U'. Ces deux unificandes ont donc même ensemble de A-solutions et l'A-extension est en fait une A-équivalence.

Proposition 2.5 (transitivité) *Pour tous unificandes U, U', et U'', et tous ensembles de variables W et W', si*

$$U'' \Rrightarrow_A^{W'} U' \wedge U' \Rrightarrow_A^W U \wedge W' \subset W$$

alors $U'' \Rrightarrow_A^{W'} U$.

Démonstration La démonstration est immédiate. ■

On pourra donc former des chaînes d'A-extension

$$U_p \Rrightarrow_A^{W_p} \ldots U_1 \Rrightarrow_A^{W_1} U_0$$

satisfaisant

$$W_1 \subset \ldots W_p$$

et en déduire que $U_p \Rrightarrow_A^{W_1} U_0$. Le lemme suivant donne une caractérisation de l'A-extension plus facile à manipuler que la définition; nous l'utiliserons par la suite pour trouver des A-extensions élémentaires.

Lemme 2.6 *Si W est une protection de $\mathcal{V}(U)$ et W' une protection de W et $\mathcal{V}(U')$, une caractérisation de $U' \equiv\!\!\rangle_A^W U$ est :*

1. $(\mathcal{V}(U') \setminus \mathcal{V}(U)) \cap W = \emptyset$
2. *tout unificateur de U' en dehors de W' est un unificateur de U,*
3. *tout unificateur de U de domaine inclus dans W d'image en dehors de W' peut être étendu en dehors de W en un unificateur de U'.*

Démonstration Supposons que $U' \equiv\!\!\rangle_A^W U$. Montrons les trois conditions ci-dessus. La première est incluse dans l'hypothèse. La seconde est immédiate en prenant pour ensemble complet d'unificateurs de U' l'ensemble Σ' de tous les unificateurs en dehors de W' et de domaine inclus dans $\mathcal{V}(U')$.

Soit μ un unificateur de U de domaine inclus dans $\mathcal{V}(U)$ d'image en dehors de W'. En utilisant la définition de l'A-extension pour l'ensemble complet d'unificateurs Σ' ci-dessus, il est assuré qu'il existe un unificateur ν' de U' en dehors de W' de domaine inclus dans $\mathcal{V}(U')$ tel que sa restriction ν à $\mathcal{V}(U)$ soit plus petite que μ, c'est-à-dire qu'il existe ξ telle que $\mu =_A \nu\xi$. On a la décomposition :

$$\nu'\xi = \nu'\xi \restriction W + \nu'\xi \restriction \overline{W}$$

Le premier membre de la somme s'écrit encore, $(\nu' \restriction W)\xi \restriction W$. Or comme le domaine de ν' est inclus dans $\mathcal{V}(U')$ et $\mathcal{V}(U') \setminus \mathcal{V}(U) \cap W$ est vide, la restriction de μ' par W est aussi égale à sa restriction par $\mathcal{V}(U')$. Le premier membre de la somme est donc simplement $\nu\xi \restriction W$ qui est A-égal à $\mu \restriction W$ ou encore μ puisque le domaine de μ est inclu dans W. La substitution $\mu + \nu'\xi \restriction \overline{W}$ étend μ en dehors de W et unifie U'.

Réciproquement, supposons les trois conditions du lemme et montrons que $U' \equiv\!\!\rangle_A^W U$. Soit Σ un ensemble complet d'unificateurs de U' en dehors de W'. La deuxième condition du lemme implique que sa restriction à U est un ensemble d'unificateurs de U, en dehors de W'. La dernière condition prouve sa complétude.

En effet, soit μ un unificateur de U, ν sa restriction à W et θ un renommage des variables de $I(\nu)$ en dehors de W'. Alors $\nu\theta$ est un unificateur de U en dehors de W', il en existe donc une extension ν' en dehors de W qui soit un unificateur de U'. Comme Σ est un ensemble complet d'unificateurs de U' en dehors de W' il existe une substitution ξ' plus petite que ν' qui soit dans Σ. La restriction de ξ' à W est donc également plus petite sur W que la restriction de ν' à W, c'est-à-dire $\nu\theta$, elle-même équivalente à ν. Comme $\mathcal{V}(U') \cup \mathcal{V}(U)$ n'intercepte pas W, la restriction de ξ' à W est identique à sa restriction à U. En résumé ξ' est un élément de $\Sigma \restriction W$ plus petit sur W que μ.

Si Σ est un ensemble complet d'unificateurs de U' en dehors de $\mathcal{V}(U')$ et $\mathcal{V}(U)$, il existe une copie φ de $\mathcal{V}$ sur W' par laquelle Σ devient un ensemble complet d'unificateurs en dehors de W', sa restriction à $\mathcal{V}(U)$ est alors un ensemble complet d'unificateurs de U en dehors de W' dont l'image par φ^{-1} est un ensemble complet d'unificateurs en dehors de $\mathcal{V}(U')$ et $\mathcal{V}(U)$. ■

2.1.3 Transformations d'unificandes

Proposition 2.7 *Si U, U' et U'' sont trois unificandes, alors*

$$U \langle\!\!\equiv\!\!\rangle_A U' \Longrightarrow \wedge \begin{cases} U \sqcap U'' \langle\!\!\equiv\!\!\rangle_A U' \sqcap U'' \\ U \sqcup U'' \langle\!\!\equiv\!\!\rangle_A U' \sqcup U'' \end{cases}$$

Démonstration Découle directement des définitions de l'A-équivalence et des opérations de conjonction et disjonction. ■

Proposition 2.8 *Si U et U'' sont deux unificandes protégés par W et si U' est un unificande, alors*

$$U' \equiv\!\!\rangle_A^W U \Longrightarrow \wedge \begin{cases} U' \sqcap U'' \equiv\!\!\rangle_A^W U \sqcap U'' \\ U' \sqcup U'' \equiv\!\!\rangle_A^W U \sqcup U'' \end{cases}$$

Démonstration La disjonction se montre directement en utilisant la propriété 2.4. La conjonction se montre en utilisant la caractérisation du lemme 2.6. ■

2.2 Multi-équations

Définition 2.9 Une *multi-équation* est un multi-ensemble fini non vide de termes. Une *équation* est simplement une multi-équation réduite à deux termes. On notera les multi-équations par e, e' et e''. L'ensemble des multi-équations muni des opérations

1. $var = \mathcal{V}$,
2. $sub\,(e, \mu) = \{\sigma\mu \mid \sigma \in e\}$,
3. $val\,(e) \iff (\forall \sigma,\ \tau \in e, \sigma =_A \tau)$.

est un ensemble d'unificandes.

On note $\doteq$ l'union pour les multi-équations. On note également $V(e)$ pour $e \cap \mathcal{V}$ et $T(e)$ pour $e \setminus \mathcal{V}$ que l'on étendra aux unificandes composés par

$$V(\mathcal{M}) = \bigcup_{e \in \mathcal{M}} V(e), \qquad T(\mathcal{M}) = \bigcup_{e \in \mathcal{M}} T(e).$$

Une substitution μ est *A-solution* de la multi-équation e si pour tous éléments σ et τ de e, on a $\sigma\mu =_A \tau\mu$. □

Une auto-conjonction de multi-équations est également appelée un *système* et une auto-disjonction est simplement appelée une *disjonction* de systèmes. On se ramènera toujours à une disjonction de conjonctions. Les systèmes de multi-équations seront notés $\mathcal{M}$ et $\mathcal{N}$.

Nous allons maintenant étudier l'ensemble d'unificandes des multi-équations. Cette étude consiste essentiellement à découvrir des A-équivalences et A-extensions sur cet ensemble. La plupart des transformations sont locales dans le sens où elles sont définies à partir d'unificandes élémentaires. Grâce aux propriétés précédentes elles s'étendent naturellement à des unificandes composés. Aussi nous essayerons toujours de définir les transformations les plus élémentaires possibles.

Pour toute A-extension $U' \Rrightarrow_A^W U$ le remplacement de l'unificande U par l'unificande U' définit une transformation sur les unificandes. Une A-équivalence $U' \langle\Longrightarrow\rangle_A^W U$ dont on aura précisé une orientation est une A-extension. Elle définit donc également une transformation sur les unificandes. Par exemple, l'orientation de la gauche vers la droite correspond à l'A-extension ci-dessus, et à la transformation remplaçant le membre droit par le membre gauche[2].

On distingue essentiellement cinq types de transformations d'A-extension. La *généralisation* permet de diminuer la taille des termes dans les multi-équations en les "découpant" en leurs sous-termes. On introduit de nouvelles variables pour "nommer" les sous-termes. Cette transformation est utilisée en particulier lorsque l'on veut se ramener à des termes de taille au plus 1, par exemple afin d'effectuer un partage maximum des calculs. Elle ne doit pas être confondue avec la *décomposition* qui "décortique" simultanément deux termes d'une même multi-équation, toujours à partir de leur racine. La *collision* est un cas d'échec de la décomposition. Elle se produit lorsque deux termes d'une même multi-équation sont incompatibles. Les transformations précédentes s'appliquent à des multi-équations prises indépendamment. La *fusion* transforme deux multi-équations ayant des termes-variables en commun en une seule. Ces quatre transformations sont suffisantes dans la théorie vide pour toujours se ramener à une forme complètement décomposée. Dans une théorie non vide on regroupe sous le nom de *mutation* les autres transformations qui permettent d'aboutir à une forme complètement décomposée. Trouver des transformations de mutation nœthériennes est en général une difficulté essentielle dans l'étude d'une théorie équationnelle particulière.

Nous décrivons de façon précise ces cinq transformations.

[2] L'A-extension $U' \Rrightarrow_A U$ doit être lue comme une implication : *d'un ensemble complet d'unificateurs de U', je déduis un ensemble complet d'unificateurs de U*. Le but d'une transformation est de trouver une hypothèse de laquelle on puisse déduire la conclusion. L'inversion entre le sens de l'A-dépendance et le sens de la transformation n'est donc pas surprenant.

2.2.1 Généralisation

Proposition 2.9 (Généralisation) *Soit U un unificande et μ une substitution dont l'image est en dehors de son domaine. Alors,*

$$\left(U \sqcap \bigsqcap_{\alpha \in D(\mu)} (\alpha \doteq \alpha\mu)\right) \Rrightarrow_A^{D(\mu)} U\mu$$

Définition 2.10 On appelle *généralisation* la transformation d'A-extension précédente. □

Démonstration Si ν est un unificateur de U' on a l'égalité

$$\nu \restriction D(\mu) =_A \mu\nu \restriction D(\mu).$$

L'égalité complémentaire,

$$\nu \restriction \overline{D}(\mu) =_A \mu\nu \restriction \overline{D}(\mu),$$

est toujours vérifiée. Ainsi ν et $\mu\nu$ sont A-égales donc $\mu\nu$ est une solution de U, ou ce qui est revient au même, ν est un unificateur de $U\mu$. Ce raisonnement peut être retourné pour montrer qu'une solution ν de $U\mu$ peut être étendue en dehors de U par la solution $\mu\nu$. ■

Exemple 2.7 Considérons l'équation e égale à

$$f(ga, \alpha) \doteq f(b, b)$$

et la substitution μ définie par

$$\begin{cases} \beta' \mapsto ga \\ \alpha' \mapsto b \end{cases},$$

où les variables α' et β' sont distinctes de α et β. On peut écrire e comme

$$\big(f(\beta', \alpha) \doteq f(\alpha', \alpha')\big)\mu$$

Le système U' égal à

$$\bigsqcap \begin{cases} f(\beta', \alpha) \doteq f(\alpha', \alpha') \\ \alpha' \doteq ga \\ \beta' \doteq b \end{cases}$$

est une donc généralisation de U. En écrivant ga comme $g\beta''(\beta'' \mapsto a)$, on peut à nouveau le généraliser et obtenir un système composé seulement avec des termes de taille au plus 1.

Remarque 2.8 On utilise souvent une forme restreinte de la généralisation qui consiste à A-étendre une multi-équation $\sigma \doteq e$ protégée par W par le système :

$$(\sigma[\alpha/u] \doteq e) \sqcup (\alpha \doteq \sigma_{/u})$$

où α est une variable prise en dehors de W.

Exemple 2.9 Dans l'exemple précédent, on prendra plutôt deux variables distinctes α' et α'' pour les deux occurrences de b.

Corollaire 2.10 (Remplacement) *Par conjonction de la généralisation ci-dessus avec le système $\bigsqcup_{\alpha \in D(\mu)} (\alpha \doteq \alpha\mu)$, on obtient une nouvelle A-extension qui est en fait une A-équivalence, car les deux unificandes ont les mêmes variables.*

$$\left(U \sqcap \bigsqcap_{\alpha \in D(\mu)} (\alpha \doteq \alpha\mu)\right) \Lleftrightarrow_A \left(U\mu \sqcap \bigsqcap_{\alpha \in D(\mu)} (\alpha \doteq \alpha\mu)\right)$$

Exemple 2.10 Le système de multi-équations

$$(f(a, \alpha) \doteq c) \sqcap (\alpha \doteq c)$$

peut être remplacé par le système

$$(f(a, c) \doteq \beta) \sqcap (\alpha \doteq c) \tag{1}$$

qui est constitué de deux équations n'ayant pas de variables en commun.

2.2.2 Collision

Définition 2.11 On appelle *doublet de collision* tout couple de symboles (f, g) tel qu'un terme σ ayant le symbole f en tête et un terme τ ayant le symbole g en tête ne soient jamais A-égaux :

$$\forall (f,\ g) \in \mathcal{C}^c, \forall \sigma \in f(\mathcal{T}),\ \forall \tau \in g(\mathcal{T}),\ \sigma \neq_A \tau$$

On appelle *ensemble de collision* tout ensemble, noté $\mathcal{C}^c$, de doublets de collision. □

Remarque 2.11 La définition précédente ne demande pas que $\mathcal{C}^c$ soit maximal. La raison est simplement que l'on ne connaît pas de façon mécanique de déterminer l'ensemble de tous les doublets de collision.

Dans la plupart des cas, certains couples de symboles sont des doublets de collision de façon évidente, et on se contentera d'un ensemble de collision formé seulement de ces couples.

- Si la théorie est triviale, aucun doublet ne crée de collision.
- Si la théorie est non potente, alors on peut quotienter l'ensemble des symboles par la fermeture réflexive, symétrique et transitive de la relation :

$$\bigcup_{(\sigma \doteq \tau) \in A} (\mathrm{Top}\,(\sigma), \mathrm{Top}\,(\tau))$$

  Alors deux symboles pris dans des classes différentes forment un doublet de collision.
- Si la théorie est régulière, tout doublet formé de symboles différents n'apparaissant dans aucun membre d'axiome est un doublet de collision. Si la théorie est potente, il se peut qu'un symbole n'apparaissant dans aucun axiome à l'occurrence nulle ne crée pas de collision. Par exemple dans la théorie régulière $\{\alpha \doteq f(\alpha), \beta \doteq g(\beta)\}$, le couple (f, g) ne crée pas de collision.

Dans la théorie $\{(a \doteq a'), (b \doteq b')\}$, les couples (a, b) et (a', b) sont des doublets de collision, mais (a, a') n'est pas un doublet de collision. Il n'y a donc pas de sous-ensemble maximum de $\mathcal{C}$ tel que le produit cartésien soit un ensemble de collision.

Proposition 2.11 *Si (f, g) est un doublet de collision, σ un terme de $f(\mathcal{T})$ et τ un terme de $g(\mathcal{T})$, alors une multi-équation contenant à la fois σ et τ n'a pas de solution.*

Démonstration La démonstration est immédiate. S'il existait une solution μ, à la multi-équation e, alors $\sigma\mu$ et $\tau\mu$ seraient A-égaux, or $\sigma\mu$ est dans $f(\mathcal{T})$ et $\tau\mu$ est dans $g(\mathcal{T})$. ■

La collision permet de détecter qu'un système n'a pas de solution[3]. Plus l'ensemble de symboles de collision sera grand, plus cette détection sera efficace.

2.2.3 Décomposition

Définition 2.12 On appelle *symbole décomposable*[4] tout symbole f tel que

$$\forall \sigma,\ \tau \in f(\mathcal{T}), (\sigma =_A \tau \iff \forall i \in [1, \varrho(f)],\ \sigma_{/i} =_A \tau_{/i}).$$

On note $\mathcal{C}^d$ un ensemble de symboles décomposables. □

Remarque 2.12 On ne connaît pas de façon systématique de déterminer tous les symboles décomposables. On se contente donc en général d'un ensemble de symboles décomposables trivial. Dans le cas d'une théorie non potente tous les symboles qui n'apparaissent dans aucun axiome à l'occurrence nulle sont décomposables. Il n'y pas de lien simple entre les ensembles $\mathcal{C}^c$ et $\mathcal{C}^d$ comme le montrent les exemples suivants.

Exemple 2.13 Dans la théorie $\{(a \doteq a')\}$, tous les symboles sont décomposables, mais (a, a') n'est pas un doublet de collision.

Dans la théorie $\{(f(a) \doteq f(b))\}$, le doublet (f, a) est un doublet de collision mais f n'est pas décomposable.

[3] La réciproque est fausse ; la collision ne permet pas de s'assurer qu'un système admet des solutions.

[4] A la différence de [36], nous avons distingué la collision de la décomposition. Etre un symbole décomposable est donc ici une propriété plus faible que dans [36].

Définition 2.13 Soit f un symbole décomposable, σ et τ des termes ayant le symbole f en tête, et e une multi-équation. On appelle *décomposition* de la multi-équation[5] $\sigma \doteq \tau \doteq e$, son remplacement par le système de multi-équations

$$(\sigma \doteq e) \sqcap \prod_{i \in [1,p]} (\sigma_{/i} \doteq \tau_{/i}).$$

□

Proposition 2.12 *La décomposition est une A-équivalence.*

Démonstration La démonstration est immédiate. ■

Remarque 2.14 La décomposition d'un unificande fait décroître la taille du système (somme des tailles de ses termes). L'itération de la décomposition termine. Elle n'est pas confluente car le choix du terme conservé dans la multi-équation décomposée est arbitraire. On notera $Dec(U)$ un système décomposé obtenu à partir de U par décomposition.

Hypothèse Dans la suite, on supposera fixé un ensemble de collision $\mathcal{C}^c$ et un ensemble de symboles décomposables $\mathcal{C}^d$.

2.2.4 Fusion

Définition 2.14 Soit α une variable, e et e' deux multi-équations. On appelle *fusion* des multi-équations $\alpha \doteq e'$ et $\beta \doteq e''$, le remplacement du système $(\alpha \doteq e') \sqcap (\alpha \doteq e'')$ par la multi-équation $\alpha \doteq e \doteq e'$. □

Proposition 2.13 *La fusion est une A-équivalence.*

Démonstration La démonstration est immédiate. ■

Remarque 2.15 La fusion d'un unificande fait décroître le nombre de multi-équations. L'itération de l'opération de fusion à partir d'un unificande initial U termine, et est de toute évidence confluente. On note $Fus(\mathcal{M})$ le système obtenu.

2.2.5 Mutation

Le processus de fusion-décomposition fait décroître dans l'ordre lexicographique la taille de l'unificande, puis le nombre de multi-équations. L'itération de ce processus sur un unificande initial U conduit à un unificande décomposé et fusionné, noté *Dec-Fus*(U).

Lorsque $Card\,(T(e)) \leq 1$, on dit que la multi-équation e est *complètement décomposée*. Un unificande est complètement décomposé, lorsque toutes les multi-équations qui le composent sont complètement décomposées. Un unificande est dit *indécomposable* s'il n'est pas complètement décomposé et si les multi-équations qui le composent ne sont pas décomposables par fusion et décomposition et s'il ne comporte pas de collisions.

Exemple 2.16 La multi-équation e égale à $(\alpha \doteq \beta \doteq f(a) \doteq b)$ n'est pas complètement décomposée car elle contient les deux termes non variables $f(a)$ et b. Elle est indécomposable si (f, b) n'est pas un doublet de collision. Par exemple, dans la théorie vide, on peut choisir $\mathcal{C}^c$ égal à $\mathcal{C} \times \mathcal{C} \setminus \Delta(\mathcal{C})$ où $\Delta(\mathcal{C})$ est la diagonale de $\mathcal{C}$. Alors la multi-équation e n'est pas complètement décomposée, car il y a collision. L'ensemble de collision ci-dessus est maximal, mais il est possible d'en choisir un plus petit, auquel cas la multi-équation e peut être indécomposable.

Exemple 2.17 Dans la théorie $\{(f(f(\alpha)), g(\alpha))\}$, la multi-équation e, égale à $f(\alpha) \doteq f(\beta)$, est indécomposable mais pas complètement décomposée. En effet (f, g) ne peut pas être un doublet

[5] Nous rappelons que $\doteq$ est la réunion pour les multi-ensembles, donc la notation $\sigma \doteq e$ désigne la multi-équation formé de tous les termes de e et de σ.

de collision. Il est clair qu'une solution principale à cette multi-équation est $(\alpha \mapsto f\gamma, \beta \mapsto \gamma)$. Le système

$$(\alpha \doteq f\gamma) \sqcap (\beta \doteq \gamma)$$

est une A-extension de e, qui ne correspond à aucune des transformations précédentes, on dira que le système est obtenu par mutation de la multi-équation e.

Définition 2.15 Un unificande U indécomposable est dit *mutable* s'il existe un unificande noté $Mut(U)$ qui A-étend U. Une théorie est *mutable* si tout unificande indécomposable est mutable et si le processus de mutation-fusion-décomposition termine. □

Ce processus s'arrête soit sur une collision, soit sur un unificande complètement décomposé. Une multi-équation e qui compose un système complètement décomposé est de l'une des formes suivantes :

1. $T(e) = \emptyset$,
2. $T(e) = \{\sigma\}$ avec $V(e) \cap \mathcal{V}(\sigma) = \emptyset$,
3. $T(e) = \{\sigma\}$ avec $V(e) \cap \mathcal{V}(\sigma) \neq \emptyset$.

Nous allons voir que sous de bonnes conditions, le dernier cas peut se ramener à l'un des deux premiers, pour lesquels nous aurons des ensembles complets de solutions.

2.2.6 Elimination des cycles

Définition 2.16 Soit e et e' deux multi-équations. On dit que e est *liée* à e' et on note $e \preceq e'$ s'il existe un terme variable de e contenu dans un terme non variable de e',

$$e \preceq e' \iff V(e) \cap \mathcal{V}(T(e')) \neq \emptyset.$$

On note $\preceq_{\mathcal{M}}$ la fermeture transitive de $\preceq$ sur l'ensemble des multi-équations d'un système $\mathcal{M}$. □

Exemple 2.18 Dans le système de multi-équations

$$(\alpha \doteq f(\beta, \gamma)) \sqcap (\beta \doteq g(\beta)) \sqcap (\gamma \doteq a)$$

la dernière multi-équation est liée à la première, la deuxième est liée à elle-même et à la première.

Notation Une relation $\mathcal{R}$ entre multi-équations est étendue à un système par :

$$\mathcal{M}\ \mathcal{R}\ \mathcal{M}' \iff (\forall e \in \mathcal{M},\ \forall e' \in \mathcal{M}',\ e\ \mathcal{R}\ e')$$

Définition 2.17 Nous dirons qu'une suite de multi-équations $(e_i)_{1 \in [1,p]}$ est *ordonnée de l'extérieur vers l'intérieur* si

$$\forall i \in [1, p-1],\ \forall j \in [i, p],\ e_i \not\preceq e_j.$$

Elle est *ordonnée de l'intérieur vers l'extérieur* si la suite inversée $(e_{p-i})_{i \in [1,p]}$ est ordonnée de l'extérieur vers l'intérieur. □

Une bonne intuition est de lire *e' est lié à e* comme *e' est un sous-terme de e*.

Définition 2.18 Une théorie A est *stricte* si $\preceq_{\mathcal{M}}$ est un ordre strict pour tout système de multi-équations $\mathcal{M}$ admettant des A-solutions. □

Remarque 2.19 Si $\mathcal{M}$ est un système de multi-équations pour lequel $\preceq_{\mathcal{M}}$ est un ordre strict, alors $\mathcal{M}$ peut être ordonné de l'extérieur vers l'intérieur et de l'intérieur vers l'extérieur.

Définition 2.19 Une théorie A est *simple* si un terme n'est jamais A-égal à un de ses sous-termes. □

Proposition 2.14 *Une théorie A est stricte si et seulement si elle est simple.*

Démonstration Cette propriété est présentée dans [3, 5] et démontrée dans [4] :

Si A n'est pas simple, alors il existe un terme σ et un terme τ contenant le terme σ à une occurrence u tels que $\sigma =_A \tau$. Il suffit d'introduire une variable α prise en dehors de σ et τ. Alors la substitution $(\alpha \mapsto \sigma)$ unifie les termes α et $\tau(u \mapsto \alpha)$, donc A n'est pas stricte.

Si A n'est pas stricte, alors il existe un système de multi-équations $(e_i)_{i\in[1,p]}$ A-unifiable tel que l'on ait la chaîne de dépendance

$$e_1 \preceq \dots e_n \preceq e_1,$$

c'est-à-dire qu'il existe pour chaque i de $[1,p]$ une variable α_i et un terme σ_i tels que α_i apparaisse dans σ_{i+1} en comptant modulo n. Pour tout i de $[1,n]$, nous notons μ_i la substitution $(\alpha_i \mapsto \sigma_i)$ et σ le terme $\sigma_1\mu_n \dots \mu_1$. Le système de multi-équations admet un unificateur ν. En particulier $\alpha_i\nu = \sigma_i\nu$ pour tout i de $[1,n]$. En remplaçant dans $\sigma_1\nu$ toutes les occurrences du termes $\alpha_n\nu$ par le terme A-égal $\sigma_n\nu$ on obtient le terme $\sigma_1\mu_n\nu$. En répétant le processus on obtient ainsi le terme $\sigma\mu_n \dots \mu_1\nu$, c'est-à-dire $\tau\nu$ qui est A-égal à $\alpha_1\nu$. Or τ contient α_1, donc la théorie A n'est pas simple. ■

Définition 2.20 Un système complètement décomposé $\mathcal{M}$ est dit *libre* si deux multi-équations quelconques de $\mathcal{M}$ ne sont pas liées l'une à l'autre. □

Exemple 2.20 Le système (1) de l'exemple 2.10 est un système libre.

2.2.7 Résolution d'un système complètement décomposé

Définition 2.21 Soit $\mathcal{M}$ un système libre complètement décomposé. Il peut toujours s'écrire

$$\prod_{i\in[1,p]} (e_i \doteq \sigma_i)$$

où tous les ensembles $T(e_i)$ sont vides. La substitution de domaine inclus dans $V(\mathcal{M})$ qui pour tout i de $[1,p]$ envoie $V(e_i)$ sur le terme σ_i est appelé une *pré-solution* de $\mathcal{M}$ et est notée $[\mathcal{M}]$. □

Il existe en général plusieurs pré-solutions, car dans le cas où une multi-équation e ne contient que des variables, il y a autant de façon d'écrire e sous la forme $e_0 \doteq \sigma_0$ que de variables dans e.

Exemple 2.21 Si $\mathcal{M}$ est la multi-équation $\alpha \doteq \sigma$, une présolution de $\mathcal{M}$ est $(\alpha \mapsto \sigma)$. Cet exemple canonique justifie la notation $[\alpha \doteq \sigma]$ pour cette présolution.

Hypothèse Les propriétés qui suivent ne sont valables que lorsque toutes les substitutions sont admissibles, c'est-à-dire lorsque $\mathcal{S}_a$ est égal à $\mathcal{S}$ tout entier.

Proposition 2.15 *Soit $\mathcal{M}$ un système libre protégé par W. Si θ est un renommage des variables de $I([\mathcal{M}])$ en dehors de W, alors $[\mathcal{M}]\theta$ est une solution principale de $\mathcal{M}$ en dehors de W.*

Démonstration La substitution $[\mathcal{M}]$ est une solution de $\mathcal{M}$.

Soit e une multi-équation. Si $T(e)$ est vide, alors tous les termes de e sont des variables et ont une même image. Sinon la multi-équation e contient un seul terme σ. Toutes les variables ont σ pour image. Il est lui-même sa propre image. En effet, le système étant libre, σ ne peut contenir aucune variable qui soit dans $V(\mathcal{M})$. Or $V(\mathcal{M})$ contient le domaine de $[\mathcal{M}]$.

La composition de $[\mathcal{M}]$ avec θ est une solution en dehors de W. Réciproquement, une solution ν de $\mathcal{M}$ vérifie pour toute multi-équation $e \doteq \sigma$ de $\mathcal{M}$ et toute variable α de e

$$\alpha\nu =_A \sigma\nu,$$

Mais σ_i est A-égal à $\alpha[\mathcal{M}]$, d'où

$$\nu =_A^{D([\mathcal{M}])} [\mathcal{M}]\nu.$$

A fortiori, cette égalité se prolonge en dehors de $D(\mu)$. Ce qui prouve que ν est plus petite que $[\mathcal{M}]\theta$ par $\theta^{-1}\nu$. ■

Lemme 2.16 *Soit e une multi-équation et $\mathcal{N}$ un système de multi-équations tel que le système $e \sqcap \mathcal{N}$ soit complètement décomposé. Si $\mathcal{N} \not\preceq e$, alors*

- $\mathcal{N}[e] \sqcap e \Longleftrightarrow_A \mathcal{N} \sqcap e$
- $\mathcal{N}[e]$ *est complètement décomposé.*
- *pour toutes multi-équations* e' *et* e'' *de* $\mathcal{N}$,

$$e' \preceq e'' \iff e'[e] \preceq e''[e].$$

- $\mathcal{N}[e]$ *et* e *sont indépendants.*

Démonstration L'équivalence ci-dessus est un remplacement. Comme $\mathcal{M}$ est fusionné, toute multi-équation e' de $\mathcal{N}$ est telle que $V(e')$ et $V(e)$ n'aient pas de variables en commun. Comme le domaine de $[e]$ est inclus dans $V(e)$, la multi-équation e' n'est pas modifiée par $[e]$. Ainsi :

- $\mathcal{N}[e]$ est complètement décomposé.
- Pour toute multi-équation e'' de $\mathcal{N}$,

$$V(e') \cap \mathcal{V}(T(e'')) = V(e') \cap \mathcal{V}(T(e''[e]))$$

car ces deux ensembles n'interceptent pas le domaine de $[e]$.

■

Proposition 2.17 *Un système* $\mathcal{M}$ *complètement décomposé tel que l'ordre* $\preceq_{\mathcal{M}}$ *soit strict est A-équivalent à un système libre.*

Démonstration Il suffit d'appliquer le lemme précédent à toutes les multi-équations, dans un ordre de l'intérieur vers l'extérieur. ■

En combinant les propriétés 2.17 et 2.15, on obtient le corollaire.

Corollaire 2.18 *Tout système complètement décomposé et libre admet un* $\mathcal{S}$*-A-unificateur principal.*

On peut généraliser le lemme 2.16 à la proposition suivante.

Proposition 2.19 *Soient* $\mathcal{M}$ *et* $\mathcal{N}$ *deux systèmes tels que*

1. $\mathcal{M} \sqcap \mathcal{N}$ *soit complètement décomposé,*
2. $\mathcal{M} \not\preceq \mathcal{N}$,
3. $\mathcal{M}$ *et* $\mathcal{N}$ *soient libres.*

Alors $\mathcal{M}[\mathcal{N}] \sqcap \mathcal{N}$ *est un système libre équivalent à* $\mathcal{M} \sqcap \mathcal{N}$.

Démonstration Il suffit de voir que les remplacements utilisés pour libérer $\mathcal{M} \sqcap \mathcal{N}$ par le lemme ci-dessus forment exactement la substitution $[\mathcal{N}]$. ■

Avec les notations du corollaire ci-dessus, $[\mathcal{M}][\mathcal{N}]$ est une pré-solution de $\mathcal{M}[\mathcal{N}] \sqcap \mathcal{N}$. On n'a donc pas besoin de libérer $\mathcal{M} \sqcap \mathcal{N}$, mais simplement de savoir l'écrire dans un ordre de l'extérieur vers l'intérieur pour en connaître une pré-solution. On notera aussi $[\mathcal{M}]$ une pré-solution d'un système équivalent par libération à $\mathcal{M}$. Si $\preceq_{\mathcal{M}}$ est strict, on connaît donc un unificateur principal de $\mathcal{M}$ sans le libérer.

Théorème 2 *Pour toute théorie stricte, mutable telle que le processus de décomposition-fusion-mutation termine, il existe un algorithme qui calcule un ensemble complet de* $\mathcal{S}$*-A-solutions pour un problème d'unification quelconque.*

Un algorithme obtenu par décomposition-fusion-mutation-résolution est dit *standard*.

2.3 Unificandes hiérarchisés

Dans cette partie nous allons considérer un ensemble $\mathcal{S}_a$ particulier, dans lequel les substitutions devront faire décroître une fonction mesurant le degré d'un terme. Cette application trouvera son application dans le chapitre 3, dans le système d'inférence hiérarchique.

Très informellement, un des problèmes de la synthèse de types dans ML est la généralisation des variables libres, encore appelées variables fraîches, possible à tout instant. Les degrés permettront de mesurer la fraîcheur d'un terme. Lors de la fusion de deux objets, la fraicheur résultante devra être diminuée à la fraîcheur minimale de chacun des objets. Cette contrainte sera exprimée par le fait qu'une variable ne pourra être substituée que par un terme de moindre degré.

Hypothèse Nous supposons que tous les axiomes sont réguliers.

2.3.1 Hiérarchisation

Définition 2.22 Une *hiérarchie* est une fonction totale, notée ϕ, de $\mathcal{V}$ dans $\mathbb{N}$ telle que l'image réciproque de chaque entier soit $\mathcal{K}$-isomorphe[6] à $\mathcal{V}$. Elle est étendue en une opération de $\mathcal{T}$ dans $\mathbb{N}$ par

$$\phi(\sigma) = \sup_{\alpha \in \mathcal{V}(\sigma)} \phi(\alpha)$$

pour tout terme σ. La hiérarchie d'une constante est donc l'entier 0. □

Il est clair qu'il existe des hiérarchies. Dans toute cette partie, nous supposerons fixée une hiérarchie ϕ.

Définition 2.23 Le *degré* d'un terme est son image par ϕ. On note[7] $\mathcal{T}_n$ et $\mathcal{V}_n$ les ensembles des termes et des variables de degré au plus n et $\mathcal{T}^n$ et $\mathcal{V}^n$ les ensembles des termes et des variables de degré exactement n. Une substitution μ est *ϕ-admissible* si toute variable est de degré plus élevé que son substitué, c'est-à-dire $\phi \circ \mu \leq \phi$. □

Nous nous intéressons désormais à l'ensemble $\mathcal{S}_a$ des substitutions ϕ-admissibles, et à la $\mathcal{S}_a$-A-théorie que nous appellerons ϕ-A-théorie. Le préfixe A- sera en général omis mais le préfixe ϕ- ne sera jamais omis. Par exemple une extension est une A-extension (pour la théorie où toutes les substitutions sont admissibles), et une ϕ-extension est une ϕ-A-extension (pour la ϕ-A-théorie).

Remarque 2.22 Comme la théorie A est régulière, les degrés des membres droits et gauches de chaque axiome sont toujours égaux. Deux termes ϕ-égaux seront donc toujours de même degré.

Nous pouvons appliquer l'ensemble des résultats précédents. Nous noterons $\mathcal{U}_A^\phi(U)$ l'ensemble des ϕ-solutions d'un unificande U. Un unificande peut également être considérer comme un ϕ-unificande. On s'intéressera donc aussi à la comparaison de la A-théorie avec la ϕ-A-théorie

Une présentation avec sortes ordonnées

La ϕ-A-théorie est en fait une théorie ordo-sortée. L'unification dans les théories ordo-sortées a été beaucoup étudiée ces dernières années. Une classification des différentes théories ordo-sortées et des résultats généraux sur ces théories se trouvent dans [53]. C. Kirchner a également incorporé l'unification ordo-sortée dans le formalisme de [36].

Pour simplifier supposons que $\mathcal{C}$ est simplement gradué, et que A est la théorie vide. Considérons l'ensemble des sortes $\mathcal{K}$ égal à $\mathbb{N}$ ordonné par $\leq$, avec les assertions de sorte $\alpha : \phi(\alpha)$ et pour chaque symbole f de $\mathcal{C}$ et toute sorte n

$$f : \underbrace{n \otimes \ldots n}_{\varrho(f)} \Rightarrow n.$$

Pour tout terme σ la plus petite sorte de σ est égale au sup de l'ensemble des sortes de ses variables. L'ensemble des Σ-substitutions est donc égal à l'ensemble des substitutions ϕ-admissibles. Ainsi la

[6] Cette condition garantit qu'il y a assez de variables de chaque sorte dans chaque image réciproque.
[7] Nous essayerons toujours que les exposants correspondent à une projection et les indices à une réunion.

théorie ordo-sortée ci-dessus est égale à la ϕ-théorie. C'est encore vrai pour une théorie initiale non vide et un ensemble de sortes $\mathcal{K}$ quelconque.

On pourrait donc également étudier la ϕ-théorie dans le formalisme des théories ordo-sortées. Mais nous n'aurions pas vraiment besoin de toute la puissance de ce formalisme. D'autre part, le calcul incrémental sur les sortes dont nous aurions besoin pour obtenir un algorithme efficace nécessiterait d'étendre le formalisme de [38]. La présentation par restriction des substitutions admissibles est plus simple.

2.3.2 Transformations hiérarchisées

Proposition 2.20 *Deux unificandes sont ϕ-équivalents s'ils sont équivalents.*

Démonstration Si deux unificandes sont équivalents, alors ils ont même ensemble de solutions, ils ont a fortiori même ensemble de ϕ-solutions. Réciproquement deux unificandes ϕ-équivalents ont même ensemble de ϕ-solutions. Ils ont donc même ensemble de ϕ-solutions à valeurs dans $\mathcal{T}^0$. Une solution à valeur dans $\mathcal{T}^0$ est toujours une ϕ-solution. Donc les deux unificandes ont même ensemble de solutions à valeurs dans $\mathcal{T}^0$. Or $\mathcal{V}^0$ est une copie[8] de $\mathcal{V}$. Comme l'ensemble des solutions est fermé par renommage, les deux unificandes ont donc même ensemble de solutions à valeurs dans $\mathcal{T}$. ■

Par contre une extension n'est pas nécessairement une ϕ-extension. Un exemple est la ϕ-généralisation, qui n'est possible que par une substitution ϕ-admissible. En général, le lemme suivant sera suffisant.

Proposition 2.21 *Soit U un unificande de degré n et W un ensemble de variables contenant $\mathcal{V}_n$. Si un unificande étend U en dehors de W, alors il ϕ-étend U en dehors de W.*

Démonstration Supposons que U' étend U en dehors de W. La restriction à U d'un unificateur admissible de U' est un unificateur de $\mathcal{M}$ et est ϕ-admissible.

Inversement, soit μ un ϕ-unificateur de U. Il peut être étendu en dehors de W en un unificateur de U', par une substitution ν de domaine pris en dehors de $\mathcal{V}_n$. L'ensemble $I(\nu) \setminus \mathcal{V}_n$ peut-être copié dans $\mathcal{T}_n \setminus (I(\mu) \cup I(\nu))$ par un isomorphisme dont on note θ la restriction à $I(\nu)$. La somme $\mu + \nu\theta$ est un unificateur de U'. Elle est admissible car μ l'est par hypothèse et le domaine de $\nu\theta$ est en dehors de $\mathcal{V}_n$ et son image est dans $\mathcal{T}_n$. C'est donc un ϕ-unificateur de U' étendant μ en dehors de W. ■

Corollaire 2.22 *Les processus de ϕ-décomposition et ϕ-fusion sont égaux aux processus de décomposition et fusion. Une théorie mutable est ϕ-mutable.*

2.3.3 Résolution d'un système complètement décomposé

Définition 2.24 Pour tout système $\mathcal{M}$ complètement décomposé on définit la fonction hiérarchique de $\mathcal{M}$, notée $\phi_{\mathcal{M}}$ comme la plus grande hiérarchie majorée par ϕ et constante sur chaque multi-équation de $\mathcal{M}$. □

Nous montrerons dans la partie suivante que cette hiérarchie existe toujours et est unique. L'introduction de la fonction hiérarchique est motivée par le lemme suivant.

Lemme 2.23 *Soit $\mathcal{M}$ un système complètement décomposé protégé par W admettant une pré-solution $[\mathcal{M}]$. Si θ est un renommage des variables $\mathcal{V}(\mathcal{M})$ en dehors de W tel que $\phi \circ \theta$ soit égale à $\phi_{\mathcal{M}}$ sur le domaine de θ, alors $[\mathcal{M}]\theta$ est une ϕ-solution principale de $\mathcal{M}$ en dehors de W.*

Démonstration Remarquons que $[\mathcal{M}]$ associe à toute variable de $\mathcal{M}$ un terme de la même multi-équation donc ayant même image par $\phi_{\mathcal{M}}$. Ainsi, en composant l'égalité $\phi \circ \theta = \phi_{\mathcal{M}}$ par $[\mathcal{M}]$, on obtient

$$\phi \circ [\mathcal{M}]\theta = \phi_{\mathcal{M}}.$$

Il est clair que $[\mathcal{M}]\theta$ est une solution, c'est même une solution principale. Elle est ϕ-admissible car pour toute variable α de son domaine, c'est-à-dire de $\mathcal{M}$, le degré $\phi(\alpha[\mathcal{M}]\theta)$ de $\alpha[\mathcal{M}]\theta$ est égal à $\phi_{\mathcal{M}}(\alpha)$ qui est lui-même plus petit que $\phi(\alpha)$.

[8] Bien sûr, ce n'est pas une ϕ-copie, car une ϕ-copie "conserve" les degrés.

Réciproquement une ϕ-solution de $\mathcal{M}$ est aussi solution de $\mathcal{M}$, donc plus petite par une substitution ξ que $[\mathcal{M}]\theta$ que l'on peut toujours supposer de domaine inclus dans $I([\mathcal{M}]\theta)$. Montrons que ξ est ϕ-admissible. La fonction $\phi \circ ([\mathcal{M}]\theta\xi)$ est constante sur chaque multi-équation, et plus petite que ϕ car $[\mathcal{M}]\theta\xi$ est admissible. Elle est donc aussi plus petite que $\phi_{\mathcal{M}}$, c'est-à-dire $\phi \circ ([\mathcal{M}]\theta)$. Toute variable α de $D(\xi)$ étant dans le domaine de $[\mathcal{M}]\theta$, on en déduit sans peine l'inégalité $\phi \circ \xi \leq \phi$, ce qui prouve que ξ est ϕ-admissible. ■

Théorème 3 *Si la théorie A possède un algorithme d'unification standard, alors elle possède un algorithme de ϕ-unification standard.*

Démonstration Dire qu'une théorie A possède un algorithme standard revient à dire qu'elle est stricte, mutable et que le processus de décomposition-fusion-mutation termine. Elle est donc ϕ-mutable, par le même processus de mutation mais restreint à n'introduire que des variables de degrés suffisamment élevés. Si un système n'admet pas de solution, alors il n'admet pas de ϕ-solution. Sinon, il peut être ϕ-étendu par un système complété qui admet au moins une pré-solution. On peut alors calculer sa fonction hiérarchique qui permet d'obtenir une ϕ-solution principale. ■

Définition 2.25 Un système de multi-équations complètement décomposé $\mathcal{M} \sqcap \mathcal{N}$ est une *conjonction d'équilibre* au degré n si $\mathcal{N}$ contient exactement les multi-équations d'images supérieures ou égales à n par la fonction hiérarchique $\phi_{\mathcal{M}\sqcap\mathcal{N}}$. □

Proposition 2.24 *Soit $\mathcal{M} \sqcap \mathcal{N}$ une conjonction d'équilibre au degré n telle que $\mathcal{N}$ admette une pré-solution $[\mathcal{N}]$. S'il existe une solution du système $\mathcal{M} \sqcap \mathcal{N}$ alors il existe un unificateur principal μ de $\mathcal{M}$ tel que $[\mathcal{N}]\mu$ soit ϕ-plus petit qu'un unificateur principal de $\mathcal{M} \sqcap \mathcal{N}$.*

Remarque 2.23 Si $\mathcal{M} \sqcap \mathcal{N}$ est une conjonction d'équilibre au degré n, nécessairement $\mathcal{N} \not\preceq \mathcal{M}$. En effet, une variable de $V(\mathcal{N})$ a une image par $\phi_{\mathcal{M}\sqcap\mathcal{N}}$ au moins égale à n. Elle ne peut donc pas apparaître dans un terme de $T(\mathcal{M})$, ce qui rendrait son image par $\phi_{\mathcal{M}\sqcap\mathcal{N}}$ au moins égale à n. Le calcul de la fonction hiérarchique dans la partie suivante montrera qu'alors la restriction de la fonction hiérarchique $\phi_{\mathcal{M}\sqcap\mathcal{N}}$ à $\mathcal{M}$ est égale à la fonction hiérarchique $\phi_{\mathcal{M}}$.

Démonstration S'il existe une solution au système, l'ordre $\preceq_{\mathcal{M}\sqcap\mathcal{N}}$ est strict, et il existe alors une pré-solution $[\mathcal{N}]$ de $\mathcal{N}$, et une pré-solution $[\mathcal{M}]$ de $\mathcal{M}$ qui soit ϕ-plus petite par η qu'un unificateur principal de $\mathcal{M}$. Comme $\mathcal{N}$ n'est pas liée à $\mathcal{N}$, la composition $[\mathcal{N}][\mathcal{M}]$ est une pré-solution de $\mathcal{M} \sqcap \mathcal{N}$. Elle doit être composée avec un renommage pour devenir un ϕ-unificateur principal de $\mathcal{M} \sqcap \mathcal{N}$. Celui-ci est défini par $\phi_{\mathcal{M}\sqcap\mathcal{N}}$, et la remarque précédente assure que η convient donc pour les variables de $\mathcal{M}$. Il peut être étendu par un renommage θ pour les variables restantes. On obtient ainsi un unificateur $[\mathcal{N}][\mathcal{M}]\eta\theta$ de $\mathcal{M} \sqcap \mathcal{N}$. ■

2.3.4 Calcul de la fonction hiérarchique

Soit $\mathcal{M}$ un système complètement décomposé. Nous supposons que pour chaque variable α de $\mathcal{M}$ il existe une multi-équation e de $\mathcal{M}$ telle que α soit dans $V(\mathcal{M})$. Nous pouvons toujours nous ramener à un tel système par ϕ-équivalence et en conservant la complète décomposition, simplement en ajoutant à $\mathcal{M}$ des multi-équations réduites à une seule variable.

Dans cette partie, nous montrons comment calculer la fonction hiérarchique de $\mathcal{M}$. Le système $\mathcal{M}$ restera fixé. Nous cherchons la plus grande hiérarchie majorée par ϕ et constante sur chaque multi-équation. Pour simplifier, nous nous ramenons à des fonctions de $\mathcal{M}$ vers $\mathbb{N}$ que nous appellerons *fonctions d'états*. Nous désignerons les fonctions d'états par φ, ψ et χ. Nous appellerons *état d'une multi-équation* e par φ la valeur $\varphi(e)$ et *état d'un système de multi-équations* $\mathcal{N}$ par φ le sup de φ sur $\mathcal{N}$. En général φ est précisé par le contexte et est sous-entendu.

La constance de $\phi_{\mathcal{M}}$ sur chaque multi-équation s'écrit

$$\forall e \in \mathcal{M},\ \exists n \in \mathbb{N},\ \forall \sigma \in e,\ n = \phi_{\mathcal{M}}(\sigma).$$

En considérant e comme la réunion de $V(e)$ et de $T(e)$, la première projection nous assure l'existence d'un état $\varphi_{\mathcal{M}}$ tel que

$$\forall e \in \mathcal{M},\ \forall \alpha \in V(e),\ \varphi_{\mathcal{M}}(e) = \phi_{\mathcal{M}}(\alpha).$$

La projection sur $T(e)$ s'écrit aussi

$$\forall e \in \mathcal{M},\ \forall \sigma \in T(e),\ \varphi_{\mathcal{M}}(e) = \sup_{\beta \in \mathcal{V}(\sigma)} \phi_{\mathcal{M}}(\beta).$$

Or pour toute variable β de $\mathcal{M}$, il existe une multi-équation e' telle que β soit dans $V(e')$. Comme le système est complètement décomposé, la multi-équation est nécessairement unique, et nous la noterons $e(\beta)$. Ce qui permet d'écrire $\varphi_{\mathcal{M}}(e(\beta))$ au lieu de $\phi_{\mathcal{M}}(\beta)$. En regroupant les variables de $\mathcal{V}(\sigma)$ qui sont dans une même multi-équation, qui est nécessairement une multi-équation liée[9] à e La condition précédente s'écrit

$$\forall e \in \mathcal{M},\ \forall \sigma \in T(e),\ \varphi_{\mathcal{M}}(e) = \sup_{\substack{e' \in \mathcal{M} \\ e' \preceq e}} \sup_{\substack{\beta \in \mathcal{V}(\sigma) \\ \beta \in V(e')}} \Big(\varphi_{\mathcal{M}}(e(\beta))\Big),$$

qui se simplifie finalement en

$$\forall e \in \mathcal{M},\ \big(T(e) \neq \emptyset \Longrightarrow \varphi_{\mathcal{M}}(e) = \sup_{e' \preceq e} \varphi_{\mathcal{M}}(e')\big). \tag{1}$$

Il est clair que $\varphi_{\mathcal{M}}$ est inférieur à la fonction d'état φ_0 qui associe l'entier $\inf(\phi(V(e)))$ à chaque multi-équation e de $\mathcal{M}$. Nous sommes donc ramenés à la recherche de la plus grande fonction d'état majorée par φ_0 et vérifiant la condition (1) ci-dessus.

Nous définissons les deux transformations suivantes sur les fonctions d'états.

- Soit e et e' deux multi-équations telles que e' soit liée à e. Si l'état de e' par ψ est une valeur k supérieure strictement à l'état de e, on appelle *propagation* de e dans e' le remplacement de ψ par la fonction d'état égale à ψ sauf pour la multi-équation e' où elle prend la valeur k.
- Soit e une multi-équation contenant un terme non variable, et $\mathcal{N}$ l'ensemble des multi-équations liées à e. Si l'état de $\mathcal{N}$ par ψ est une valeur k inférieure strictement à l'état de e, on appelle *réalisation* de e, le remplacement de ψ par la fonction d'état égale à ψ sauf pour la multi-équation e où elle prend la valeur k.

En résumé, si e est une multi-équation et $\mathcal{N}$ le système des multi-équations qui lui sont liées, la première transformation propage son état dans toutes les multi-équations de $\mathcal{N}$, et la seconde réalise son état à celui de $\mathcal{N}$.

De façon évidente, si ψ majore $\varphi_{\mathcal{M}}$, alors une propagation ou une réalisation de ψ majore $\varphi_{\mathcal{M}}$. Elle est plus petite que ψ. Le processus de propagation-réalisation termine donc avec une fonction d'état φ_f. La propagation n'est plus possible, ce qui assure la propriété (1) donc que $\varphi_{\mathcal{M}}$ est atteinte.

Un contrôle efficace

Le calcul de la fonction hiérarchique peut se faire en un nombre d'étapes linéaire par rapport à la taille[10] de l'ordre $\preceq$. Pour un système ne comportant que des termes de taille 1, la taille de $\preceq_{\mathcal{M}}$ est majorée par la taille du système. Nous nous limiterons donc à une seule propagation et une seule réalisation par relation de liaison. Pour simplifier, nous effectuerons d'abord toutes les propagations, puis toutes les réalisations. Avant tout, il faut s'assurer que cette séparation est correcte, c'est-à-dire que la réalisation d'un système propagé est un système propagé. La vérification est immédiate.

Nous appelons propagation de degré n la propagation d'une multi-équation dont l'état est égal à n. Nous ne pouvons propager une telle multi-équation qu'un nombre de fois égal au mombre de multi-équations qui lui sont liées. Pour respecter la limite que nous nous sommes fixée, il suffit de garantir que chaque multi-équation ne sera propagée qu'à un seul degré. Cela est clairement obtenu en effectuant d'abord les propagations des degrés les plus élevés. Si de plus on les effectue de l'extérieur vers l'intérieur, on sera assuré de ne pas examiner plusieurs fois une même paire de multi-équations.

Nous appelons réalisation de degré n la réalisation d'une multi-équation dont l'état est égal à n. A l'inverse de la propagation, la réalisation doit se faire de l'intérieur vers l'extérieur. Ainsi, la

[9] Nous rappelons que e' *est lié à* e se note $e' \preceq e$.

[10] C'est-à-dire le cardinal de l'ensemble $\{(e', e) \in \mathcal{M} \times \mathcal{M} \mid e' \preceq e\}$.

propagation d'une multi-équation e suppose que toutes les multi-équations transitivement liées à e sont propagées. Comme elles sont aussi réalisées, leur état ne sera pas ultérieurement modifié par une réalisation et la réalisation de e est définitive. A nouveau, on aura examiné une seule fois chaque couple de multi-équations liées.

Incrémentatilité de l'équilibrage

Dans les applications ultérieures, nous demanderons de connaître l'image réciproque par $\phi_{\mathcal{M}}$ d'un entier n sans pour autant demander le calcul complet de $\phi_{\mathcal{M}}$. En fait nous chercherons seulement des conjonctions d'équilibre au degré n. Le système $\mathcal{M}$ sera ensuite mélangé à un autre système $\mathcal{M}'$. La question qui se pose est double. Peut-on ne calculer qu'une partie de $\phi_{\mathcal{M}}$? Peut-on réutiliser ce calcul pour obtenir une information ultérieure sur $\phi_{\mathcal{M}\sqcap\mathcal{M}'}$?

La réponse à la première question est simple. Pour obtenir une conjonction d'équilibre au degré n, il suffit de connaître une approximation de $\varphi_{\mathcal{M}}$ permettant de décider si une multi-équation est de degré plus grand ou plus petit que n par $\varphi_{\mathcal{M}}$. Nous pouvons simplement calculer une fonction d'état $\varphi^n_{\mathcal{M}}$ telle $(\varphi^n_{\mathcal{M}})^{-1}$ soit égale à $(\varphi_{\mathcal{M}})^{-1}$ pour tout entier plus grand ou égal à n et laisse globalement invariant le segment $[0, n-1]$. Il suffit simplement dans le calcul précédent de se limiter à des propagations et des réalisations de degrés supérieurs ou égaux à n. Cela se vérifie très facilement.

En revanche la réutilisation ultérieure des calculs d'équilibrages est plus complexe, essentiellement parce que la conjonction de deux systèmes complètement décomposés n'est pas en général un système complètement décomposé. Nous sommes donc amenés à étudier le comportement de la propagation et de la réalisation avec les transformations de décomposition et de fusion. Ces transformations créent ou consomment des multi-équations. L'incrémentalité du calcul que nous demandons ne peut s'obtenir qu'en délocalisant le calcul des états dans les unificandes eux-mêmes.

2.3.5 Multi-équations hiérarchisées contraintes

Dans la partie précédente, nous avons fixé un système $\mathcal{M}$ et calculé des fonctions d'états φ de ce système. Nous avons en fait calculé sur des objets $(\mathcal{M}, \varphi)$ mais en laissant $\mathcal{M}$ invariant. Nous voulons maintenant autoriser des transformations de $\mathcal{M}$. Pour cela, il est plus simple de délocaliser la fonction d'état φ dans les multi-équations du système.

Définition 2.26 On appelle *multi-équation contrainte*, un triplet noté $e \downarrow p \uparrow q$, formé d'une multi-équation e et de deux entiers p et q tels que $p \leq q$ appelés respectivement état de propagation et état de réalisation. On note également $\downarrow(e)$ l'entier p et $\uparrow(e)$ l'entier q. □

On peut projeter toute multi-équation contrainte sur la multi-équation simple obtenue en oubliant les contraintes. Cette projection permet de définir les opérations

1. $var\,(e \downarrow p \uparrow q) = var\,(e)$,
2. $sub\,(e \downarrow p \uparrow q, \mu) = sub\,(e, \mu) \downarrow p \uparrow q$,
3. $val\,(e \downarrow p \uparrow q) \iff val\,(e)$.

qui munissent donc l'ensemble des multi-équations contraintes de la même structure d'unificande que l'ensemble des multi-équations.

Définition 2.27 On appelle *système contraint*, un système de multi-équations contraintes qui vérifie les conditions suivantes

1. $\forall \alpha \in \mathcal{V}(\mathcal{M}),\ \exists e \in \mathcal{M},\ \alpha \in V(e)$
2. $\forall \mu \in \mathcal{U}_A(\mathcal{M}),\ \forall e \in \mathcal{M},\ \forall \alpha \in V(e),\ \phi(\alpha\mu) \leq \downarrow(e) \leq \phi(\alpha)$
3. $\forall e, e' \in \mathcal{M},\ e' \preceq_{\mathcal{M}} e \Longrightarrow \downarrow(e') \leq \uparrow(e)$

□

La première condition est technique et évite de devoir traiter séparément des cas pathologiques. La seconde condition assure que $\downarrow(e)$ sera une majoration pas trop grande de $\phi_{\mathcal{M}}$. La troisième condition est essentielle pour l'incrémentalité. Elle certifie que les valeurs des multi-équations qui lui sont transitivement liées (et donc peut-être très éloignées) auront un état d'équilibre plus faible que son état actuel. Nous précisons comment les transformations de ϕ-extensions s'étendent aux systèmes contraints. Ces transformations resterons des ϕ-extensions car les projections des transformations sont identiques aux transformations des projections.

Définition 2.28 Soit f un symbole décomposable, σ et τ des termes ayant le symbole f en tête, et e une multi-équation. On appelle *décomposition contrainte* de la multi-équation $\sigma \doteq \tau \doteq e' \downarrow p \uparrow q$, son remplacement par le système de multi-équations

$$(\sigma \doteq e' \downarrow p \uparrow q) \sqcup \bigsqcup_{i \in [1,p]} (\sigma_{/i} \doteq \tau_{/i} \downarrow p \uparrow q).$$

□

Proposition 2.25 *Si $\mathcal{M}$ est un système contraint, la décomposition d'une multi-équation de $\mathcal{M}$ est un système contraint.*

Définition 2.29 Soit deux multi-équations de la forme $\alpha \doteq e \downarrow p \uparrow q$ et $\alpha \doteq e' \downarrow p' \uparrow q'$. On appelle *fusion contrainte* des deux multi-équations précédentes la multi-équation

$$\alpha \doteq e \doteq e' \downarrow (p \inf p') \uparrow (q \sup q')$$

□

Proposition 2.26 *Si $\mathcal{M}$ est un système contraint, alors la fusion de deux multi-équations de $\mathcal{M}$ est un système contraint.*

Définition 2.30 On appellera mutation d'un système contraint toute mutation du système qui est un système contraint. □

Il est immédiat qu'un système contraint est mutable si et seulement si la projection du système est mutable.

Définition 2.31 Soit $\mathcal{M}$ un système contraint $e \downarrow p \uparrow q$ et $e' \downarrow p' \uparrow q'$ deux multi-équations de $\mathcal{M}$ telles que e' soit liée à e. Si $p < p'$, on appelle *propagation contrainte* de e dans e' le remplacement dans $\mathcal{M}$ de $e' \downarrow p' \uparrow q'$ par $e' \downarrow p \uparrow q'$. □

Définition 2.32 Soit $\mathcal{M}$ un système contraint $e \downarrow p \uparrow q$ une multi-équation de $\mathcal{M}$ et Q l'entier

$$\inf\{\uparrow(e') \mid e' \in \mathcal{M} \wedge e' \preceq e\}$$

Si $Q < q$ on appelle *réalisation contrainte* de e, le remplacement dans $\mathcal{M}$ de $e \downarrow p \uparrow q$ par $e \downarrow (p \inf Q) \uparrow Q$. □

Proposition 2.27 *Si $\mathcal{M}$ est un système contraint, alors toute propagation contrainte et toute réalisation contrainte de $\mathcal{M}$ sont des systèmes contraints.*

Les propriétés ci-dessus montrent que le calcul de l'état d'équilibre du système peut se mélanger avec le calcul de sa forme complètement décomposée. Il n'y a pas de surcoût dû à l'incrémentalité. Nous avons simplement demandé aux opérations de ϕ-extension de tenir à jour l'information sur l'état du système. Ces résultats seront utilisés dans les algorithmes de synthèse type, donnés en annexe.

2.4 Théories syntaxiques

Dans cette partie, nous considérons un ensemble de sortes $\mathcal{K}$, un ensemble de symboles $\mathcal{C}$, une signature Σ de $\mathcal{C}$ sur $\mathcal{K}$ et la Σ-algèbre $\mathcal{T}(\mathcal{C},\mathcal{V})$ libre sur un $\mathcal{K}$-ensemble de variables $\mathcal{V}$, munie d'une théorie équationnelle A. Nous étudions l'opération de mutation pour les unificandes canoniques.

Nous reprenons le formalisme des théories syntaxiques définies par C. Kirchner [36], basé sur la notion d'ensemble d'axiomes résolvants. L'essentiel de cette partie est l'introduction d'un formalisme permettant de simplifier l'étude des théories syntaxiques, et d'obtenir des conditions suffisantes pour qu'une présentation A d'une théorie soit syntaxique plus générales que celles de [36].

Les travaux de [36], notamment sur la complétion d'un ensemble d'axiomes pour aboutir à une présentation résolvante ont été approfondis dans [39, 20]. Tobias Nipkow a également montré que certaines transformations de preuves par réécriture dans les théories équationnelles étaient liées à des théories résolvantes [47] .

Hypothèses Nous supposons dans cette partie que la théorie équationnelle est non *potente*, c'est-à-dire que la racine des termes des axiomes n'est jamais une variable.

2.4.1 Chemins

Rappelons qu'un axiome d'une théorie A est un couple de termes. Nous désignerons les axiomes par les lettres r et s. Nous noterons r_g sa première projection et r_d sa seconde projection. Si $\sigma \doteq \tau$ est un axiome, $\tau \doteq \sigma$ n'est pas nécessairement un axiome, mais son ajout dans A ne modifie pas la théorie A. A chaque couple (σ,τ) on peut associer le couple symétrique (τ,σ). Pour éviter de distinguer le sens d'utilisation d'un axiome dans une preuve, nous supposerons que la présentation A est fermée par symétrie. Nous noterons parfois $r\uparrow$ et $r\downarrow$ deux axiomes symétriques.

La notion d'occurrence permet de désigner un sous-terme dans un terme, mais elle donne très peu de renseignements sur la structure du terme. De plus, étant donné un alphabet gradué $\mathcal{C}$, la plupart des occurrences n'ont aucun sens, car elles ne sont définies pour aucun terme de l'algèbre $\mathcal{T}(\mathcal{C},\mathcal{V})$ libre sur $\mathcal{V}$. Une notion mieux adaptée est celle de chemin.

Soit $\mathcal{C}$ un ensemble de symboles gradués[11]. Chaque symbole f de $\mathcal{C}$ possède une arité $\varrho(f)$. Une *direction* est un couple (f,k), souvent noté $\underline{f}_k$ formé d'un symbole f d'arité non nulle et d'un entier k de $[1,\varrho(f)]$. Un *chemin* est une suite finie de directions.

Soit u un chemin de la forme $(f_i,x_i)_{i\in[1,p]}$, on peut lui associer l'occurrence $(x_i)_{i\in[1,p]}$. Soit σ un terme. On dira que le chemin précédent est un chemin dans σ si

- l'occurrence $(x_i)_{i\in[1,p]}$ est une occurrence de σ,
- pour tout k de $[2,p]$, à l'occurrence $(x_i)_{i\in[1,k-1]}$ se trouve le symbole f_k.

On note alors $\sigma_{/u}$ le sous-terme de σ à l'occurrence associée à u.

Exemple 2.24 La suite $\underline{f}_1\underline{g}_2$ est un chemin. L'occurrence associée est la suite 12 de longueur 2.

Deux chemins sont dits disjoints si aucun n'est préfixe de l'autre.

Si les symboles sont simplement signés, chaque chemin est au moins chemin d'un terme de l'algèbre. Mais si les symboles sont munis de sortes, la notion de chemin n'est pas suffisamment précise. La bonne notion serait celle de terme mono-frontière linéaire introduite dans la partie 4.5.3 du chapitre 4. Bien que nous considérerons toujours des termes avec sortes, nous nous contenterons du concept de chemin.

Les chemins sont beaucoup plus précis que les occurrences. Parfois même trop précis. En fait il est agréable d'avoir une notion intermédiaire entre celle de chemin et celle d'occurrence. En remarquant qu'une occurrence u peut toujours être considérée comme l'ensemble de tous les chemins ayant u pour occurrence associée, il est naturel de considérer des ensembles de chemins. En fait nous nous intéresserons souvent à des ensembles de chemins très simples que nous désignerons par le langage suivant inspiré de celui des expressions régulières :

- Nous notons ϵ le singleton réduit au chemin vide.

[11] Un ensemble de symboles avec sortes convient très bien car il est a fortiori gradué.

- Pour tout symbole f d'arité p non nulle, nous notons $\underline{f}$ l'ensemble de chemins $\{\underline{f}_x \mid x \in [1, \varrho(f)]\}$.
- Pour tout entier x, nous notons $\underline{x}$ l'ensemble de chemins $\{\underline{f}_x \mid f \in \mathcal{C} \wedge \varrho(f) \geq x\}$. Pour des valeurs élevées de x, cet ensemble peut être vide.
- Nous identifions un chemin au singleton formé de ce seul chemin.
- Nous notons $\mathbf{.}$ le chemin de longueur 1 formé de l'ensemble de toutes les directions.
- Si u et v sont des ensembles de chemins, nous notons uv la concaténation de ces ensembles formée de toutes les concaténations d'un chemin de u avec un chemin de v.
- Si u et v sont des ensembles de chemins, nous notons $(u \mid v)$ la réunion des ensembles u et v formée de tous les chemins de u et de v.

  Nous notons u^δ la réunion $(\epsilon \mid u)$.
- Si K est un ensemble totalement ordonné, et $(u_k)_{k\in K}$ une suite d'ensembles de chemins dépendant de k, on note $(u_k)^{k\in K}$ la concaténation de tous les ensembles de chemins dans l'ordre des k croissants.

  Lorsque u ne dépend pas de k et si K est fini, on retrouve la notation classique de l'exponentiation des expressions régulières en notant simplement u^K.

  On note u^* pour $(u^\delta)^N$.

Exemple 2.25 L'expression $(\mathbf{..}^*)$ désigne l'ensemble des chemins de longueur non nulle.

2.4.2 Relations de prouvabilité

Soit σ un terme et u un chemin dans σ. S'il existe un axiome r et une substitution μ telle que $\sigma_{/u}$ soit égal à $r_d\mu$, alors le terme τ égal à $\sigma[r_g\mu/u]$ est A-égal à σ. On dit que τ est obtenu à partir de σ par une application de l'axiome r au chemin u. On dit aussi que l'égalité $\sigma =_A \tau$ est prouvable par application de l'axiome r au chemin u et on note

$$\sigma \xrightarrow[r,(u)]{} \tau.$$

La prouvabilité est une relation. La réunion des relations

$$\xrightarrow[r,(u)]{} \cup \xrightarrow[s,(v)]{}$$

est la relation de prouvabilité de l' A-égalité de deux termes par application de l'axiome r au chemin u ou par l'axiome s à le chemin v. Lorsque r et s sont identiques, on peut écrire

$$\xrightarrow[r,(u|v)]{}$$

qui est la relation de prouvabilité d'une A-égalité par application de l'axiome r à un chemin de l'ensemble $(u \mid v)$. On dira qu'une relation $\xrightarrow[r,(u)]{}$ est élémentaire si u est un singleton.

Si R est un sous-ensemble de A, on note $\xrightarrow[R,(u)]{}$ la relation de prouvabilité par l'application d'un axiome de R à l'occurrence u égale à

$$\bigcup_{r\in R} \left(\xrightarrow[r,(u)]{} \right)$$

et simplement $\xrightarrow[(u)]{}$ lorsque R est égal à A. Par défaut R est égal à A et (u) est l'ensemble de tous les chemins. On retrouve ainsi la notation habituelle $\longrightarrow$ pour une relation de prouvabilité élémentaire quelconque, c'est-à-dire la relation $=_A$. La relation d'égalité sur $\mathcal{T}$ que l'on notera Δ pour éviter toute ambiguité avec la méta-notation est une relation de prouvabilité. On note $\xrightarrow[x]{\delta}$ la réunion $\xrightarrow[x]{} \cup\, \Delta$.

La composition d'une relation de prouvabilité $\underset{x}{\longrightarrow}$ avec une relation de prouvabilité $\underset{y}{\longrightarrow}$ égale à $\underset{y}{\longrightarrow} \circ \underset{x}{\longrightarrow}$ s'écrit simplement $\underset{x}{\longrightarrow}\underset{y}{\longrightarrow}$. Si K est un ensemble fini totalement ordonné, et $\left(\underset{x_k}{\longrightarrow}\right)_{k\in K}$ une suite de relations de prouvabilité dépendant de k, on note

$$\left(\underset{x}{\longrightarrow}\right)^{k\in K}$$

ou plus simplement $\underset{x}{\overset{k\in K}{\longrightarrow}}$ la composition définie récursivement par

$$\wedge \begin{cases} \underset{x}{\overset{0}{\longrightarrow}} = \Delta, \\ \underset{x_k}{\overset{k\in[1,p+1]}{\longrightarrow}} = \underset{x_k}{\overset{k\in[1,p]}{\longrightarrow}}\underset{x_{k+1}}{\longrightarrow}. \end{cases}$$

où $p+1$ désigne le plus petit élément supérieur strictement à p dans K, et $[1,p]$ l'ensemble de tous les éléments inférieur ou égal à p dans K. Lorsque x ne dépend pas de k, on retrouve l'itération de l'opération de composition que l'on note simplement $\underset{x}{\overset{K}{\longrightarrow}}$.

On note $\underset{x}{\overset{*}{\longrightarrow}}$ la fermeture transitive de la relation $\underset{x}{\longrightarrow}$

Exemple 2.26 La notation

$$\underset{(\bullet^k\bullet^*)}{\overset{*}{\longrightarrow}}\underset{(\epsilon)}{\overset{\delta}{\longrightarrow}}$$

est la relation de prouvabilité par un nombre quelconque d'applications d'axiomes à un chemin de longueur au moins k suivi éventuellement de l'application d'un axiome à la racine.

La notation

$$\underset{(\epsilon)}{\longrightarrow}\underset{(\bullet\bullet^*)}{\overset{*}{\longrightarrow}}\underset{(2\bullet^*)}{\longrightarrow}$$

désigne la relation de prouvabilité par l'application d'un axiome à la racine suivi d'un nombre quelconque d'applications d'axiomes à des occurrences non nulles puis l'application d'un axiome dans le deuxième sous terme direct.

La relation d'A-égalité se note simplement

$$\underset{(\bullet^*)}{\overset{*}{\longrightarrow}}$$

Pour tous termes σ et τ A-égaux, il existe donc une composition de relations de prouvabilité élémentaires

$$\underset{x_i}{\overset{i\in[1,p]}{\longrightarrow}},$$

et une suite de termes intermédiaires $(\sigma_i)_{i_{[\in[1,1],p-1]}}$ telles que

$$\sigma \underset{x_1}{\longrightarrow} \sigma_1 \ldots \underset{x_i}{\longrightarrow} \sigma_i \ldots \sigma_{p-1} \underset{x_p}{\longrightarrow} \tau$$

L'écriture ci-dessus est une *preuve* de la relation $\sigma \underset{x_i}{\overset{i\in[1,p]}{\longrightarrow}} \tau$ entre σ et τ.

Si H est un sous-ensemble de $\mathcal{T}$, et si σ et τ, et tous les termes intermédiaires sont dans H, on dit que la preuve est une *preuve dans* H.

Définition 2.33 Soit H est un ensemble de termes. On dit que relation $\underset{x}{\longrightarrow}$ se *réduit* dans H sur la relation $\underset{y}{\longrightarrow}$, et on note

$$\underset{x}{\longrightarrow} \subset_H \underset{y}{\longrightarrow}$$

si pour tous termes σ et τ tels qu'il existe une preuve de $\sigma \underset{x}{\longrightarrow} \tau$ dans H, il existe aussi une preuve de $\sigma \underset{y}{\longrightarrow} \tau$ dans H. Par défaut H est l'ensemble $\mathcal{T}$ tout entier et on note simplement $\underset{x}{\longrightarrow} \subset \underset{y}{\longrightarrow}$. □

Remarque 2.27 La relation $\subset_H$ n'est pas exactement la restriction de $\subset$ à H. Plus exactement

$$\underset{x}{\longrightarrow} \subset_H \underset{x}{\longrightarrow}$$

n'est en général pas équivalent à

$$(\underset{x}{\longrightarrow} \restriction H^2) \subset (\underset{y}{\longrightarrow} \restriction H^2)$$

car une preuve de $\underset{x}{\longrightarrow} \restriction H$ n'est une preuve dans H de $\underset{x}{\longrightarrow}$ avec certitude que si $\underset{x}{\longrightarrow}$ est une relation de prouvabilité élémentaire.

Remarque 2.28 La relation $\subset$ est la relation d'inclusion sur l'ensemble des relations de prouvabilité. La relation $\subset_H$ est la relation d'inclusion des relations de prouvabilité dans H. Les propriétés suivantes sont donc immédiates.

1. La relation $\subset_H$ est réflexive et transitive.
2. $\left(\underset{x}{\longrightarrow} \subset_H \underset{z}{\longrightarrow} \wedge \underset{v}{\longrightarrow} \subset_H \underset{z}{\longrightarrow}\right) \iff \left((\underset{x}{\longrightarrow} \cup \underset{v}{\longrightarrow}) \subset_H \underset{z}{\longrightarrow}\right)$

Proposition 2.28 *Nous avons également de façon évidente les réductions suivantes :*

$$\left(\left(\underset{r_i,(u_i)}{\xrightarrow{i \in [0,p]}}\right) \subset_H \left(\underset{s_j,(v_j)}{\xrightarrow{j \in [0,q]}}\right)\right) \Longrightarrow \left(\left(\underset{r_{p-i},(u_{p-i})}{\xrightarrow{i \in [0,p]}}\right) \subset_H \left(\underset{s_{q-j},(v_{q-j})}{\xrightarrow{j \in [0,q]}}\right)\right)$$

et pour tous chemins u et v disjoints,

$$\underset{r,(u)}{\longrightarrow}\underset{s,(v)}{\longrightarrow} \subset_H \underset{s,(v)}{\longrightarrow}\underset{r,(u)}{\longrightarrow}$$

Remarque 2.29 Si u est un chemin, d'une preuve $\sigma \underset{r,(u)}{\longrightarrow} \tau$, on obtient une preuve $\sigma_{/u} \underset{r,(\epsilon)}{\longrightarrow} \tau_{/v}$ dite la sous-preuve à l'occurrence u. Réciproquement, si $\sigma_{/u} \underset{r,(\epsilon)}{\longrightarrow} \tau$, alors $\sigma \underset{r,(u)}{\longrightarrow} \sigma[\tau/u]$.

2.4.3 Présentations résolvantes

Pour tous symboles homogènes f et g, on note $A(f,g)$ l'ensemble $A \cap f(\mathcal{T}) \times g(\mathcal{T})$.

Définition 2.34 Un couple de symboles homogènes (f,g) est *résolvant* pour la présentation A si pour tout couple de termes A-égaux de $f(\mathcal{T}) \times g(\mathcal{T})$, il existe une preuve de leur A-égalité qui contienne au plus une occurrence d'un axiome à la racine. Une présentation A est *résolvante* si tous les couples de symboles homogènes sont résolvants. Une théorie est *résolvante* si'l existe une présentation résolvante de cette théorie. □

Exemple 2.30 La présentation $\{a \doteq b, b \doteq c\}$ n'est pas résolvante car le couple (a,b) est résolvant mais le couple (a,c) ne l'est pas. Cependant la théorie engendrée est résolvante car l'ajout de l'axiome $a \doteq c$ rend tous les couples résolvants.

La théorie vide est résolvante. La théorie $\{f(g(\alpha)) \doteq g(\alpha)\}$ est résolvante. Mais la théorie $\{f(g(\alpha)) \doteq f(\alpha)\}$ n'est pas résolvante. On pourra trouver de nombreux exemples dans [36] et [39].

Il existe trois sortes de problèmes que l'on peut se poser.

- Une présentation est-elle résolvante ?
- Une théorie est-elle résolvante ?
- Trouver la présentation résolvante d'une théorie résolvante.
- On peut également s'intéresser à minimaliser une présentation résolvante.

Nous nous intéressons dans la suite uniquement au premier problème. Mais la méthode proposée pourrait donner de nouvelles réponses à la troisième question. Une caractérisation des théories résolvantes par la complexité des ensembles complets minimaux de solutions est faite dans [39]. L'étude la plus récente sur la complétion d'une présentation pour la rendre résolvante est certainement [20].

Remarque 2.31 La résolvabilité d'une présentation A s'écrit

$$\xrightarrow{*} \subset \xrightarrow[(\bullet\bullet^*)]{*} \xrightarrow[(\epsilon)]{\delta} \xrightarrow[(\bullet\bullet^*)]{*} .$$

Nous étudierons plus loin la résolvabilité à partir de cette réduction. Mais on pourra aussi l'étudier à partir de la caractérisation suivante.

Lemme 2.29 *Un couple de symboles homogènes f d'arité p et g d'arité q est résolvant si et seulement si pour tous termes σ de $f(\mathcal{T})$ et τ de $g(\mathcal{T})$:*

$$\sigma =_A \tau \iff \vee \left[\begin{array}{l} f = g \wedge (\forall i \in [1,\ p], \sigma_{/i} =_A \tau_{/i}) \\ \exists r \in A(f,g), \exists \mu \in \mathcal{S}, (\forall i \in [1,\ p], \sigma_{/i} =_A r_{g/i}\mu) \wedge (\forall j \in [1,\ q], \tau_{/j} =_A r_{d/j}\mu) \end{array}\right.$$

Démonstration La condition suffisante est évidente. Montrons que la condition est nécessaire. Considérons un couple de symboles (f,g) et deux termes σ de $f(\mathcal{T})$ et τ de $g(\mathcal{T})$ A-égaux.

S'il existe une preuve de leur A-égalité sans application d'axiome à la racine, alors il est facile de montrer par récurrence sur la taille de cette preuve que f et g sont égaux et que tous les sous-termes directs de σ sont A-égaux aux sous-termes directs correspondant de τ.

S'il existe une preuve de leur A-égalité avec exactement une occurrence d'axiome à la racine, elle est de la forme

$$\sigma \xrightarrow[(\bullet\bullet^*)]{*} \rho \xrightarrow[r,(\epsilon)]{} \pi \xrightarrow[(\bullet\bullet^*)]{*} \tau.$$

Il existe donc une substitution μ telle que $r_g\mu = \rho$ et $r_d\mu = \pi$. Comme la preuve de $\sigma =_A \rho$ ne comporte pas d'occurrence à la racine, on a $\sigma_{/i} =_A r_g\mu_{/i}$ pour tout i de $[1,p]$ où p est l'arité de f. De même, comme la preuve de $\pi =_A \tau$ ne comporte pas d'occurrence à la racine, on a $\tau_{/j} =_A r_g\mu_{/i}$ pour tout j de $[1,q]$ où q est l'arité de g. ■

Proposition 2.30 *Un couple formé d'un seul symbole décomposable est résolvant.*

Démonstration Il suffit de montrer que la proposition

$$\exists r \in A(f,\ f), \exists \mu \in \mathcal{S},\ \forall i \in [1,\ p], (\sigma_{/i} =_A r_{g/i}\mu) \wedge (\tau_{/j} =_A r_{d/j}\mu),$$

implique la proposition

$$\forall i \in [1,p],\ \sigma_{/i} =_A \tau_{/i}.$$

Puisque f est décomposable tout axiome r de $A(f,f)$ satisfait

$$\forall i \in [1,\ p], r_{g/j} =_A r_{d/j}.$$

On obtient directement la conclusion en composant pas μ et en utilisant les égalités en hypothèses. La condition pour que le couple (f,f) soit résolvant se réduit alors à

$$\forall \sigma, \tau \in f(\mathcal{T}),\ (\sigma =_A \tau \iff \forall i \in [1,p],\ \sigma_{/i} =_A \tau_{/i})$$

qui est vérifiée grâce à la remarque précédente. ■

Proposition 2.31 *Un couple de symboles de collision est résolvant.*

Démonstration Si $\{\mathrm{Top}\,(\sigma), \mathrm{Top}\,(\tau)\}$ est un doublet de collision, on n'a jamais $\sigma =_A \tau$ et a fortiori $A(\mathrm{Top}\,(\sigma), \mathrm{Top}\,(\tau))$ est vide. ■

2.4.4 Premières conditions suffisantes de résolvabilité

Montrer qu'un couple est résolvant revient à montrer que les preuves d'A-égalités peuvent s'écrire sous des formes canoniques. On peut espérer obtenir la résolvance par des systèmes de ré-écriture canoniques qui permettent de normaliser les preuves. C'est en partie vrai comme le montre la propriété suivante.

Proposition 2.32 *Soit (f,g) un couple de symboles. S'il existe une orientation canonique $\mathcal{R}$ des axiomes de A telle qu'il existe au plus une règle r de $\mathcal{R}$ ayant l'un des symboles f ou g en tête de l'hypothèse, et aucune règle de $\mathcal{R}$ ayant le cas échéant le symbole* $\mathrm{Top}(r_d)$ *en tête de l'hypothèse, alors le couple (f,g) est résolvant.*

Démonstration Il est clair que dans les conditions précédentes la normalisation par le système $\mathcal{R}$ de chacun des deux termes ne peut contenir qu'une utilisation de règle à une occurrence nulle. Il existe donc une preuve de l'A-égalité de ces deux termes avec au plus une seule occurrence à la racine. ■

Les conditions ci-dessus sont très particulières, et l'utilisation de systèmes de réécriture canoniques pour montrer la résolvance ne convient que pour des cas en général triviaux. Nous développons une méthode assez générale d'étude de la résolvance d'un ensemble d'axiomes. L'idée générale est très simple et consiste à décortiquer la relation $=_A$ en une réunion de relations de prouvabilité

$$\bigcup_{i\in[1,p]} \xrightarrow[x_i]{}$$

pour lesquelles on saura facilement montrer

$$\xrightarrow[x_i]{} \subset \xrightarrow[(..^*)]{*} \xrightarrow[(\epsilon)]{\delta} \xrightarrow[(..^*)]{*} .$$

La résolvabilité de A découlera alors immédiatement de la décomposition ci-dessus. La suite de cette partie consiste à construire des relations de prouvabilité satisfaisant la réduction précédente et à étudier leurs combinaisons.

Lemme 2.33 *Une présentation A est résolvante si et seulement si*

$$\xrightarrow[(\epsilon)]{} \xrightarrow[(..^*)]{*} \xrightarrow[(\epsilon)]{} \subset_H \xrightarrow[(..^*)]{*} \xrightarrow[(\epsilon)]{\delta} \xrightarrow[(..^*)]{*} .$$

pour cette présentation.

Démonstration La condition nécessaire est évidente. Pour montrer la condition suffisante, il suffit de remarquer que la relation $\xrightarrow[(.^*)]{*}$ égale à

$$\bigcup_{k\in N} \left(\xrightarrow[(..^*)]{*} \left(\xrightarrow[(\epsilon)]{} \xrightarrow[(..^*)]{*} \right)^k \right) .$$

On montre alors

$$\xrightarrow[(..^*)]{*} \left(\xrightarrow[(\epsilon)]{} \xrightarrow[(..^*)]{*} \right)^k \subset_H \xrightarrow[(..^*)]{*} \xrightarrow[(\epsilon)]{\delta} \xrightarrow[(..^*)]{*}$$

pour tout k par récurrence sur k. ■

Ce lemme donne une caractérisation plus simple de la résolvabilité. On peut en déduire des conditions suffisantes qui ne seront plus des conditions nécessaires. Naturellement, Plus les conditions suffisantes seront générales, moins elle seront faciles à vérifier.

Proposition 2.34 *Une condition suffisante pour qu'une présentation soit résolvante est*

$$\wedge \begin{cases} \xrightarrow[(\epsilon)]{} \xrightarrow[(\epsilon)]{} \subset_H \xrightarrow[(..^*)]{*} \xrightarrow[(\epsilon)]{\delta} \xrightarrow[(..^*)]{*} \\ \xrightarrow[(\epsilon)]{} \xrightarrow[(..^*)]{} \subset_H \xrightarrow[(..^*)]{*} \xrightarrow[(\epsilon)]{\delta} \end{cases}$$

Cette condition est aussi appelée ϵ-confluence [20].

Démonstration On montre

$$\xrightarrow[(\epsilon)]{} \xrightarrow[(..^*)]{k} \xrightarrow[(\epsilon)]{} \subset_H \xrightarrow[(..^*)]{*} \xrightarrow[(\epsilon)]{\delta} \xrightarrow[(.^*)]{*}$$

pour tout k par récurrence sur k. Comme la relation

$$\bigcup_{k\in\mathbf{N}} \left(\xrightarrow[(\epsilon)]{} \xrightarrow[(\ldots^*)]{k} \xrightarrow[(\epsilon)]{} \right)$$

est égale à $\xrightarrow[(\epsilon)]{} \xrightarrow[(\ldots^*)]{*} \xrightarrow[(\epsilon)]{}$, on peut appliquer le lemme précédent. ■

La deuxième partie de cette condition suffisante est une condition de commutativité qui est très forte. Une façon de l'affaiblir est de ne demander la commutativité qu'à partir d'une plus grande hauteur. On obtient la condition ci-dessous.

Proposition 2.35 *Une condition suffisante pour qu'une présentation soit résolvante est*

$$\wedge \begin{cases} \xrightarrow[(\epsilon)]{} \xrightarrow[(\ldots^*)]{} \subset_H \xrightarrow[(\ldots^*)]{*} \xrightarrow[(\epsilon)]{\delta} \\ \xrightarrow[(\epsilon)]{} \xrightarrow[(\cdot)]{*} \xrightarrow[(\epsilon)]{} \subset_H \xrightarrow[(\ldots^*)]{*} \xrightarrow[(\epsilon)]{\delta} \xrightarrow[(\ldots^*)]{*} \end{cases}$$

Démonstration On montre d'abord la réduction

$$\xrightarrow[(\epsilon)]{} \xrightarrow[(\ldots^*)]{*} \xrightarrow[(\epsilon)]{} \subset_H \xrightarrow[(\ldots^*)]{*} \xrightarrow[(\epsilon)]{\delta} \xrightarrow[(\ldots^*)]{*},$$

par la même méthode que celle utilisée pour la proposition 2.34. Puis en remarquant que la relation $\xrightarrow[(\ldots^*|\cdot)]{}$ est égale à $\xrightarrow[(\ldots^*)]{}$, on en déduit la réduction

$$\xrightarrow[(\epsilon)]{} \xrightarrow[(\ldots^*)]{*} \xrightarrow[(\epsilon)]{} \subset_H \xrightarrow[(\ldots^*)]{*} \xrightarrow[(\epsilon)]{\delta} \xrightarrow[(\ldots^*)]{*}.$$

qui est l'hypothèse du lemme 2.33. ■

Remarque 2.32 En fait on pourrait généraliser à la k-ième condition suffisante

$$\wedge \begin{cases} \xrightarrow[(\epsilon)]{} \xrightarrow[(\cdot^k\cdot^*)]{} \subset_H \xrightarrow[(\ldots^*)]{*} \xrightarrow[(\epsilon)]{\delta} \\ \xrightarrow[(\epsilon)]{} \xrightarrow[(\cdot(\cdot^\delta)^{k-2})]{*} \xrightarrow[(\epsilon)]{} \subset_H \xrightarrow[(\ldots^*)]{*} \xrightarrow[(\epsilon)]{\delta} \xrightarrow[(\ldots^*)]{*} \end{cases}$$

On ne se débarassera pas ainsi de la condition de commutativité, mais l'espoir est qu'à partir d'une certaine hauteur cette condition soit facile à obtenir. C'est ce que précise la partie suivante, nous montrerons ensuite comment ramener la dernière inclusion de la condition suffisante à une étude sur des motifs finis.

2.4.5 Conditions suffisantes de commutativité

Lemme 2.36 *Soit r un axiome $\sigma \doteq \tau$ tel qu'il existe une variable apparaissant exactement une fois dans τ à l'occurrence u et q fois dans σ aux occurrences $(u_i)_{i\in[1,q]}$, alors pour tout axiome s et tout occurrence v,*

$$\xrightarrow[r,(\epsilon)]{} \xrightarrow[s,(uv)]{} \subset \xrightarrow[s,(u_j v)]{j\in[1,q]} \xrightarrow[r,(\epsilon)]{}$$

Démonstration Une preuve filtrée par $\xrightarrow[r,(\epsilon)]{} \xrightarrow[s,(uv)]{}$ est de la forme:

$$\sigma\mu \xrightarrow[r,(\epsilon)]{} \tau\mu \xrightarrow[s,(uv)]{} \tau\mu[\rho/uv]$$

Notons α la variable $\tau_{/u}$. De la preuve

$$\tau\mu \xrightarrow[s,(vw)]{} \tau\mu[\rho/uv]$$

on peut extraire la sous-preuve à l'occurrence u

$$\alpha\mu \xrightarrow[s,(v)]{} \alpha\mu[\rho/v]$$

et l'appliquer à l'ensemble des occurrences $(u_j)_{j\in[1,q]}$ de α dans σ. En notant π le terme $\alpha\mu[\rho/v]$,

$$\sigma\mu \xrightarrow[s,(u_j v)]{j\in[1,q]} \sigma\mu[\pi/u_j]_{j\in[1,q]}$$

Notons μ' la restriction de μ à $\mathcal{V}\setminus\{\alpha\}$. Le terme du membre droit s'écrit $\sigma(\mu'+(\alpha\mapsto\pi))$. On peut donc lui appliquer la règle r à l'occurrence ϵ. On obtient $\tau(\mu'+(\alpha\mapsto\pi))$ qui, comme u est la seule occurrence[12] de α dans τ, s'écrit aussi $\tau\mu[\pi/u]$ et se simplifie en remplaçant π par $\tau\mu_{/u}[\rho/v]$ en $\tau\mu[\rho/uv]$. ■

Corollaire 2.37 *Si r est un axiome non potent[13] de A tel qu'il existe une variable apparaissant exactement une fois dans r_d à l'occurrence v alors pour tout axiome s,*

$$\xrightarrow[r,(\epsilon)]{}\xrightarrow[s,(v\bullet^*)]{} \subset \xrightarrow[s,(\bullet\bullet^*)]{*}\xrightarrow[r,(\epsilon)]{}$$

Démonstration Il suffit de remarquer que les occurrences u_i ne peuvent pas être nulles. ■

Remarque 2.33 L'hypothèse de linéarité est très forte. En particulier les deux conditions suffisantes des propriétés 2.34 et 2.35 comportent une condition de commutativité qui ne sera pas satisfaite en général[14] pour des axiomes non linéaires.

Pour des ensembles d'axiomes non linéaires, on pourra repartir de la condition

$$\xrightarrow[(\epsilon)]{}\xrightarrow[(\bullet\bullet^*)]{*}\xrightarrow[(\epsilon)]{}\xrightarrow{} \subset_H \xrightarrow[(\bullet\bullet^*)]{*}\xrightarrow[(\epsilon)]{\delta}\xrightarrow[(\bullet\bullet^*)]{*} \tag{1}$$

puis séparer l'hypothèse en la réunion des relations :

$$\bigcup_{\substack{r\in A\\ s\in A}} \xrightarrow[r,(\epsilon)]{}\xrightarrow[(\bullet\bullet^*)]{*}\xrightarrow[s,(\epsilon)]{}$$

Dans la plupart des cas, on aura :

$$\xrightarrow[r,(\epsilon)]{}\xrightarrow[(u\bullet^*)]{} \subset \xrightarrow[(u\bullet^*)]{*}\xrightarrow[(\epsilon)]{\delta}$$

pour toute occurrence u d'une variable dans r_d, ou bien

$$\xrightarrow[(u\bullet^*)]{}\xrightarrow[s,(\epsilon)]{} \subset \xrightarrow[(\epsilon)]{\delta}\xrightarrow[(u\bullet^*)]{*}$$

pour toute occurrence u d'une variable dans s_g. Pour chacun de ces couples (r,s), on pourra essayer de montrer les conditions de la remarque 2.32. Pour les autres couples, il faudra montrer directement la condition (1). On pourra éventuellement simplifier l'ensemble $(\bullet\bullet^*)$ en tenant compte des combinaisons d'axiomes impossibles, mais il faudra utiliser une méthode directe.

Exemple 2.34 Considérons les deux axiomes $r\uparrow$ et $s\uparrow$ respectivement égaux à $f(\alpha,\alpha)\doteq g(\alpha)$ et $b\doteq c$ et la théorie formée de ces deux axiomes et de leurs axiomes symétriques $r\downarrow$ et $s\downarrow$. L'axiome r n'est pas linéaire. L'axiome s l'est et vérifie donc

$$\xrightarrow[s,(\epsilon)]{}\xrightarrow[(\bullet\bullet^*)]{} \subset \xrightarrow[(\bullet\bullet^*)]{}\xrightarrow[s,(\epsilon)]{} \quad\wedge\quad \xrightarrow[(\bullet\bullet^*)]{}\xrightarrow[s,(\epsilon)]{} \subset \xrightarrow[s,(\bullet\bullet^*)]{}\xrightarrow[(\epsilon)]{}$$

[12] La condition de linéarité de τ en α est indispensable ici.

[13] Cette condition est particulièrement importante ici. Bien qu'elle soit supposée vérifiée dans toute cette partie, elle n'était toutefois pas nécessaire pour le lemme ci-dessus

[14] Dans la plupart des cas non linéaires, il est facile de construire des contre-exemples à la commutativité.

Il est facile de montrer que la seule relation restant à étudier est

$$\xrightarrow[r,(\epsilon)]{} \xrightarrow[(\bullet\bullet^*)]{*} \xrightarrow[r,(\epsilon)]{} .$$

Comme $\xrightarrow[(\bullet\bullet^*)]{}$ laisse invariant le symbole de tête, les seules combinaisons possibles sont des orientations opposées des deux axiomes r. Le cas $\xrightarrow[r\uparrow,(\epsilon)]{} \xrightarrow[(\bullet\bullet^*)]{*} \xrightarrow[r\downarrow,(\epsilon)]{}$, peut s'obtenir simplement puisque

$$\xrightarrow[r\uparrow,(\epsilon)]{} \xrightarrow[(\bullet\bullet^*)]{} \subset \xrightarrow[(\bullet\bullet^*)]{} \xrightarrow[r\uparrow(\bullet)]{}$$

car la condition de linéarité est vérifiée dans ce sens.

Le cas $\xrightarrow[r\downarrow,(\epsilon)]{} \xrightarrow[(\bullet\bullet^*)]{*} \xrightarrow[r\uparrow,(\epsilon)]{}$ se montre directement. Une preuve de cette relation est de la forme

$$g(\sigma) \xrightarrow[r\downarrow,(\epsilon)]{} f(\sigma,\sigma) \xrightarrow[(\bullet\bullet^*)]{*} f(\tau,\tau) \xrightarrow[r\uparrow,(\epsilon)]{} g(\tau)$$

La relation intermédiaire $\xrightarrow[(\bullet\bullet^*)]{*}$ est de la forme $\xrightarrow[(\underline{1}\bullet^*|\underline{2}\bullet^*)]{*}$. Comme les deux ensembles de chemins sont disjoints, on peut la réduire sur la relation $\xrightarrow[(\underline{1}\bullet^*)]{*} \xrightarrow[(\underline{2}\bullet^*)]{*}$. La preuve ci-dessus prouve donc la relation

$$f(\sigma,\sigma) \xrightarrow[(\underline{1}\bullet^*)]{*} f(\tau,\sigma).$$

On peut en extraire la sous-preuve à l'occurrence 1

$$\sigma \xrightarrow[(\bullet^*)]{*} \tau$$

d'où l'on en déduit une preuve

$$g(\sigma) \xrightarrow[(\underline{1}\bullet^*)]{*} g(\tau)$$

2.4.6 Etude de la résolvabilité sur des motifs minimaux

Nous raffinons la condition suffisante 2.35 de résolvabilité de façon à diminuer le nombre de relations élémentaires de prouvabilité en hypothèse.

Lemme 2.38 *S'il existe un ensemble H^a d'ensembles de termes, muni d'une relation nœthérienne $\gg$ telle que pour tout ensemble H de H^a les propriétés :*

$$\xrightarrow[(\epsilon)]{} \xrightarrow[(\bullet\bullet\bullet^*)]{} \subset_H \xrightarrow[(\bullet\bullet^*)]{*} \xrightarrow[(\epsilon)]{\delta} \tag{h_1}$$

$$\xrightarrow[(\epsilon)]{} \left(\xrightarrow[(k)]{\delta}\right)^{k\in\mathbf{N}} \xrightarrow[(\epsilon)]{} \subset_H \xrightarrow[(\bullet\bullet^*)]{*} \xrightarrow[(\epsilon)]{\delta} \xrightarrow[(\bullet\bullet^*)]{*} \tag{h_2}$$

$$f(\sigma_i)_{i\in[1,p]} \in H \Longrightarrow \forall i \in [1,p],\ \exists H' \in H_a, H \gg H' \wedge \sigma_i \in H' \tag{h_3}$$

sont vérifiées, alors la propriété

$$\xrightarrow{*} \subset_{H^a} \xrightarrow[(\bullet\bullet^*)]{*} \xrightarrow[(\epsilon)]{} \xrightarrow[(\bullet\bullet^*)]{*}$$

est vérifiée.

Démonstration Notons (H) la propriété

$$\xrightarrow{*} \subset_H \xrightarrow[(\bullet\bullet^*)]{*} \xrightarrow[(\epsilon)]{} \xrightarrow[(\bullet\bullet^*)]{*} .$$

Soit H^n le plus grand sous-ensemble de H^a dans lequel il n'existe pas de suite décroissante de longueur supérieure à n. Nous montrons (H^n) pour tout entier n par récurrence sur n. Comme H^0

est l'ensemble vide, nous avons (H^0). Supposons (H^n) et montrons (H^{n+1}). Il suffit de montrer (H) pour tout ensemble H de H^{n+1}. Compte tenu de la condition h_1, il suffit de montrer :

$$\xrightarrow[(\epsilon)]{}\xrightarrow[(\bullet\bullet^*)]{*}\xrightarrow[(\epsilon)]{}\subset_H \xrightarrow[(\bullet\bullet^*)]{*}\xrightarrow[(\epsilon)]{\delta}\xrightarrow[(\bullet\bullet^*)]{*}$$

Considérons une preuve dans H filtrée par $\xrightarrow[(\epsilon)]{}\xrightarrow[(\bullet\bullet^*)]{*}\xrightarrow[(\epsilon)]{}$. Comme des relations de prouvabilité à des chemins non préfixes l'un de l'autre commutent, nous pouvons réarranger la preuve en regroupant les chemins ayant même première direction :

$$\xrightarrow[(\epsilon)]{}\left(\xrightarrow[(k\bullet^*)]{*}\right)^{k\in[1,p]}\xrightarrow[(\epsilon)]{}$$

La sous-preuve extraite de $\xrightarrow[(k\bullet^*)]{*}$ au chemin $\xrightarrow[(k)]{}$ filtrée par $\xrightarrow[(\bullet^*)]{}$ est dans un ensemble H' tel que $H \gg H'$, donc H' est un élément de H^n. Par l'hypothèse (H^n), cette sous preuve se réduit en une sous preuve de H de la forme $\xrightarrow[(\bullet\bullet^*)]{*}\xrightarrow[(\epsilon)]{\delta}\xrightarrow[(\bullet\bullet^*)]{*}$. Donc la preuve initiale se réduit en une preuve

$$\xrightarrow[(\epsilon)]{}\left(\xrightarrow[(k\bullet\bullet^*)]{*}\xrightarrow[(k)]{\delta}\xrightarrow[(k\bullet\bullet^*)]{*}\right)^{k\in[1,p]}\xrightarrow[(\epsilon)]{}$$

En réordonnant les chemins disjoints, nous pouvons encore l'écrire

$$\xrightarrow[(\epsilon)]{}\left(\xrightarrow[(k\bullet\bullet^*)]{*}\right)^{k\in[1,p]}\left(\xrightarrow[(k)]{\delta}\right)^{k\in[1,p]}\left(\xrightarrow[(k\bullet\bullet^*)]{*}\right)^{k\in[1,p]}\xrightarrow[(\epsilon)]{}$$

Comme

$$\xrightarrow[(\epsilon)]{}\xrightarrow[(k\bullet\bullet^*)]{}\subset_{H_{n+1}}\xrightarrow[(\bullet\bullet^*)]{}\xrightarrow[(\epsilon)]{},$$

et par image miroir

$$\xrightarrow[(k\bullet\bullet^*)]{}\xrightarrow[(\epsilon)]{}\subset_{H_{n+1}}\xrightarrow[(\epsilon)]{}\xrightarrow[(\bullet\bullet^*)]{},$$

nous récrivons la preuve précédente en une preuve de H_{n+1} filtrée par

$$\xrightarrow[(\bullet\bullet^*)]{*}\xrightarrow[(\epsilon)]{}\left(\xrightarrow[(k)]{\delta}\right)^{k\in[1,p]}\xrightarrow[(\epsilon)]{}\xrightarrow[(\bullet\bullet^*)]{*}$$

Enfin, la condition h_2 nous permet de réécrire cette preuve en une preuve

$$\xrightarrow[(\bullet\bullet^*)]{*}\xrightarrow[(\epsilon)]{}\xrightarrow[(\bullet\bullet^*)]{*}$$

de H_{n+1}. ■

Corollaire 2.39 *S'il existe une famille croissante d'ensembles $(H_k^a)_{k\in\mathbf{N}}$ vérifiant les conditions de la proposition précédente, et dont la réunion est l'ensemble de toutes les preuves, alors A est résolvant.*

Le lemme précédent et son corollaire sont des conditions plus restrictives que la condition de la propriété 2.35, mais ils ont l'avantage de porter sur des relations de prouvabilité plus élémentaires. Si le nombre d'axiomes est fini, le nombre de relations élémentaires composant la relation en hypothèse est fini. Il sera alors possible d'etudier chacune des combinaisons séparément, en utilisant les propriétés suivantes.

Si $\sigma \xrightarrow[x]{} \tau$ alors pour toute substitution μ, $\sigma\mu \xrightarrow[x]{} \tau\mu$. On peut donc étendre l'ordre $<$ sur les termes à un ordre sur les preuves.

Définition 2.35 Soit H un sous-ensemble de $\mathcal{T}$. Un *diagramme minimal* dans H pour une relation de prouvabilité est une preuve minimale dans H de cette relation entre deux termes. □

Proposition 2.40 *Une relation $\underset{x}{\longrightarrow}$ se réduit sur une relation $\underset{y}{\longrightarrow}$ dans H si et seulement si tous les diagrammes minimaux pour $\underset{x}{\longrightarrow}$ se réduisent dans H sur la relation $\underset{y}{\longrightarrow}$.*

Démonstration Cette propriété est triviale, son intérêt est essentiellement de d'insister sur l'utilisitation de diagrammes minimaux. ■

Proposition 2.41 *Si tous les axiomes sont linéaires, c'est-à-dire qu'une variable ne peut pas apparaître plus d'une fois dans un membre d'axiome, et non potents, une condition suffisante pour que*

$$\underset{(\epsilon)}{\longrightarrow}\underset{(\cdot\cdot^*)}{\longrightarrow} \subset \underset{(\cdot\cdot^*)}{\longrightarrow}\underset{(\epsilon)}{\longrightarrow}$$

est que cette propriété soit vraie pour tous les diagrammes minimaux obtenus par superposition d'axiomes.

Démonstration Les diagrammes qui ne sont pas obtenus par superposition sont composés d'applications d'axiomes à des occurrences disjointes et dans ce cas la preuve se réduit par la propriété 2 de la proposition 2.28, ou bien d'occurrences préfixes qui rentrent dans le cadre du corollaire 2.37. ■

Exemple 2.35 La théorie Cg où le seul axiome est la commutativité gauche est résolvante. Soit @ le symbole commutatif à gauche. Le seul axiome est

$$x \mathbin{@} (y \mathbin{@} z) = y \mathbin{@} (x \mathbin{@} z)$$

Cette équation conserve la taille des termes. Nous considérons les ensembles H_n formés des termes de tailles inférieures ou égales à n. La condition h_3 est satisfaite. La condition h_1 est satisfaite car les axiomes sont linéaires et leurs occurrences non variables sont de longueur au plus 1.

Deux applications consécutives à la racine s'annulent. Comme

$$\underset{(\epsilon)}{\longrightarrow}\underset{(1)}{\longrightarrow} \subset_{H_n} \underset{(21)}{\longrightarrow}\underset{(\epsilon)}{\longrightarrow}$$

le seul cas à envisager pour montrer h_2 est $\underset{(\epsilon)}{\longrightarrow}\underset{(2)}{\longrightarrow}\underset{(\epsilon)}{\longrightarrow}$. Nous avons le diagramme suivant (lire de la gauche vers la droite)

$$\begin{array}{ccc}
 & \alpha \mathbin{@} (\beta \mathbin{@} (\gamma \mathbin{@} \delta)) & \\
{\scriptstyle(\epsilon)}\swarrow & & \searrow{\scriptstyle(2)} \\
\beta \mathbin{@} (\alpha \mathbin{@} (\gamma \mathbin{@} \delta)) & & \alpha \mathbin{@} (\gamma \mathbin{@} (\beta \mathbin{@} \delta)) \\
{\scriptstyle(2)}\updownarrow & \subset & \updownarrow{\scriptstyle(\epsilon)} \\
\beta \mathbin{@} (\gamma \mathbin{@} (\alpha \mathbin{@} \delta)) & & \gamma \mathbin{@} (\alpha \mathbin{@} (\beta \mathbin{@} \delta)) \\
{\scriptstyle(\epsilon)}\searrow & & \swarrow{\scriptstyle(2)} \\
 & \gamma \mathbin{@} (\beta \mathbin{@} (\alpha \mathbin{@} \delta)) &
\end{array}$$

2.4.7 Théories syntaxiques

Définition 2.36 Une théorie non potente est *syntaxique* si'l existe une présentation résolvante A pour laquelle $A(f,g)$ est fini pour tout couple homogène (f,g). On dit alors que A en est une présentation syntaxique. □

Soit A un ensemble d'axiomes non potents, $\sigma = f(\sigma_i)_{i\in[1,p]}$ et $\tau = g(\tau_j)_{j\in[1,q]}$ deux termes protégés par W. Si $f = g$, on note $Dec_1(\sigma \doteq \tau)$ le système :

$$\bigsqcup_{i\in[1,p]} \{\sigma_i \doteq \tau_i\}$$

On note $Gen\_ax_A^W(\sigma \doteq \tau)$ la disjonction des systèmes

$$\bigsqcup_{r\in D} \left(\bigsqcap_{i\in[1,p]} \{\sigma_i \doteq r_{g/i}\} \sqcap \bigsqcap_{j\in[1,q]} \{\sigma_j \doteq r_{d/j}\} \right)$$

où D est une copie de $A(f,g)$ en dehors de W.

On note appelle *généralisation par les axiomes* le remplacement de l'équation $\sigma \doteq \tau$ par la disjonction[15] de systèmes :

$$Gen_A^W(\sigma \doteq \tau) = \begin{cases} Gen\_ax_A^W(\sigma \doteq \tau) & \text{si } f = g \\ Gen\_ax_A^W(\sigma \doteq \tau) \sqcup Dec_1(\sigma \doteq \tau) & \text{sinon} \end{cases}$$

La généralisation par les axiomes étend la décomposition pour les théories syntaxiques.

Théorème 4 *Si A est une théorie syntaxique non potente, alors pour toute équation indécomposable e, on a $Gen_A^W(e) \equiv\!\!\rangle_A^W e$.*

La décomposition par les axiomes peut donc être utilisée comme opération de mutation pourvu qu'on puisse lui associer une mesure décroissante.

Démonstration Cette preuve suit exactement la preuve de [36] mais avec des hypothèses légèrement différentes.

Notons f et g les symboles $\sigma(\epsilon)$ et $\tau(\epsilon)$ et p et q leurs arités. Soit μ un unificateur de $Gen_A^W(\sigma \doteq \tau)$ en dehors de W. Montrons que μ est solution de τ. Si $f = g$, alors $Gen_A^W(\sigma \doteq \tau)$ est égal à

$$\bigsqcup_{i \in [1,p]} \{\sigma_i \doteq \tau_i\}$$

On a ainsi $\sigma_i\mu =_A \tau_i\mu$ donc μ est solution de $\sigma \doteq \tau$. Sinon, $Gen_A^W(\sigma \doteq \tau)$ est égal à

$$\bigsqcup_{r \in D} \left(\bigsqcap_{i \in [1,p]} \{\sigma_i \doteq r_{g/i}\} \sqcap \bigsqcap_{j \in [1,q]} \{\sigma_j \doteq r_{d/j}\} \right)$$

Il existe donc un axiome r dans D tel que

$$(\forall i \in [1,\ p], \sigma\mu_{/i} =_A r_{g/i}\mu) \wedge (\forall j \in [1,\ q], \tau\mu_{/j} =_A r_{d/j}\mu)$$

Comme la théorie est résolvante, on en déduit $\sigma\mu =_A \tau\mu$.

Réciproquement, soit μ est un unificateur de $\sigma \doteq \tau$ en dehors de W de domaine inclus dans W. La théorie étant résolvante, si $f = g$ alors $\sigma\mu_{/i} = \tau\mu_{/i}$ pour tout i de $[1,p]$ donc μ est solution de $Gen_A^W(\sigma \doteq \tau)$. Sinon, il existe un axiome r dans $A(f,g)$ et une substitution ν telle que

$$(\forall i \in [1,\ p], \sigma\mu_{/i} =_A r_{g/i}\nu) \wedge (\forall j \in [1,\ q], \tau\mu_{/j} =_A r_{d/j}\nu)$$

Il existe nécessairement une copie par θ de r dans D. Comme le domaine de $\theta^{-1}\nu$ n'intercepte pas celui de μ, la substitution $\mu + \theta^{-1}\nu$ est une extension de μ. C'est une solution de

$$\bigsqcap_{i \in [1,p]} \{\sigma_i \doteq r_g\theta_{/i}\} \sqcap \bigsqcap_{j \in [1,q]} \{\sigma_j \doteq r_d\theta_{/j}\}$$

donc a fortiori solution de la disjonction $Gen_A^W(\sigma \doteq \tau)$. ■

Remarque 2.36 La présentation précédente de la commutativité gauche est syntaxique, mais la généralisation correspondante ne fournit pas une opération de mutation car elle n'est pas nœthérienne. Une solution est proposée dans [36].

[15] La condition de finitude dans la définition d'une théorie syntaxique assure qu'il s'agit bien d'une disjonction.

Chapitre 3

Typage dans le langage ML

Ce chapitre présente la synthèse de types dans le langage ML. Le premier algorithme de synthèse de types pour le langage ML est celui de L. Damas et R. Milner dans [18]. Une approche simple et complète de ML et de son typage est présentée dans [13]. L'algorithme de typage a été implanté dans de nombreux langages de la famille de ML souvent avec des extensions permettant de traiter correctement les constructions impératives. Notamment l'extension avec des variables de type faibles, étudiée par Mads Tofte dans [58]. Beaucoup d'études ont également été faites sur l'extension du typage de la récursion [46, 33, 32, 34, 35]. Mais ce n'est que très récemment que l'efficacité de l'algorithme de synthèse de types a été étudiée, notamment dans [31], où Paris Kanellakis et John C. Mitchell donnent des bornes inférieures et supérieures pour sa complexité dans le cas le pire.

Habituellement la synthèse de types en ML est spécifiée par un système de règles d'inférence, desquelles est extrait un algorithme. Or il existe différentes variantes du système d'inférence initial. Toutes les présentations sont bien sûr équivalentes, mais cela n'est pas toujours prouvé. D'autre part, les algorithmes extraits sont souvent inefficaces ou très éloignés de la spécification initiale, et les optimisations deviennent des ruses qu'il est difficile de justifier théoriquement.

Nous allons dans ce chapitre partir du système d'inférence le plus couramment utilisé. Nous le transformerons progressivement en un système d'inférence final, duquel nous extrairons très simplement un algorithme très efficace. Les systèmes intermédiaires sont tous très voisins les uns des autres, car chaque transformation s'attachera à ne résoudre qu'un seul des défauts du système précédent. La multiplicité des systèmes intermédiaires qui pouvait paraître lourde a priori est en fait un double avantage. D'une part nous montrerons très facilement la correction de chacune des transformations. D'autre part, nous obtiendrons à l'issue une panoplie de présentations de la synthèse de types en ML toutes équivalentes, ce qui lèvera l'ambiguïté citée ci-dessus.

Le résultat de ce chapitre est triple. Le premier intérêt est d'ordre pratique. L'algorithme que nous obtenons est à la fois très simple et très efficace, à notre connaissance ; c'est le plus efficace des algorithmes réellement implantés. Le second est d'ordre théorique. La preuve de correction de l'algorithme nous montrera l'existence d'un type principal pour la synthèse de types de ML. Ce résultat est obtenu pour des types de premier ordre munis d'une théorie équationnelle régulière. Il étend donc le résultat de L. Damas et R. Milner. Enfin le dernier est plus d'ordre stratégique, car les performances de l'algorithme seront liées à une incrémentalité des simplifications, qui pourra servir de guide dans toute extension du typage de ML, notamment pour l'ajout de relations d'inclusion sur les types.

3.1 Lambda calcul simplement typé

Définition 3.1 Soit V_E un ensemble dénombrable de variables de termes et un ensemble dénombrable de constantes. L'ensemble des termes du λ-calcul est le plus petit ensemble, noté Λ_E, vérifiant :

1. Si x est une variable alors x est un terme.
2. Si M et N sont deux termes alors l'application de M à N, notée MN est un terme.
3. Si M est un terme et x une variable alors l'abstraction de x dans M, notée $\lambda x.M$ est un terme.

□

Nous considérons l'alphabet gradué $\mathcal{C}$ composé du seul symbole $\rightarrow$ d'arité deux et un ensemble dénombrable de variables de types $\mathcal{V}$. Nous notons $\mathcal{T}$ l'algèbre libre $\mathcal{T}(\mathcal{C}, \mathcal{V})$ dont les éléments seront appelés *types*.

Une *assertion* est une paire notée $x : \sigma$ d'une variable x de Λ_E et d'un type σ. Un *contexte* est une suite finie d'assertions. Les contextes seront désignés par les lettres Γ et Δ. La concaténation du contexte Γ avec l'assertion $x : \sigma$ est notée $\Gamma[x : \sigma]$. On note Γ_x la sous-suite extraite de Γ constituée des assertions $x : \sigma$ de Γ. Si cette suite est vide, on dit que la variable x est *libre* dans Γ, sinon on note $\Gamma(x)$ le type σ tel que $x : \sigma$ soit le dernier élément de Γ_x et on dit que la variable x est *déclarée* de type σ dans Γ.

On définit une relation entre un contexte Γ, un terme M et un type σ, dite *jugement de typage* et notée $\Gamma \vdash M : \sigma$ comme la plus petite relation vérifiant :

$$(VAR) \qquad \frac{[\sigma = \Gamma(x)]}{\Gamma \vdash x : \sigma}$$

$$(FUN) \qquad \frac{\Gamma[x : \sigma] \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \rightarrow \tau}$$

$$(APP) \qquad \frac{\Gamma \vdash M : \sigma \rightarrow \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\ N : \tau}$$

Pour être rigoureux, il faudrait ajouter la règle d'égalité structurelle

$$(EQUAL) \qquad \frac{\Gamma \vdash M : \sigma \quad [\sigma = \tau]}{\Gamma \vdash M : \tau}$$

mais celle-ci est souvent sous-entendue et repoussée dans le méta-langage en disant que les méta-variables σ, τ et ρ désignent des termes de types modulo l'égalité structurelle. En fait on pourrait se contenter d'ajouter une prémisse structurelle à la règle (APP) :

$$(APP) \qquad \frac{\Gamma \vdash M : \sigma \rightarrow \tau \quad \Gamma \vdash N : \rho \quad [\sigma = \rho]}{\Gamma \vdash M\ N : \tau}$$

L'une ou l'autre des solutions sont satisfaisantes. Nous adopterons la convention que deux occurrences d'une même méta-variable dans un même contexte désigneront toujours le même objet.

Un arbre de preuve est aussi appelé une dérivation. Sa taille est donc le nombre de règles utilisées dans la preuve.

Il existe un algorithme de synthèse de types qui détermine s'il existe un typage pour un lambda terme de Λ_E. Ce résultat peut se déduire du résultat corrrespondant pour le langage ML, mais il peut aussi être étudié directement [1] [26].

Le système de typage précédent est le plus simple des systèmes de typage du lambda-calcul. On trouvera une description et une approche générale des différents systèmes de typage dans [28].

3.2 Présentation usuelle du typage dans le langage ML

3.2.1 Le langage ML

Nous nous limiterons à un noyau pur de ML. Le langage est alors très simple : il s'agit du λ-calcul étendu avec une nouvelle construction.

Définition 3.2 L'ensemble des termes de ML, noté ML_E, est le plus petit ensemble vérifiant :

1. $V_E \subset ML_E$
2. Si M et N sont deux termes alors MN est un terme.
3. Si M est un terme et x une variable alors $\lambda x.M$ est un terme.

4. Si M et N sont des termes alors la liaison de M à x dans N, notée *let* $x = M$ *in* N, est un terme.

□

Dans un lambda-terme, un *radical* est un sous-terme $(\lambda x.M)N$ formé par l'application d'une abstraction à un terme. La construction *let* ... *in* ... n'est qu'une abbréviation pour un radical. Elle permet de marquer certains radicaux d'un terme, qui seront typés de façon plus générale. Un terme de Λ_E est un terme de ML qui ne contient pas de liaison. On peut lui associer un autre terme obtenu en marquant tous ses radicaux.

Hypothèses

Nous considérons comme pour Λ_E, l'algèbre libre $\mathcal{T}(\mathcal{C}, \mathcal{V})$. Pour pouvoir appliquer tous les résultats de ce chapitre aux extensions des chapitres suivants, nous considérons le cas plus général où $\mathcal{T}$ est signé sur un ensemble de sortes $\mathcal{K}$. Nous munissons $\mathcal{T}$ d'une théorie équationnelle A régulière. La régularité assure que deux types A-égaux ont le même ensemble de variables, ce qui sera partout utilisé.

3.2.2 Types quantifiés

Une spécification simple du typage de ML consiste à réduire tous les radicaux marqués par des *let* sans réduire les radicaux créés, puis à typer simplement le terme réduit. Mais cette méthode est coûteuse car la réduction d'une liaison *let* $x = M$ *in* N non linéaire (c'est-à-dire x apparaît plusieurs fois dans N) duplique le terme M dans le terme $N(x \mapsto M)$.

Or tous les résidus M dans $N(x \mapsto M)$ généreront les mêmes contraintes de typages. Un moyen d'éviter la duplication du calcul consiste à typer M une seule fois et considérer le type obtenu comme un modèle. Les contraintes sur les résidus M dans $N(x \mapsto M)$ se reviendront à dire que leurs types doivent être plus précis que celui du modèle. Les systèmes de typage qui seront successivement présentés dans ce chapitre diffèrent essentiellement par leur façon de représenter les modèles.

Les modèles sont habituellement formalisés par des types avec quantificateurs en tête. Les types quantifiés en tête sont définis comme le plus petit ensemble $\mathcal{Q}$ satisfaisant :

1. $\mathcal{T} \subset \mathcal{Q}$.
2. Si σ est dans $\mathcal{Q}$ alors $\forall\alpha \cdot \sigma$ est dans $\mathcal{Q}$

   On note couramment $\forall\alpha\beta \cdot \sigma$ au lieu de $\forall\alpha \cdot \forall\beta \cdot \sigma$.

En fait, l'ordre de quantification n'importe pas, car toutes les formules seront exprimées indépendamment de cet ordre. De plus, on ne fera jamais d'application de type.

Cette notation avec quantificateurs qui semble venir de la logique ou du lambda-calcul de second ordre où l'application de type est essentielle, l'ordre de liaison des variables devenant important, est donc trop précise. Nous choisirons la solution plus simple qui consiste à représenter les variables liées par un ensemble.

Définition 3.3 L'ensemble $\mathcal{T}_q$ des *types quantifiés* est le produit cartésien de l'ensemble des parties finies de $\mathcal{V}$ par l'ensemble des types simples $\mathcal{T}$. □

La première projection d'un type quantifié est l'ensemble de ses variables liées. La deuxième projection est son type simple. On confondra parfois dans le vocabulaire ou dans la notation un type quantifié dont l'ensemble des variables liées est vide avec son type simple. On désignera les ensembles de variables liées par les lettres $\mathcal{X}$, $\mathcal{Y}$ et $\mathcal{Z}$.

Définition 3.4 Soit $\mathcal{Q}$ un ensemble. Une relation sur $\mathcal{Q} \times \mathcal{T}$ est appelée relation d'*instance simple* et est notée $\leq^{\mathcal{Q}}$ si pour tout élément $\sigma^{\mathcal{Q}}$ de $\mathcal{Q}$ et tous types τ et ρ de $\mathcal{T}$ tels que $\sigma^{\mathcal{Q}} \leq^{\mathcal{Q}} \tau$ et $\tau \leq \rho$, on ait aussi $\sigma^{\mathcal{Q}} \leq^{\mathcal{Q}} \rho$.
Si $\leq^{\mathcal{Q}}$ et $\leq^{\mathcal{R}}$ sont deux relations d'instance sur $\mathcal{Q}$ et $\mathcal{R}$, on définit une relation d'instance sur $\mathcal{Q} \times \mathcal{R}$ par

$$\forall \sigma^{\mathcal{Q}} \in \mathcal{Q},\ \tau^{\mathcal{R}} \in \mathcal{R}, \left(\sigma^{\mathcal{Q}} \leq^{\mathcal{QR}} \tau^{\mathcal{R}} \iff \left(\forall \rho \in \mathcal{T},\ \sigma^{\mathcal{Q}} \leq^{\mathcal{Q}} \rho \Longrightarrow \tau^{\mathcal{R}} \leq^{\mathcal{R}} \rho\right)\right)$$

Deux objets sont $\mathcal{QR}$-*équivalents* s'ils sont $\mathcal{QR}$-instances l'un de l'autre.
Une relation d'instance simple $\leq^{\mathcal{Q}}$ sur $\mathcal{Q}$ induit une relation d'A-*instance simple* $\leq_A^{\mathcal{Q}}$ définie par

$$\forall \sigma^{\mathcal{Q}},\ \forall \tau \in \mathcal{T},\ \left(\sigma_{\mathcal{Q}} \leq_A^{\mathcal{Q}} \tau \iff \exists \rho,\ \rho =_A \tau \wedge \sigma_{\mathcal{Q}} \leq^{\mathcal{Q}} \rho\right)$$

Nous étendons les relations d'instance à des contextes de la façon suivante: un contexte Γ est une $\mathcal{QR}$-instance d'un contexte Δ si pour tout entier n, l'assertion $\Gamma(n)$ déclare une variable d'une valeur $\sigma^{\mathcal{Q}}$ et l'assertion $\Delta(n)$ déclare la même variable d'une valeur $\tau^{\mathcal{R}}$ telle que $\sigma^{\mathcal{Q}}$ soit une $\mathcal{QR}$-instance de $\tau^{\mathcal{R}}$.
On en déduit naturellement une relation d'x-A-*équivalence* . □

Définition 3.5 Un type simple τ est une *instance simple* d'un type quantifié $\mathcal{X} \cdot \sigma$, et on note $\mathcal{X} \cdot \sigma \leq^q \tau$ s'il existe une substitution $\mu : \mathcal{X} \to \mathcal{V}$ telle que $\tau = \sigma\mu$. L'ensemble des instances simples d'un type quantifié $\mathcal{X} \cdot \sigma$ est donc

$$\{\sigma\mu \mid \mu : \mathcal{X} \to \mathcal{V}\}.$$

□

Proposition 3.1 *La relation d'A-instance quantifiée est caractérisée par :*

$$\mathcal{X} \cdot \sigma \leq_A^q \mathcal{Y} \cdot \tau \iff \wedge \begin{cases} \exists \mu : \mathcal{X} \to \mathcal{V},\ \tau =_A \sigma\mu \\ (\mathcal{V}(\sigma) \setminus \mathcal{X}) \cap \mathcal{Y} = \emptyset \end{cases}$$

Démonstration Supposons que $\mathcal{Y} \cdot \tau$ soit une A-instance de $\mathcal{X} \cdot \sigma$. En particulier τ est une A-instance simple de $\mathcal{X} \cdot \sigma$. Il existe donc une substitution $\mu : \mathcal{X} \to \mathcal{V}$ telle que $\tau =_A \sigma\mu$. Si α est une variable apparaissant dans σ en dehors de $\mathcal{X}$, alors α se retrouvera dans toutes les A-instances de $\mathcal{X} \cdot \sigma$, car la théorie A est régulière. Il est donc nécessaire qu'elle se retrouve aussi dans toutes les A-instances de $\mathcal{Y} \cdot \tau$, ce qui veut dire qu'elle ne doit jamais pouvoir être substituée donc être en dehors de $\mathcal{Y}$.

Réciproquement supposons que $(\mathcal{V}(\sigma) \setminus \mathcal{X}) \cap \mathcal{Y}$ et $\mu : \mathcal{X} \to \mathcal{V}$. Une instance simple de $\mathcal{Y} \cdot \sigma\mu$ est A-égale à $\sigma\mu\nu$ par une substitution $\nu : \mathcal{Y} \to \mathcal{V}$. Montrons la formule $\mu\nu \restriction \mathcal{V}(\sigma) : \mathcal{X} \to \mathcal{V}$. Soit α une variable de $\mathcal{V}(\sigma) \setminus \mathcal{X}$. Elle est en dehors du domaine de μ, donc n'est pas modifiée par μ. Elle est en dehors de $\mathcal{Y}$ donc n'est pas non plus modifiée par ν. Ainsi, le domaine de $\mu\nu \restriction \mathcal{V}(\sigma)$ est inclus dans $\mathcal{X}$. ■

3.2.3 Système d'inférence quantifié

Définition 3.6 Le système (ML_q) est l'ensemble des règles d'inférence suivantes formées sur des jugements de la forme $\Gamma \vdash_q M : \mathcal{X} \cdot \sigma$ où :

- Γ est un contexte liant des variables à des types quantifiés.
- M est un terme de ML_E
- $\mathcal{X} \cdot \sigma$ est un type quantifié. Nous abrégeons $\emptyset.\sigma$ en σ.

$$(VAR) \qquad \frac{[\mathcal{X} \cdot \sigma = \Gamma(x)]}{\Gamma \vdash_q x : \mathcal{X} \cdot \sigma}$$

$$(INST) \qquad \frac{\Gamma \vdash_q M : \mathcal{X} \cdot \sigma \quad [\mathcal{X} \cdot \sigma \leq_A^q \mathcal{Y} \cdot \tau]}{\Gamma \vdash_q M : \mathcal{Y} \cdot \tau}$$

$$(GEN) \qquad \frac{\Gamma \vdash_q M : \mathcal{X} \cdot \sigma \quad [\mathcal{Y} \subset \overline{\mathcal{V}}(\Gamma)]}{\Gamma \vdash_q M : \mathcal{X} \cup \mathcal{Y} \cdot \sigma}$$

$$(FUN) \qquad \frac{\Gamma[x : \sigma] \vdash_q M : \tau}{\Gamma \vdash_q \lambda x.M : \sigma \to \tau}$$

$$(APP) \qquad \frac{\Gamma \vdash_q M : \sigma \to \tau \quad \Gamma \vdash_q N : \sigma}{\Gamma \vdash_q M\ N : \tau}$$

$$(LET) \qquad \frac{\Gamma \vdash_q M : \mathcal{X} \cdot \sigma \quad \Gamma[x : \mathcal{X} \cdot \sigma] \vdash_q N : \tau}{\Gamma \vdash_q \text{let } x = M \text{ in } N : \tau}$$

□

Il faut noter que la règle d'égalité

$$(EQUAL) \quad \frac{\Gamma \vdash_q M : \sigma \quad \sigma =_A \tau}{\Gamma \vdash_q M : \tau}$$

est contenue dans la règle $(INST)$.

Lorsque la théorie A est vide, le système précédent est la présentation usuelle du système d'inférence de ML modulo le remplacement des quantificateurs par des ensembles de variables. Pour une variable, la liberté est une propriété locale, car elle dépend du contexte dans lequel la variable est utilisée. Ceci crée le phénomène de capture de variables qui se produit lorsqu'une variable libre est introduite dans un contexte dans lequel elle se retrouve, et qui rend difficile l'énoncé de propriétés globales sur le système (ML_q). Nous allons remplacer la notion de variable liée par celle de variable générique qui sera globale.

3.3 Système d'inférence générique

Nous introduisons un ensemble de variables génériques $\mathcal{V}^g$ isomorphe à $\mathcal{V}$ par ϕ. On notera $\mathcal{V}_g$ la réunion $\mathcal{V} \cup \mathcal{V}^g$ et $\mathcal{T}_g$ l'algèbre libre $\mathcal{T}(\mathcal{C}, \mathcal{V}_g)$ dont les éléments seront appelés *types génériques*. Par la suite nous considérerons que $\mathcal{T}$ est un sous-ensemble de $\mathcal{T}_g$. Aussi, un type générique (appartenant à $\mathcal{T}_g$) qui appartient aussi à $\mathcal{T}$ sera dit type simple. L'ensemble $\mathcal{T}^g$ est muni de la même théorie équationnelle que $\mathcal{T}$.

Notation

- σ, τ et ρ représentent toujours des types simples.
- σ_g, τ_g et ρ_g représentent des types génériques.
- μ, ν et ξ représentent des substitutions de $\mathcal{T}$ étendues naturellement à des substitutions dans $\mathcal{T}_g$, c'est-à-dire des substitutions de $\mathcal{T}_g$ de domaine non générique et envoyant les variables non génériques sur des types non génériques.
- μ^g, ν^g et ξ^g représentent des substitutions dans $\mathcal{T}_g$ de domaine inclus dans $\mathcal{V}^g$, et d'image incluse dans $\mathcal{T}_g$.
- θ^g et η^g représentent des substitutions bijectives de $\mathcal{V}$ dans $\mathcal{V}^g$ ou de $\mathcal{V}^g$ dans $\mathcal{V}$.

Définition 3.7 Une *instance simple* d'un type générique σ_g est un type obtenu par une substitution $\mu^g : \mathcal{V}^g(\sigma_g) \to \mathcal{T}$ de σ_g. □

Définition 3.8 Le système (ML_g) est l'ensemble des règles d'inférence suivantes formées sur des jugements de la forme $\Gamma \vdash_g M : \sigma_g$ où :

- Γ est un contexte liant des variables à des types génériques.
- M est un terme de ML_E.
- σ_g est un type générique, parfois restreint à un type simple, ce qui sera indiqué par une méta variable de type simple.

$$(VAR) \qquad \frac{[\sigma_g = \Gamma(x)]}{\Gamma \vdash_g x : \sigma_g}$$

$$(INST) \qquad \frac{\Gamma \vdash_g M : \sigma_g \quad [\sigma_g \leq^g_A \tau_g]}{\Gamma \vdash_g M : \tau_g}$$

$$(GEN) \qquad \frac{\Gamma \vdash_g M : \sigma_g \quad [D(\theta^g) \subset \overline{\mathcal{V}}(\Gamma)]}{\Gamma \vdash_g M : \sigma_g\theta^g}$$

$$(FUN) \qquad \frac{\Gamma[x : \sigma] \vdash_g M : \tau}{\Gamma \vdash_g \lambda x.M : \sigma \to \tau}$$

$$(APP) \qquad \frac{\Gamma \vdash_g M : \sigma \to \tau \quad \Gamma \vdash_g N : \sigma}{\Gamma \vdash_g M\ N : \tau}$$

$$(LET) \qquad \frac{\Gamma \vdash_g M : \sigma_g \quad \Gamma[x : \sigma_g] \vdash_g N : \tau}{\Gamma \vdash_g let\ x = M\ in\ N : \tau}$$

□

Proposition 3.2 *Pour tous contextes A-équivalents* Γ *et* Δ *et tous types A-équivalents* σ *et* τ*,*

$$\Gamma \vdash M : \sigma \iff \Delta \vdash N : \tau$$

Démonstration Cette propriété sera énoncée par la proposition 3.8, démontrée dans le système (ML_h), puis sera transportée successivement dans les systèmes (ML_s) puis (ML_g) par les propriétés d'équivalence 3.6 et 3.5. ■

Remarque 3.1 Tout type quantifié est équivalent à un type générique et réciproquement.

Démonstration Il suffit d'associer à $\mathcal{X} \cdot \sigma$ le type $\sigma\theta^g$ où $\theta^g : \mathcal{X} \to \mathcal{V}^g$. On pourra prendre en particulier θ^g égale à $\phi \restriction \mathcal{X}$ et on appellera φ cette transformation. Réciproquement un type générique σ_g est équivalent au type quantifié $\tau_g\eta^g$ si $\eta^g : \mathcal{V}^g(\tau_g) \to \overline{\mathcal{V}}(\tau_g)$. ■

Lemme 3.3 *Les deux systèmes* (ML_q) *et* (ML_g) *sont A-équivalents, c'est-à-dire, pour tout contexte* Γ *et tout contexte générique* Δ *A-équivalents, et pour tout type quantifié* $\mathcal{X} \cdot \sigma$ *et tout type générique* τ_g *A-équivalents,*

$$\Gamma \vdash_q M : \mathcal{X} \cdot \sigma \iff \Delta \vdash_g M : \tau_g$$

Démonstration Pour montrer la première partie, il suffit d'étendre φ de façon naturelle en un homomorphisme des contextes, des jugements puis des dérivations de (ML_q) dans les contextes, les jugements et les dérivations de (ML_g). La transformation φ conserve la validité des dérivations règle à règle :

- Les règles (VAR), $(INST)$, (FUN), (APP) et (LET) se correspondent directement par φ.
- Règle (GEN) Il suffit de choisir θ^g égale à $\varphi \restriction \mathcal{Y}$.

On a donc

$$\Gamma \vdash_q M : \mathcal{X} \cdot \sigma \Longrightarrow \varphi(\Gamma) \vdash_g M : \varphi(\mathcal{X} \cdot \sigma).$$

Il découle de la propriété 3.2 que

$$\varphi(\Gamma) \vdash_g M : \varphi(\mathcal{X} \cdot \sigma) \iff \Delta \vdash_q M : \tau_g,$$

ce qui montre l'implication directe.

La réciproque est plus délicate car pour associer un type quantifié à un type générique, il faudra éventuellement renommer certaines variables afin d'éviter leur capture. Nous la montrons par récurrence sur la longueur de la dérivation dans (ML_g). On notera $\vdash^\ell$ au lieu de $\vdash$ pour un jugement dérivable en ℓ étapes.

Pour une dérivation de longueur 1, si $\Delta \vdash_g x : \Gamma(x)$, alors pour tout contexte quantifié Γ A-équivalent à Δ, on peut bien sûr dériver le jugement $\Gamma \vdash_q x : \Gamma(x)$. Supposons que si un jugement générique est prouvable en ℓ étapes, alors tout jugement quantifié A-équivalent est prouvable, et considérons un jugement valide $\Delta \vdash_g^{\ell+1} M : \tau_g$. Nous montrons par cas sur la dernière étape de la dérivation du jugement précédent que tout jugement quantifié A-équivalent est valide.

- Le cas (VAR) est impossible si $\ell > 1$.
- Les cas (APP), (FUN) et (LET) se déduisent immédiatement de l'hypothèse de récurrence.
- Dans le cas $(INST)$, la dernière dérivation est

$$\frac{\Delta \vdash_g^n M : \tau_g \quad [\tau_g \leq_A^g \rho_g]}{\Delta \vdash_g^{n+1} M : \rho_g}$$

  Soit $\Gamma \vdash_q M : \mathcal{X} \cdot \sigma$ une dérivation A-équivalente à la conclusion. Il existe un type $\mathcal{Z} \cdot \rho$ A-équivalent à τ_g. Le jugement $\Gamma \vdash_q M : \mathcal{Z} \cdot \rho$ A-équivalent à la prémisse est donc valide. Comme ρ_g est une A-instance de τ_g et $\mathcal{X} \cdot \sigma$ est A-équivalent à ρ_g, $\mathcal{X} \cdot \sigma$ est une A-instance de $\mathcal{Z} \cdot \rho$. Par la règle $(INST)$ de (ML_q) on en déduit la validité du jugement $\Gamma \vdash_q^n M : \mathcal{X} \cdot \sigma$.

- Dans le cas (GEN), la dernière dérivation est

$$\frac{\Delta \vdash_g^n M : \tau_g \quad [D(\theta^g) \subset \overline{\mathcal{V}}(\Delta)]}{\Delta \vdash_g^n M : \tau_g \theta^g} \quad (GEN)$$

  Soit $\Gamma \vdash_q M : \mathcal{Z} \cdot \rho$ une dérivation A-équivalente à la conclusion. Il existe un type $\mathcal{X} \cdot \sigma$ A-équivalent à τ_g. Le jugement $\Gamma \vdash_q M : \mathcal{X} \cdot \sigma$ A-équivalent à la prémisse est donc valide. Comme la théorie est régulière, $\mathcal{V}(\Gamma)$ et $\mathcal{V}(\Delta)$ sont égaux donc le domaine de θ^g n'intercepte pas Γ, et la dérivation

$$\frac{\Gamma \vdash_q M : \mathcal{X} \cdot \sigma \quad [D(\theta^g) \subset \overline{\mathcal{V}}(\Gamma)]}{\Gamma \vdash_q M : \mathcal{X} \cup D(\theta^g) \cdot \sigma} \quad (GEN)$$

  Il est facile de se convaincre que $\mathcal{X} \cup D(\theta^g) \cdot \sigma$ est A-équivalent à $\tau_g \theta^g$, donc aussi à $\mathcal{Z} \cdot \rho$, ce qui permet de dériver le jugement $\Gamma \vdash_q M : \mathcal{Z} \cdot \rho$ par la règle $(INST)$.

■

Le système (ML_g) est clairement plus simple que le système (ML_q). Il protège les variables génériques en leur donnant une nature différente des variables simples. En particulier, la preuve ci-dessus et essentiellement le cas (GEN) de la réciproque intègre une fois pour toutes le phénomène de capture de variables. Mais la formulation précédente n'est pas entièrement satisfaisante. Nous avons défini les jugements de (ML_g) comme étant de la forme $\Gamma \vdash_g M : \sigma_g$ où σ_g est un type générique. Or certaines règles ne sont énoncées que dans le cas où σ_g est un type simple. Il est donc naturel de se demander si toutes les règles peuvent restreindre σ_g à être un type simple ou au contraire toutes l'autoriser à être générique.

La première approche revient à considérer que les règles sont essentiellement non génériques. On s'aperçoit alors que les types génériques se comportent comme des modèles desquels on pourra copier plusieurs instances. On ne saura pas calculer sur ces modèles mais simplement les mémoriser dans les contextes. C'est cette approche que nous développerons dans la prochaine partie avec le système (ML_s). C'est de ce système que sont inspirés la plupart des algorithmes implantés.

Mais il semble aussi que les règles dans lesquelles σ_g est restreint à un type simple puissent être étendues afin de lever cette restriction. Tous les types deviennent possiblement génériques, ce qui semble dans un premier temps plus économique car on pourra faire des calculs sans prendre de copies. Mais les calculs que l'on sait faire sans copies sont en fait très limités. Cette approche ne sera pas appronfondie.

Une solution moyenne sera étudiée avec le système hiérarchisé (ML_h). Tout en conservant le mécanisme de copie de la première solution, elle réduira le coût de la fabrication du modèle au minimum en le construisant de façon incrémentale. Nous verrons qu'en découle un algorithme très efficace.

3.4 Système d'inférence simplifié

Définition 3.9 Le système (ML_s) est l'ensemble des règles d'inférence suivantes formées sur des jugements de la forme $\Gamma \vdash_s M : \sigma$ où :

- Γ est un contexte de déclarations génériques $x : \tau_g$
- M est un terme de ML_E
- σ est un type non générique.

$$(EQUAL) \qquad \frac{\Gamma \vdash_s M : \sigma \quad [\sigma =_A \tau]}{\Gamma \vdash_s M : \tau}$$

$$(VAR) \qquad \frac{[\sigma_g = \Gamma(x)] \quad [\mu^g : \mathcal{V}^g(\sigma_g) \to \mathcal{T}]}{\Gamma \vdash_s x : \sigma_g \mu^g}$$

$$(FUN) \qquad \frac{\Gamma[x : \sigma] \vdash_s M : \tau}{\Gamma \vdash_s \lambda x.M : \sigma \to \tau}$$

$$(APP) \qquad \frac{\Gamma \vdash_s M : \sigma \to \tau \quad \Gamma \vdash_s N : \sigma}{\Gamma \vdash_s M\ N : \tau}$$

$$(LET) \qquad \frac{\Gamma \vdash_s M : \sigma \quad [D(\theta^g) \subset \overline{\mathcal{V}}(\Gamma)] \quad \Gamma[x : \sigma\theta^g] \vdash_s N : \tau}{\Gamma \vdash_s \text{let } x = M \text{ in } N : \tau}$$

□

On étend les substitutions aux contextes de la façon suivante : l'assertion $(x : \sigma)\mu$ déclare x de type $\sigma\mu$. Le contexte $\Gamma\mu$ est la suite des substitutions des éléments de Γ par μ.

Lemme 3.4 *L'opération de substitution conserve la validité des jugements :*

$$(SUB) \qquad \frac{\Gamma \vdash_s^\ell M : \sigma}{\Gamma\mu \vdash_s^\ell M : \sigma\mu}$$

Remarque 3.2 Si u et v sont deux substitutions quelconques de domaines disjoints, alors il existe une substitution w de même domaine que u telle que $uv =_A vw$. (Il suffit de prendre $w =_A uv \restriction dom\ u$.)

Démonstration Par récurrence sur la longueur ℓ de la dérivation

Pour $\ell = 1$ la dérivation se réduit à l'axiome :

$$\frac{[\sigma_g = \Gamma(x)] \quad [\mu^g : \mathcal{V}^g(\sigma_g) \to \mathcal{T}]}{\Gamma \vdash_s^1 x : \sigma_g \mu^g} \quad (VAR)$$

Pour toute substitution μ, il est immédiat que $x : \sigma_g\mu$ est dans $\Gamma\mu$. D'autre part μ^g et μ ayant des domaines disjoints, il existe une substitution ν^g de même domaine que μ^g telle que $\mu^g\mu =_A \mu\nu^g$ (1). On peut donc appliquer l'axiome :

$$\frac{\dfrac{[\sigma_g\mu = \Gamma(x)] \quad [\nu^g : \mathcal{V}^g(\sigma_g\mu) \to \mathcal{T}]}{\Gamma \vdash_s^1 x : \sigma_g\mu\nu^g} \quad (VAR)}{\Gamma \vdash_s^1 x : \sigma_g\mu^g\mu} \quad (1)$$

Supposons la propriété vraie pour ℓ et le jugement $\Gamma \vdash_s^{\ell+1} M : \sigma$ valide (H_ℓ). Considérons une substitution μ quelconque. Nous montrons que le jugement $\Gamma\mu \vdash_s^{\ell+1} M : \sigma\mu$ est valide par cas sur la dernière règle de la dérivation du jugement précédent.

- Le cas (VAR) est impossible

- Cas (LET)

$$\frac{\Gamma \vdash_s^\ell M : \sigma \quad [D(\theta^g) \subset \overline{\mathcal{V}}(\Gamma)] \quad \Gamma[x : \sigma\theta^g] \vdash_s^\ell N : \tau}{\Gamma \vdash_s^{\ell+1} \mathit{let}\ x = M\ \mathit{in}\ N : \tau}$$

Considérons une bijection θ de même domaine que θ^g mais de domaine réciproque isolé de Γ, σ et de la substitution μ. La généralisation η^g égale à $\theta^{-1}\theta^g$ et la substitution ν égale à $\theta\mu$ vérifient :

(1) $D(\eta^g) \subset \overline{\mathcal{V}}(\Gamma)$ car Γ est isolé de θ^{-1}

(2) $\sigma\nu\eta^g =_A \sigma\theta^g\mu$

(3) $\Gamma\nu =_A \Gamma\mu$ car le domaine de θ égal à celui de θ^g est isolé de Γ.

Appliquons (H_ℓ) avec ν à la prémisse gauche et avec μ à la prémisse droite, puis la règle (LET) avec la généralisation η^g.

$$\frac{\dfrac{\dfrac{\Gamma \vdash_s^\ell M : \sigma}{\Gamma\nu \vdash_s^\ell M : \sigma\nu}\ (H_\ell)}{\Gamma\mu \vdash_s^\ell M : \sigma\nu}\ (3) \qquad [(1)] \qquad \dfrac{\dfrac{\Gamma[x : \sigma\theta^g] \vdash_s^\ell N : \tau}{\Gamma\mu[x : \sigma\theta^g\mu] \vdash_s^\ell N : \tau\mu}\ (H_\ell)}{\Gamma\mu[x : \sigma\nu\eta^g] \vdash_s^\ell N : \tau\mu}\ (2)}{\Gamma\mu \vdash_s^{\ell+1} \text{let } x = M \text{ in } N : \tau\mu}\ (LET)$$

- Les cas (APP) et (FUN) sont une conséquence directe de la distributivité de l'opération de substitution sur le symbole $\rightarrow$.

- Le cas $(EQUAL)$ est immédiat car les images par une substitution de deux types A-égaux sont deux types A-égaux.

■

Lemme 3.5 *Le jugement $\Gamma \vdash_g M : \sigma_g$ est dérivable dans le système (ML_g) si et seulement si pour toute instance simple σ de σ_g le jugement $\Gamma \vdash_s M : \sigma$ est dérivable dans (ML_s).*

Démonstration L'inclusion de (ML_s) dans (ML_g) est immédiate : il suffit de réécrire les règles $(EQUAL)$, (VAR) et (LET) de (ML_s) respectivement en fonction des règles $(INST)$, $(VAR)(INST)$ et $(GEN)(LET)$ de (ML_g). La réciproque se démontre par récurrence sur la taille des dérivations dans (ML_g).

Pour $\ell = 1$ la dérivation se réduit à l'axiome :

$$\frac{[\sigma_g = \Gamma(x)]}{\Gamma \vdash_g x : \sigma_g}\ (VAR)$$

Pour toute substitution $\mu^g : \mathcal{V}(\sigma_g) \rightarrow \mathcal{T}$, on peut appliquer l'axiome (VAR) avec la substitution μ^g.

Supposons que le jugement $\Gamma \vdash_g^{\ell+1} M : \sigma_g$ soit vrai (H_ℓ). Soit μ^g une substitution générique quelconque telle que $\sigma_g\mu^g \in \mathcal{T}$. Nous montrons que le jugement $\Gamma \vdash_s M : \sigma_g\mu^g$ est vrai par cas sur la dernière règle de la dérivation du jugement précédent.

- Le cas (VAR) est impossible.

- Cas $(INST)$

$$\frac{\Gamma \vdash_g^\ell M : \tau_g \quad [\tau_g \leq_A^g \rho_g]}{\Gamma \vdash_g^{\ell+1} M : \tau_g\nu^g}$$

Il existe donc une substitution ν^g telle que $\rho_g =_A \tau_g\nu^g$. Il suffit d'appliquer (H_ℓ), avec la substitution générique $\nu^g\mu^g$, puis la règle $(EQUAL)$.

- Cas (GEN)

$$\frac{\Gamma \vdash_g^\ell M : \tau_g \quad [D(\theta^g) \subset \overline{\mathcal{V}}(\Gamma)]}{\Gamma \vdash_g^{\ell+1} M : \tau_g\theta^g}$$

Comme θ^g et μ^g ont des domaines disjoints, il existe une substitution[1] μ de même domaine que θ^g telle que $\theta^g\mu^g$ soit égale à $\mu^g\mu$. Comme par hypothèse, $\mathcal{V}^g(\tau_g\theta^g\mu^g)$ est vide, il est clair que $\mu \restriction \mathcal{V}(\tau_g\theta^g)$ est non générique.

Il suffit d'appliquer (H_ℓ), avec la substitution générique μ^g, puis d'appliquer le lemme (SUB) avec la substitution $\mu \restriction \mathcal{V}(\tau_g\mu^g)$ dont le domaine n'intercepte pas $\mathcal{V}(\Gamma)$.

- Cas (LET)

$$\frac{\Gamma \vdash^\ell_g M : \tau_g \quad \Gamma[x : \tau_g] \vdash^\ell_g N : \tau}{\Gamma \vdash^\ell_g \mathit{let}\ x = M\ \mathit{in}\ N : \tau}$$

Il suffit d'appliquer (H_ℓ) à la prémisse gauche avec une substitution $\nu^g : \mathcal{V}^g(\tau_g) \to \overline{\mathcal{V}}(\Gamma)$ et à la prémisse droite avec la substitution identité.

- Les règles (APP) et (FUN) de (ML_g) sont également dans (ML_s).

■

En fait on peut se contenter de n'utiliser la règle $(EQUAL)$ qu'immédiatement après une règle (VAR) ou immédiatement avant l'une des deux branches d'une règle (APP). On aurait donc pu se dispenser de la règle $(EQUAL)$ en la fusionnant avec les règles (VAR) et (APP), de façon analogue à la règle (GEN). Mais la règle $(EQUAL)$ est beaucoup plus régulière et peut être prise en compte directement par les algorithmes d'unification. C'est ainsi mieux de la garder dans toute sa généralité, ce qui laisse plus de liberté pour les choix ultérieurs du contrôle.

3.5 Système d'inférence hiérarchique

Habituellement on dit que la présentation (ML_s) est dirigée par la syntaxe, ce qui signifie que pour chaque construction du langage ML permettant d'obtenir un terme M il existe une seule règle permettant de dériver un jugement de typage concernant M. Cette propriété est liée à la réversibilité du système d'inférence, et donc à la facilité d'en extraire un algorithme.

Nous voulons maintenant soulever un problème important concernant l'efficacité des algorithmes extraits qui a souvent été passé sous silence. Si l'on veut à partir d'un contexte Γ et un terme $\mathit{let}\ x = M\ \mathit{in}\ N$ trouver un type σ tel que l'on ait le jugement

$$\Gamma \vdash_g \mathit{let}\ x = M\ \mathit{in}\ N : \sigma,$$

il nous faut utiliser la règle (LET) et donc commencer par trouver un type τ tel que l'on ait le jugement

$$\Gamma \vdash_g M : \tau,$$

puis une substitution $D(\theta^g) \subset \overline{\mathcal{V}}(\Gamma)$ pour augmenter le contexte Γ avec l'assertion $x : \tau\theta^g$, etc. Tous les algorithmes d'inférence de ML procèdent de cette façon. Si l'on veut obtenir un typage le plus général, il faudra que θ^g généralise le plus de variables possibles, donc toutes les variables qui ne sont pas dans le contexte Γ. Un algorithme directement dérivé du système d'inférence ci-dessus ne peut faire mieux que de calculer ces variables à chaque généralisation. Or cette opération peut être très coûteuse, si le contexte Γ devient très grand.

En particulier, si le jugement $\Gamma \vdash_g M : \sigma$ est valide, alors quel que soit le contexte Γ_0, les deux problèmes de typage $\Gamma_0\Gamma \vdash_g M : \alpha$ et $\Gamma \vdash_g M : \alpha$ ont le même ensemble de solutions. Or le second ensemble sera plus coûteux à calculer, car le contexte Γ_0, bien qu'il soit passif, alourdira toutes les opérations de généralisation. Ceci n'est bien sûr pas souhaitable et on peut même espérer que ces deux calculs aient le même coût.

Pour être efficace, un algorithme doit donc maintenir une structure de donnée plus riche que celle des types génériques. Nous allons voir dans ce qui suit comment les types hiérarchiques permettent de résoudre le problème d'équité mentionné ci-dessus.

Nous utilisons les types hiérarchiques présentés dans le chapitre précédent. Nous rappelons que la théorie équationnelle est supposée régulière.

[1] Nous commettons un abus de notation, car une partie de μ peut être générique.

Définition 3.10 Le système d'inférence hiérarchique (ML_h) est l'ensemble des règles d'inférence suivantes formées sur des jugements de la forme $\Gamma \vdash_h M :_n \sigma$ où :

- Γ est un contexte de liaisons génériques de degré au plus n.
- M est un terme de ML_E.
- σ est un type de degré au plus n.

$$(EQUAL) \qquad \frac{\Gamma \vdash_h M :_n \sigma \quad [\sigma =_A \tau]}{\Gamma \vdash_h M :_n \tau}$$

$$(VAR) \qquad \frac{[\sigma_g = \Gamma(x)] \quad [\mu^g : \mathcal{V}^g(\sigma_g) \to \mathcal{T}_n]}{\Gamma \vdash_h x :_n \sigma_g \mu^g}$$

$$(FUN) \qquad \frac{\Gamma[x : \sigma] \vdash_h M :_n \tau}{\Gamma \vdash_h \lambda x.M :_n \tau}$$

$$(APP) \qquad \frac{\Gamma \vdash_h M :_n \sigma \to \tau \quad \Gamma \vdash_h N :_n \sigma}{\Gamma \vdash_h M\ N :_n \tau}$$

$$(LET) \qquad \frac{\Gamma \vdash_h M :_{n+1} \sigma \quad [\theta^g : \mathcal{V}^{n+1}(\sigma) \to \mathcal{V}^g] \quad \Gamma[x : \sigma\theta^g] \vdash_h N :_n \tau}{\Gamma \vdash_h \text{let } x = M \text{ in } N :_n \tau}$$

□

Lemme 3.6 *Soit Γ est un contexte et σ un type tous deux de degré au plus n. Le jugement $\Gamma \vdash_s M : \sigma$ est dérivable dans le système (ML_s) si et seulement si le jugement $\Gamma \vdash_h M :_n \sigma$ est dérivable dans (ML_h).*

Démonstration L'inclusion réciproque est immédiate car toute inférence dans (ML_h) est aussi une inférence dans (ML_s). En particulier pour la règle (LET) le fait que Γ soit de degré au plus n garantit que le domaine de θ^g et $\mathcal{V}(\Gamma)$ ont une intersection vide.

L'inclusion directe se montre par récurrence sur la longueur de la dérivation dans (ML_s). Si le jugement provient de l'axiome (VAR), les hypothèses sur les degrés de Γ et σ font que le jugement est également bien formé dans (ML_h) et donc l'axiome (VAR) de (ML_h) s'applique. Supposons que l'inclusion soit vraie pour toute dérivation de longueur l dans (ML_s), et qu'il existe une preuve de longueur $\ell + 1$ du jugement $\Gamma \vdash_s M : \sigma$ dans (ML_s). On montre par cas sur la structure de la dernière règle appliquée dans la dérivation du jugement $\Gamma \vdash_s^{\ell+1} M : \sigma$ dans (ML_s).

- Le cas (VAR) n'est pas possible.
- Si la dernière dérivation est une règle (FUN), les conditions de bonne formation dans (ML_h) de la conclusion garantissent que la prémisse est également bien formée ; on peut lui appliquer l'hypothèse de récurrence, puis la règle (FUN) de (ML_h).
- On peut appliquer le même raisonnement pour la règle $(EQUAL)$.
- Cas (APP)

  $$\frac{\Gamma \vdash_s^{\ell} M : \sigma \to \tau \quad \Gamma \vdash_s^{\ell} N : \sigma}{\Gamma \vdash_s^{\ell+1} M\ N : \tau}$$

  Le type σ peut être de degré supérieur à n. Mais les variables en cause n'apparaissent pas dans Γ. Une substitution quelconque de ces variables en des types de $\mathcal{T}_n$ transforme les prémisses en des jugements bien formés dans (ML_h). Ces jugements sont prouvables dans (ML_s) grâce au lemme (SUB). On peut alors appliquer l'hypothèse aux prémisses puis la règle (APP) de (ML_h).

- Cas (LET)

$$\frac{\Gamma \vdash_s^\ell M : \sigma \quad [D(\theta^g) \subset \overline{\mathcal{V}}(\Gamma)] \quad \Gamma[x : \sigma\theta^g] \vdash_s^\ell N : \tau}{\Gamma \vdash_s^{\ell+1} \text{let } x = M \text{ in } N : \tau}$$

Soit θ un affaiblissement renommant les variables de σ de degré supérieur à n qui ne sont pas dans le domaine de θ^g en des variables de $\mathcal{V}^n$ isolées de Γ et de τ. Soit η un renommage des variables de σ qui sont dans le domaine de θ^g en des variables de $\mathcal{V}^{n+1}$. Alors :

(1) La composition des renommages θ et η laisse Γ invariant,

(2) $D(\eta^{-1}\theta^g) \subset \mathcal{V}^n$ et $\sigma\theta\theta^g \in \mathcal{T}_n$,

(3) θ commute avec θ^g et laisse τ invariant.

donc on peut dériver :

$$\frac{\dfrac{\dfrac{\Gamma \vdash_s^\ell M : \sigma}{\Gamma \vdash_s^\ell M : \sigma\theta\eta}\,(SUB)(1)}{\Gamma \vdash_h M :_{n+1} \sigma\theta\eta}\,(H_\ell) \qquad [(2)] \qquad \dfrac{\dfrac{\Gamma[x : \sigma\theta^g] \vdash_s^\ell N : \tau}{\Gamma[x : \sigma\theta\theta^g] \vdash_s^\ell N : \tau}\,(SUB)(3)}{\Gamma[x : \sigma\theta\eta^g] \vdash_h N :_n \tau\theta}\,(H_\ell)}{\Gamma \vdash_h \text{let } x = M \text{ in } N :_n \tau}\,(LET)$$

■

Lemme 3.7 *Le système (ML_h) est stable par substitution admissible :*

$$(SUB) \qquad \frac{\Gamma \vdash_h^\ell M :_n \sigma}{\Gamma\mu \vdash_h^\ell M :_n \sigma\mu}$$

Démonstration La démonstration est la même que dans le cas du système (ML_s). La remarque 3.2 reste valable si l'on se restreint à des substitutions admissibles. Pour le cas (LET) on choisira bien sûr une substitution θ admissible de même degré que θ^g

Une autre démonstration est de considérer ce lemme comme un corollaire du lemme précédent[(1)]. En effet,

$$\frac{\dfrac{\dfrac{\Gamma \vdash_h M :_n \sigma}{\Gamma \vdash_s M : \sigma}\,(1)}{\Gamma\mu \vdash_s M : \sigma\mu}\,(SUB)}{\Gamma\mu \vdash_h M :_n \sigma\mu}\,(1)$$

On remarquera que la restriction à une substitution admissible n'est pas nécessaire, pourvu que les jugements soient bien formés. Mais bien sûr, cela n'a pas d'intérêt dans le système (ML_h). ■

Définition 3.11 L'ensemble des *instances hiérarchiques* de degré n d'un type générique σ_g est égal à

$$\{\sigma_g\mu^g \mid \mu^g : \mathcal{V}^g(\sigma_g) \to \mathcal{T}_n\}.$$

□

Lemme 3.8 *Le système (ML_h) est stable par enrichissement du contexte.*

$$(WEAK) \qquad \frac{\Gamma \vdash_h M :_n \sigma \quad [\Gamma' \leq_A^n \Gamma]}{\Gamma' \vdash_h M :_n \sigma}$$

Démonstration Soit ℓ la longueur commune aux contextes Γ et Γ'. Il existe une suite de substitutions génériques G de longueur ℓ tel que pour chaque entier n plus petit que ℓ le type σ_g de $\Gamma(n)$ s'obtienne par la substitution $G(n)$ à partir du type de $\Gamma'(n)$ à A-égalité près. Si l'élément n déclare x dans Γ (et Γ'), on note $G(x)$ au lieu de $G(n)$. On étend G sur $I\!N$ avec la substitution de domaine vide.

La dérivation obtenue en remplaçant simultanément toutes les applications de règles (VAR)

$$\frac{[\nu^g : \mathcal{V}^g(\Gamma\Delta(x)) \to \mathcal{T}_n]}{\Gamma\Delta \vdash_h N :_n \Gamma\Delta(x)\nu^g}\,(VAR)$$

par

$$\frac{[G(x)\nu^g : \mathcal{V}^g(\Gamma'\Delta(x)) \to \mathcal{T}_n]}{\Gamma'\Delta \vdash_h N :_n \Gamma'\Delta(x)G(x)\nu^g} \quad (VAR)$$
$$\frac{\Gamma'\Delta \vdash_h N :_n \Gamma'\Delta(x)G(x)\nu^g}{\Gamma'\Delta \vdash_h N :_n \Gamma\Delta(x)\nu^g} \quad (EQUAL)$$

et les contextes $\Gamma\Delta$ par les contextes $\Gamma'\Delta$ partout dans la dérivation de $\Gamma \vdash_h M :_n \sigma$ est une dérivation valide de $\Gamma' \vdash_h M :_n \sigma$. ■

Corollaire 3.9 *Si deux jugements sont A-équivalents, ils ont même valeur de vérité.*

Remarque 3.3 Nous pouvons transporter le lemme ci-dessus dans (ML_g) puis (ML_q). Il est remarquable que si le corollaire pouvait se démontrer facilement directement dans (ML_s) ou même (ML_g) car il n'y a pas d'apparition de variables libres, ce n'est plus du tout vrai du lemme $(WEAK)$. En particulier la démonstration ci-dessus ne serait pas correcte car il y pourrait y avoir capture de variables.

3.6 L'algorithme W pour le système hiérarchisé

L'algorithme W habituellement présenté est dérivé du système (ML_s). Le traitement de la généralisation dans cet algorithme l'empêche d'être efficace. La spécification (ML_s) suggère de parcourir l'environnement Γ au moment de généraliser pour y découvrir l'ensemble des variables libres. Or ce parcours qui dépend de la taille de Γ peut devenir très coûteux, et même prépondérant. Différentes astuces peuvent être utilisées pour réduire l'espace de recherche des variables libres dans Γ, en distinguant les variables éventuellement généralisables de celles qui ne le seront pas, mais ces optimisations ne sont que des approximations de la méthode que nous allons présenter et elles n'empêchent pas la généralisation de devenir prépondérante sur le temps de calcul. D'autre part, les astuces ne sont jamais justifiées très formellement.

L'intérêt du système (ML_h) est essentiellement d'introduire le calcul de la généralisation dans le système d'inférence lui-même, et de permettre ce calcul de façon incrémentale, et nous verrons de coût négligeable.

Une *équation de typage* est un quadruplet (Γ, M, n, σ), noté $\Gamma \Vdash M :_n \sigma$, telle que Γ et σ soit au plus de degré n. L'ensemble des variables d'une équation de typage $\Gamma \Vdash M :_n \sigma$ est l'ensemble $\mathcal{V}(\Gamma) \cup \mathcal{V}(\sigma)$. Nous nous plaçons dans la ϕ-A-théorie et nous prendrons donc $\mathcal{S}_a$ pour l'ensemble des substitutions non génériques ϕ-admissibles. L'ensemble des équations de typages muni des opération

1. $var\,(\Gamma \Vdash M :_n \sigma) = \mathcal{V}(\Gamma) \cup \mathcal{V}(\sigma)$
2. $sub\,((\Gamma \Vdash M :_n \sigma), \mu) = (\Gamma\mu \Vdash M :_n \sigma\mu)$
3. $val\,(\Gamma \Vdash M :_n \sigma) \iff (\Gamma \vdash_h M :_n \sigma)$.

est un ensemble d'unificande. Nous en faisons la conjonction avec les unificandes hiérarchisés.

Nous donnons ci-dessous des transformations d'A-extensions qui mettront en évidence un échec ou permettront de se ramener à des unificandes ne comportant plus que des multi-équations. Nous en déduirons un algorithme de résolution pour tout système d'équations de typage.

Proposition 3.10 *Soit $\Gamma \vdash_h x :_n \sigma$ une équation de typage protégée par W. Si $\Gamma(x)$ n'est pas défini, alors l'équation de typage précédente n'a pas de solution. Sinon pour tout renommage η^g des variables génériques de $\Gamma(x)$ en des variables de degré n prises en dehors de W,*

$$(\sigma \doteq \Gamma(x)\eta^g) \not\Rightarrow^W_A (\Gamma \Vdash x :_n \sigma).$$

Démonstration Si μ est une solution de $\sigma \doteq \Gamma(x)\eta^g$, il est immédiat par la règle (VAR) que μ est solution de $\Gamma \Vdash x :_n \sigma$. Réciproquement si μ satisfait $\Gamma\mu \vdash_h x :_n \sigma\mu$, il est nécessaire que $\Gamma(x)$ soit défini, et $\sigma\mu$ doit être obtenu par le remplacement de toutes les variables génériques de $\Gamma(x)$ en des termes de $\mathcal{T}^n$. Soit ν^g ce remplacement. Nous avons $\Gamma(x)\nu^g =_A \sigma\mu$. La substitution $(\eta^g)^{-1}\nu^g$ est de domaine en dehors de W. Sa somme avec μ est clairement une solution de $\sigma \doteq \Gamma(x)\eta^g$. ■

Proposition 3.11 *Soit Γ un contexte et σ un type de degré n tous deux protégés par W. Pour toutes variables α et β de degré n prises en dehors de W,*

$$(\Gamma[x:\alpha] \Vdash M :_n \beta) \sqcap (\alpha \to \beta \doteq \sigma) \equiv\!\!\rangle_A^W (\Gamma \Vdash \lambda x.M :_n \sigma)$$

$$(\Gamma \Vdash M :_n \alpha \sqcap \Gamma \Vdash N :_n \beta) \sqcap (\alpha \doteq \beta \to \sigma) \equiv\!\!\rangle_A^W (\Gamma \Vdash M\ N :_n \sigma)$$

Démonstration Une solution du membre gauche est clairement une solution du membre droit par les règles (FUN) et (APP) de (ML_h). Inversement si μ valide le jugement

$$\Gamma\mu \vdash_h \lambda x.M :_n \sigma\mu,$$

il doit exister un type τ telle que l'on puisse prouver

$$\Gamma\mu[x:\tau] \vdash_h M :_n \sigma\mu.$$

Alors la somme de μ avec $(\alpha \mapsto \tau)$ est une solution du membre gauche. Si μ valide le jugement

$$\Gamma\mu \vdash_h M\ N :_n \sigma\mu,$$

il doit exister un type τ tel que

$$\Gamma\mu \vdash_h M :_n \tau \to \sigma\mu \ \wedge\ \Gamma\mu \vdash_h N :_n \tau.$$

Alors la somme de μ avec $(\alpha \mapsto (\tau \to \sigma\mu), \beta \mapsto \tau)$ est solution du membre gauche. ■

Proposition 3.12 *Soit l'équation de typage $\Gamma \Vdash$ let $x = M$ in $N :_n \sigma$ protégée par W et α une variable de degré $n+1$ prise en dehors de W. Soit $\mathcal{M} \sqcap \mathcal{N}$ une conjonction d'équilibre au degré $n+1$ de l'équation de typage $\Gamma \Vdash M :_{n+1} \alpha$ en dehors de W. Si $\mathcal{N}$ n'a pas de solution, alors l'équation de typage $\Gamma \Vdash$ let $x = M$ in $N :_n \sigma$ n'a pas de solution.*

Sinon, il existe une pré-solution $[\mathcal{N}]$ de $\mathcal{N}$. Si $\theta^g : \mathcal{V}^{n+1}(\alpha[\mathcal{N}]) \to \mathcal{V}^g$, alors

$$\mathcal{M} \sqcap (\Gamma[x:\alpha[\mathcal{N}]\theta^g] \Vdash N :_n \sigma) \equiv\!\!\rangle_A^W (\Gamma \Vdash \text{let } x = M \text{ in } N :_n \sigma)$$

Démonstration Comme les variables de $\mathcal{N}$ sont de degré $n+1$, il existe un renommage admissible θ tel que $[\mathcal{N}]\theta$ soit un unificateur principal de $\mathcal{N}$ (lemme 2.23). Il existe également un unificateur principal μ de $\mathcal{M}$ tel que $[\mathcal{N}]\theta\mu$ soit un unificateur principal de $\mathcal{N} \sqcap \mathcal{M}$ (proposition 2.24).

Une solution de

$$\mathcal{M} \sqcap (\Gamma[x:\alpha[\mathcal{N}]\theta^g] \Vdash N :_n \sigma)$$

est de la forme $\mu\nu$. Ainsi, $[\mathcal{N}]\theta\mu\nu$ est une solution de $\mathcal{N} \sqcap \mathcal{M}$ donc également une solution de l'équation $\Gamma \Vdash M :_{n+1} \alpha$. Comme Γ et σ sont dans $\mathcal{T}_n$, il ne sont pas modifiés par $[\mathcal{N}]\theta$. De façon analogue, le domaine de $\mu\nu$ est inclus dans $\mathcal{V}_n$, aussi $\mathcal{V}^{n+1}(\alpha[\mathcal{N}]\theta\mu\nu)$ est égal à $\mathcal{V}^{n+1}(\alpha[\mathcal{N}]\theta)$. On a donc la dérivation

$$\frac{\Gamma\mu\nu \vdash_h M :_{n+1} \alpha[\mathcal{N}]\theta\mu\nu \quad [\theta^{-1}\theta^g : \mathcal{V}^{n+1}(\alpha[\mathcal{N}]\theta\mu\nu) \to \mathcal{V}^g] \quad \Gamma\mu\nu[x:\alpha[\mathcal{N}]\theta\mu\nu\theta^{-1}\theta^g] \vdash_h N :_n \sigma\mu\nu}{\Gamma\mu\nu \vdash_h \text{let } x = M \text{ in } N :_n \sigma\mu\nu}$$

Ce qui montre que μ est aussi une solution de l'équation $\Gamma \Vdash$ let $x = M$ in $N :_n \sigma$.

Inversement, soit ν une solution de l'équation de typage

$$\Gamma \Vdash \text{let } x = M \text{ in } N :_n \sigma$$

de domaine inclus dans $\mathcal{T}_n$. Le jugement $\Gamma\nu \vdash_h$ let $x = M$ in $N :_n \sigma\nu$ n'a pu être dérivé que par une règle (LET) de (ML_h). Il existe donc un type τ et une généralisation $\eta^g : \mathcal{V}^{n+1}(\tau) \to \mathcal{V}^g$ tels que

$$\Gamma\nu \vdash_h M :_{n+1} \tau \quad \wedge \quad \Gamma\nu[x:\tau\eta^g] \vdash_h N :_n \sigma\nu.$$

La substitution $\nu + (\alpha \mapsto \tau)$ est une solution du système

$$\Gamma \Vdash M :_{n+1} \alpha$$

Elle peut être étendue en dehors de $W \sqcup \{\alpha\}$ en une solution ν' de $\mathcal{N} \sqcap \mathcal{M}$, moins générale qu'une solution principale $[\mathcal{N}]\theta$ de $\mathcal{N}$. En particularisant cette inégalité à la variable α, il apparaît que τ est moins général que $\alpha[\mathcal{N}]\theta$. Cette inégalité est conservée par la généralisation des variables de degré $n+1$.

En effet τ est égal à $\alpha[\mathcal{N}]\theta\nu'$, et sa généralisation par η^g à $\alpha[\mathcal{N}]\theta\nu'\eta^g$ que l'on veut comparer à $\alpha[\mathcal{N}]\theta^g$. Il suffit alors de comparer les substitutions $\nu'\eta^g$ et $\theta^{-1}\theta^g$. Cette dernière substitution généralise exactement toutes les variables de degré $n+1$ de $\alpha[\mathcal{N}]$. La précédente substitue éventuellement des variables de degré quelconque puis généralise dans le terme obtenu toutes les variables de degré $n+1$, c'est-à-dire des variables qui étaient déjà de degré $n+1$ dans $\alpha[\mathcal{N}]$ ou qui proviennent de la substitution par ν' de variables de degré $n+1$.

On peut donc appliquer le lemme 3.8 ($WEAK$) au jugement

$$\Gamma\nu[x : \tau\eta^g] \vdash_h N :_n \sigma\nu,$$

et en déduire

$$\Gamma\nu[x : \alpha[\mathcal{N}]\theta^g] \vdash_h N :_n \sigma\nu,$$

ce qui montre que ν' est aussi une solution de

$$\Gamma[x : \alpha[\mathcal{N}]\theta^g] \Vdash N :_n \sigma.$$

■

Définition 3.12 Nous appelons VAR, FUN, APP et LET-réductions les quatre A-extensions précédentes dans leur ordre respectif d'apparition. □

Toutes les réductions font décroître la somme des tailles des expressions du langage ML dans le système d'équations de typage ou le nombre d'équations de typage. Le processus de réduction combiné au processus de simplification des multi-équations termine donc toujours si le processus de simplification seul termine.

Remarque 3.4 La règle LET-réduction telle qu'elle est écrite mémorise dans l'environnement la pré-solution $x : \alpha[\mathcal{M}]\theta^g$ perdant ainsi le partage contenu dans le système $\mathcal{M}$. La seule utilisation ultérieure possible de cet environnement est par la VAR-réduction d'une équation de typage $\Gamma \Vdash x :_n \sigma$, qui prendra une copie du type $\alpha[\mathcal{M}]\theta^g$ par une substitution $\eta^g : D(\theta^g) \to \mathcal{V}$ et produira l'équation $\sigma \doteq \alpha[\mathcal{M}\theta^g\eta^g]$. Celle-ci est équivalente au système $\sigma \doteq \alpha \sqcap \mathcal{M}\theta^g\eta^g$ que l'on aurait pu obtenir directement en considérant qu'une assertion de typage est un quadruplet $x : \alpha, \mathcal{M}, \theta^g$ et en remplaçant les VAR et LET-réductions par

$$(\sigma \doteq (fst \circ \Gamma(x)) \sqcap (snd \circ \Gamma(x))\eta^g) \Rrightarrow^W_A (\Gamma \Vdash x :_n \sigma),$$

et

$$\mathcal{N} \sqcap (\Gamma[x : \alpha, [\mathcal{M}], \theta^g] \Vdash N :_n \sigma) \Rrightarrow^W_A e \sqcap (\Gamma \Vdash \mathit{let}\ x = M\ \mathit{in}\ N :_n \sigma).$$

On peut même remplacer $\mathcal{N}$ par le sous-système des multi-équations liées à α, qui en général sera beaucoup plus petit, après avoir vérifié qu'il admettait au moins une solution. C'est cette méthode qui est utilisée dans l'algorithme donné en annexe.

Remarque 3.5 Les règles de réduction ci-dessus peuvent être utilisées dans un ordre quelconque. En particulier, on peut réduire plusieurs sous-expressions simultanément y compris des liaisons. La seule restriction est que pour la réduction d'une liaison $\mathit{let}\ x = M\ \mathit{in}\ N$, dans un système $\mathcal{M}$, on ne peut ajouter les nouvelles équations de typage crées par la réduction de M à $\mathcal{M}$ que lorsque l'on est sûr que leur degré est inférieur au degré de la liaison.

Remarque 3.6 Les degrés sont des entiers. On aurait pu utiliser un ensemble de degrés plus structuré. La seule condition est que cet ensemble possède un inf et soit suffisamment grand. Un cas particulier est de prendre l'ensemble des occurrences des nœuds *let* dans l'expression de départ ordonné naturellement par l'ordre préfixe.

Théorème 5 *Si la théorie A est régulière, et si l'unification y est décidable et unitaire, alors étant donné un contexte générique Γ, un terme M, et une variable de type α, si le problème de typage $\Gamma \vdash_g M : \alpha$ admet une solution, alors il admet une solution principale.*

Théorème 6 *Dans les conditions précédentes, il existe un algorithme qui calcule l'existence d'une solution et lorsque c'est possible une solution principale au problème de typage précédent.*

Démonstration Il suffit de combiner l'ensemble des résultats précédents. ■

Nous avons énoncé les deux théorèmes précédents dans le système (ML_g) car l'énoncé y est le plus simple, mais on peut évidemment les transposer dans chacun des systèmes étudiés.

L'algorithme mentionné n'est pas complètement spécifié. En particulier, nous n'avons pas précisé l'ordre des transformations, ni le choix des structures de données. Nous détaillons brièvement ces points dans la partie suivante et donnons une implantation d'un algorithme dans le langage CAML en annexe.

3.7 Une bonne stratégie

Dans cette partie nous décrivons et justifions la stratégie qui a été retenue pour l'implantation de l'algorithme donné en annexe. Cette partie n'a qu'un intérêt pratique. En effet, la complexité de la synthèse de type pour le langage ML a été récemment étudiée par Paris Kanellakis et John C. Mitchell qui montrent dans [31] que sa complexité est au plus exponentielle en temps si l'on ne compte pas l'impression du résultat, mais ne donnent qu'une borne inférieure plus faible. Dans [35], A.J. Kfoury, J. Tiuryn et P. Urzyczyn et indépendamment [41] annoncent que la borne supérieure est effectivement atteinte.

L'efficacité que nous allons discuter ci-dessous est négligeable si l'on étudie le cas le pire. Mais d'un point de vue pratique, il s'avère que le cas le pire n'apparaît pas naturellement. Cette séparation entre le résultat théorique et l'apparente efficacité pratique est très bien discutée dans [31].

Le cas défavorable ne peut apparaître que dans un programme imbriquant les définitions locales de la façon suivante

$$let\ x = let\ x' = \ldots N\ in\ M\ in\ M'.$$

Remarquons qu'une preuve qu'un tel programme est bien typé dans le système (ML_h) nécessite d'utiliser des termes d'un degré au moins égal à la profondeur d'imbrication des $let\ \ldots\ in$. Il se trouve en pratique que ce niveau d'imbrication n'est jamais très élevé.

Au contraire un programme ML est plutôt composé de constructions $let\ \ldots\ in\ \ldots$ juxtaposées. Le meilleur exemple est une suite de définitions globales,

$$\begin{array}{l} let\ x_1 = M_1\ ;; \\ let\ x_2 = M_2\ ;; \\ \ldots \\ let\ x_n = M_n\ ;; \\ N \end{array}$$

qui se comporte exactement comme une longue chaîne de déclarations locales juxtaposées. D'ailleurs il arrive parfois que l'on effectue effectivement cette transformation pour cacher des identificateurs auxquels on ne veut pas donner libre accès à l'utilisateur.

$$\begin{array}{l} let\ x_1 = M_1\ in \\ let\ x_2 = M_2\ in \\ \ldots \\ let\ x_n = M_n\ in \\ N \end{array}$$

Il est donc important que l'algorithme de synthèse de type ait un bon comportement sur cette dernière configuration, la taille du programme pouvant devenir très importante. Bien que de moindre intérêt, on peut aussi envisager la configuration

$$\begin{array}{l} let\ x_n = \\ \quad \ldots \\ \qquad let\ x_2 = \\ \qquad\quad let\ x_1 = M_1\ in \\ \qquad\quad M_2\ in \\ \qquad \ldots \\ \quad M_n\ in \\ N \end{array}$$

Plus généralement, les coûts de la synthèse de type pour les deux programmes

$$P\big[Q\,[let\ x = M\ in\ N]\big], \qquad\qquad P\big[let\ x = M\ in\ Q[N]\big].$$

sont identiques lorsque x n'apparaît pas dans Q, et z n'apparaît qu'une seule fois dans $P[z]$ et dans $Q[z]$.

Enfin, nous ne pouvons pas ne pas mentionner les deux exemples pathologiques suivants, qui ont été remarqués simultanément par P. Buneman, P. Kanellakis, P. O'Keefe, J. C. Mitchell, A. Ohori, M. Wand et mais étaient sans doute connus par bien d'autres personnes. Le premier

Exemple 3.7

$$\begin{array}{l} let\ x_0 = \lambda x.x\,;; \\ let\ x_1 = x_0, x_0\,;; \\ let\ x_2 = x_1, x_1\,;; \\ \vdots \\ let\ x_n = x_{n-1}, x_{n-1}\,;; \\ x_n \end{array}$$

a un type principal de taille 2^n, représentable par un arbre dont toutes les feuilles sont des variables distinctes. Le second

Exemple 3.8

$$\begin{array}{l} let\ x_0\ y = \lambda z.zyy\,;; \\ let\ x_1\ y = y(y\ x_0)\,;; \\ let\ x_2\ y = y(y\ x_1)\,;; \\ \vdots \\ let\ x_n\ y = y(y\ x_{n-1})\,;; \\ x_n \end{array}$$

a un type principal de taille 2^{2^n}, mais représentable par un graphe orienté de taille 2^n. Notre algorithme trouve chacun des types ci-dessus en un temps proportionnel à 2^{cn}, sans compter les temps d'impression[2]. La plupart des implantations réelles sont très loin de cette performance, soit parce qu'elles n'effectue pas un partage maximal dans la représentation des types, soit parce qu'elles réalisent de trop nombreux parcourts de la structure des données, par exemple en effectuant un test d'occurrence à chaque substitution, ou en recherchant les variables généralisables dans un ensemble beaucoup trop grand.

A titre d'exemple, nous considérerons les expressions

$$\begin{array}{l} let\ r = \quad let\ x_0\ y = \lambda z.zyy\,;; \\ \quad let\ x_1\ y = y(y\ x_0)\,;; \\ \quad let\ x_2\ y = y(y\ x_1)\,;; \\ \quad \vdots \\ \quad let\ x_n\ y = y(y\ x_{n-1})\,;; \\ \quad x_n\,;; \\ () \end{array}$$

qui est une variante du dernier exemple évitant l'impression d'un résultat gigantesque. Nos mesures sont faites sur un SUN 3/60 avec 8 méga-octets de mémoire centrale. Il est mentionné dans [41] que pour $n = 5$, le langage *Standard ML of New Jersey, Version 0.33* fait croître sa mémoire jusqu'à presque 60 méga-octets dans la version qui imprime son résultat. Nous n'avons pas pu vérifier ce comportement sur SUN 3/60, car au bout d'un temps de calcul utilisateur de $251s$ le système d'exploitation ne peut plus allouer de mémoire; le processus avait alors une taille de 18.4 méga-octets. Sur un SUN 3/280 il faut $447s$ et 80 méga-octets de mémoire. Toujours pour $n = 5$ le langage

[2]Ceux-ci dépendent énormément du formatage des expressions. Ils dépendent également de la représentation choisie, et peuvent toujours être effectués en un temps proportionnel à la taille en mémoire de la structure qui les représente, simplement en imprimant cette mémoire.

CAML (version V2-6) avec une image mémoire de 8 méga-octets effectue le calcul en un temps utilisateur de $232s$ mesuré par le système d'exploitation.

Notre maquette effectue le calcul en un temps si faible que le temps mesuré par le système d'exploitation égal[3] à $0.7s$ n'est pas significatif, car il est essentiellement dû au chargement de l'image mémoire. Nous donnons des résultats plus précis mesurés par le système CAML en décomptant le temps passé à glaner les cellules. L'image mémoire fait 4 méga-octets.

n	5	6	7	8	9	10
temps	0.38s	0.61s	1.26s	2.08s	3.73s	7.36s

Pour $n = 10$ le temps total y compris le temps de glanage mesuré par CAML est de $25.3s$, celui donné par le système d'exploitation est de $28.8s$, et comprend le chargement de l'image mémoire. La régularité de la suite confirme que la complexité pratique est égale à la complexité théorique de 2^{cn}. Ce n'est certes pas le cas pour SML qui calcule le cas $n = 4$ en[4] $1.9s$ seulement !

La comparaison précédente ne change aucunement les questions soulevées par la complexité de l'algorithme de typage de ML, mais elle souligne simplement qu'une bonne implémentation ne fait pas exploser la machine avec un programme de cinq lignes : il en faut au moins le double !

Le test ci-dessus est éliminatoire, mais ne permet pas de certifier un synthétiseur de types. Il détecte d'abord un test de cyclicité mal placé, ou une perte de partage, mais a assez peu de chances de détecter l'efficacité de la généralisation, car les liaisons λ n'apparaissent qu'aux feuilles des expressions.

3.7.1 Contrôle

Nous donnons le contrôle sur lequel nous raisonnerons[5] par une fonction récursive *type* (algorithme 3.1). Elle A-étend un problème de typage $\Gamma \Vdash M :_0 \sigma$ en un système de multi-équations simplifié $\mathcal{M}$. Cet algorithme utilise deux fonctions auxiliaires, $simplify^n$ qui transforme un système par fusion-décomposition-mutation des multi-équations de bornes au moins égales à n, en un système complètement décomposé, et $equilibre^n$ qui transforme un système en une conjonction d'équilibre au degré n. Chacune de ces fonctions peut échouer, auquel cas le problème de typage initial n'a pas de solution.

La correction de l'algorithme est garantie par les règles de réduction, et la terminaison est évidente car à chaque appel récursif la taille de l'expression a diminué.

3.7.2 Analyse de l'algorithme

Les traitements d'une abstraction et d'une application sont de coût égaux à la somme des coûts de leurs sous-termes augmentée d'une constante. Les deux cas qui peuvent devenir coûteux sont ceux d'une variable et d'une définition locale. Nous considérons que $\Gamma(x)$ se calcule en temps constant. Le calcul de *type* revient au calcul de l'instance $\Gamma(x)\theta^g$. Il dépend linéairement de la taille de $\Gamma(x)$. En utilisant une structure de données adéquate, on peut le rendre linéaire par rapport à la taille du squelette générique[6]. Comme cas particulier, une instance d'un type non générique se calcule en temps constant. Le calcul de *type* pour une définition locale *let* $x = M$ *in* N se décompose en quatre étapes.

On effectue d'abord le calcul de *type* pour l'expression interne M dont le coût dépend de la taille des squelettes génériques des assertions liant les variables de M dans l'environnement Γ. Le nombre de variables introduites pendant ce calcul en dépend également de la même façon. Le système obtenu est ensuite simplifié au degré $n + 1$.

Dans une théorie équationnelle vide, et avec de bonnes structures de données, ce calcul peut être rendu proportionnel au nombre de variables introduites dans la phase précédente[7]. La plupart des algorithmes implantés sont en $n \log n$ dans le cas le pire, mais bien souvent quasi-linéaires en pratique. Pour une théorie équationnelle non vide, il dépend de la théorie, et peut devenir très

[3] Pour une image mémoire de 4 méga-octets.

[4] Le temps est mesuré par le système d'exploitation et comprend le chargement de l'image mémoire.

[5] Une implantation complète en CAML est donnée en annexe.

[6] Un type générique provient du renommage des variables de degré $n + 1$ dans une conjonction d'équilibre $\mathcal{M} \sqcap \mathcal{N}$ de degré n.

[7] Ce calcul est équivalent à un problème d'unification et peut être effectué en temps linéaire[51].

```
let rec type (W, M, Γ ⊩⊢ M :_n σ) =
 match M
  with << x >> ->
        let θ^g : V^g(Γ(x)) → V^n \ W in
        (W ∪ I(θ^g), M ⊓ (σ ≐ Γ(x)θ^g)

     | << λx.M >> ->
        let α ∈ V^n \ W in
        let β ∈ V^n \ (W ∪ {α}) in
        type (W ∪ {α, β}, M ⊓ (α → β ≐ σ), Γ[x : α] ⊩⊢ Mβ)

     | << MN >> ->
        let α ∈ V^n \ W in
        let β ∈ V^n \ (W ∪ {α}) in
        let W', U' = type (W ∪ {α, β}, M ⊓ (α ≐ β → σ), Γ ⊩⊢ M :_n α) in
        type (W', U', Γ ⊩⊢ N :_n β)

     | << let x = M in N >> ->
        let α ∈ ¬W in
        let W', M' = type (W ∪ {α}, M, Γ ⊩⊢ M :_{n+1} α in
        let N ⊓ N' = (equilibre^{n+1} ∘ simplify^{n+1})(M') in
        let θ^g : V^{n+1}(α[N']) in
        type (W', N, Γ[X : α[N']θ^g] ⊩⊢ N :_n σ) in

function << Γ ⊩⊢ M :_0 σ >> ->
    (simplify ∘ snd ∘ type) (V(σ), ∅, Γ ⊩⊢ M :_0 σ)
; ;
```

Algorithme 3.1: type

supérieur car de nouvelles variables ou pire des disjonctions de systèmes peuvent être introduites pendant la mutation. L'équilibrage au degré $n+1$ est de coût proportionnel au nombre total de variables de degré $n+1$, donc comparable au coût de la simplification. Enfin, on calcule le typage de N dans l'environnement étendu.

Le surcoût dû à l'équilibrage est donc au pire comparable au coût des autres calculs, mais il n'augmente en aucun cas la complexité des calculs.

3.8 Extensions du langage ML

Nous avons montré dans ce chapitre que l'algorithme de typage du langage ML s'étend au cas où l'ensemble des types est muni d'une théorie équationnelle régulière avec sortes. Nous appellerons $\mathcal{K}$-A-ML une telle extension ou simplement $\mathcal{K}$-ML lorsque la théorie équationnelle est vide. Les théorèmes 5 et 6 se factorisent dans le théorème suivant.

Théorème 7 *Si la théorie A est régulière et l'unification y est décidable, alors le typage dans le langage $\mathcal{K}$-A-ML est décidable. Si de plus l'unification dans la théorie A est unitaire, alors une expression typable admet un type principal.*

Dans le chapitre suivant nous coderons les objets enregistrements dans le langage $\mathcal{K}$-A-ML d'abord dans une théorie vide, puis dans une théorie non vide lorsque les enregistrements seront de tailles non bornées.

Chapitre 4

Typage des objets enregistrements

4.1 Une solution simple pour des enregistrements de taille bornée

Dans cette partie, nous allons montrer comment les objets enregistrements peuvent être codés très simplement dans l'extension $\mathcal{K}$-ML du langage ML. Nous découvrirons d'abord le codage de façon intuitive, puis nous l'étudierons plus formellement. Nous le généraliserons dans une prochaine partie à des enregistrements de taille non bornée. Nous ne recherchons donc pas dans cette partie la puissance du codage ni son efficacité, mais simplement à en comprendre sa simplicité et sa généralité.

Nous nous donnons un ensemble fini d'étiquettes $\mathcal{L}$.

4.1.1 Une approche intuitive

Avertissement Dans cette partie nous resterons très informel.

Si nous nous laissons guider par la sémantique intuitive des enregistrements, ce sont des fonctions partielles de l'ensemble des étiquettes vers l'ensemble des valeurs. Nous étudions le cas plus simple où il n'y a que deux étiquettes, et une valeur unique, notée $\bullet$, de type *présent*[1]. Nous pouvons alors représenter les enregistrements dans des boîtes à deux cases.

Exemple 4.1 L'enregistrement $\boxed{\bullet}\boxed{\ }$ associe la valeur $\bullet$ à la première étiquette et n'est pas défini sur la seconde.

Supposons que nous puissions définir les objets

$$X = \boxed{\bullet}\boxed{\ } \qquad\qquad Y = \boxed{\bullet}\boxed{\bullet}$$

en ML. Comment les typer ? Il est toujours possible de leur donner deux nouveaux types incompatibles. Mais alors nous ne pourrions pas les mélanger, par exemple dans les deux branches d'une construction *if* ... *then* ... *else* ..., ni les passer en arguments à une même fonction. Une première solution serait de leur donner plusieurs valeurs, par exemple dire que X et Y représentent les ensembles de valeurs :

$$X = \boxed{\ }\boxed{\ } \mid \boxed{\bullet}\boxed{\ } \qquad\qquad Y = \boxed{\ }\boxed{\ } \mid \boxed{\bullet}\boxed{\ } \mid \boxed{\ }\boxed{\bullet} \mid \boxed{\bullet}\boxed{\bullet}.$$

Cela a l'avantage de permettre à un enregistrement ayant plus de champs d'être utilisé à la place d'un enregistrement ayant moins de champs. L'inconvénient est que l'introduction de valeurs multiples, qui est une forme de surcharge, n'est pas disponible dans le langage ML. D'autre part elle est souvent très coûteuse. La seule solution consiste donc à donner à X et Y des types compatibles. C'est-à-dire qu'il faut que la case pleine soit compatible avec la case vide. Il suffit simplement de remplir les cases vides avec une nouvelle constante $\circ$ d'un nouveau type, *absent* par exemple[2], qui mentionne explicitement que l'enregistrement n'est pas défini sur cette case. Ainsi

$$X = \boxed{\bullet}\boxed{\circ} \qquad\qquad Y = \boxed{\bullet}\boxed{\bullet},$$

[1] On pourra lire **noir** de type **présent**.
[2] On pourra **blanc** de type **absent**.

et l'on peut typer ces valeurs par

$$X : \Pi(\textit{présent}, \textit{absent}) \qquad\qquad Y : \Pi(\textit{présent}, \textit{présent}),$$

où Π est un nouveau symbole de type permettant de construire des types enregistrements. Malheureusement Y ne peut pas encore être utilisé à la place de X. Pour cela il faudrait que $\bullet$ soit aussi de type *absent*. Soit ! décidons que $\bullet$ est à la fois de type *absent* et de type *présent*, c'est-à-dire de type principal ε, où ε ne peut prendre ses instances que sur les types *présent* et *absent*.

$$X : \Pi(\varepsilon, \textit{absent}) \qquad\qquad Y : \Pi(\varepsilon, \varepsilon').$$

Lorsque l'on accède à une composante, il faut s'assurer que celle-ci est définie, c'est-à-dire que la valeur de sa case est $\bullet$. Il suffit que le type de sa case soit *présent*. Les projections fst et snd ont les types suivants :

$$fst : \Pi(\textit{présent}, \varepsilon) \to \textit{présent} \qquad\qquad snd : \Pi(\varepsilon, \textit{présent}) \to \textit{présent}.$$

On vérifie aisément que fst peut être appliqué à X et Y mais que snd ne peut être appliqué qu'à Y. Plutôt que des valeurs ordinaires, *présent* et *absent* sont des drapeaux indiquant si la case qui les contient est remplie ou non. On représentera désormais les boîtes par

$$X = \boxed{\bullet \;|\; \circ} \qquad\qquad Y = \boxed{\bullet \;|\; \bullet}.$$

Les cases contiennent maintenant de la place pour une vraie valeur. Par exemple,

$$X = \boxed{\bullet\ 1 \;|\; \circ} \qquad\qquad Y = \boxed{\bullet\ 2 \;|\; \bullet\ V},$$

mais avec quelle valeur doit-on remplir une case dont le drapeau est $\circ$? En fait une telle case ne sera jamais lue, donc peu importe son contenu. Nous utiliserons une nouvelle valeur Ω de type α.

$$X = \boxed{\bullet\ 1 \;|\; \circ\ \Omega} \qquad\qquad Y = \boxed{\bullet\ 2 \;|\; \bullet\ V},$$

que nous typons par

$$X : \Pi(\varepsilon.num, \textit{absent}.\alpha) \qquad\qquad Y : \Pi(\varepsilon.num, \varepsilon'.void)$$

où "." est un symbole infixé d'arité deux. Il est clair que ε et ε' parcourent les types *présent* et *absent* des drapeaux $\bullet$ et $\circ$, alors que α parcourt les types des autres valeurs. On peut facilement obtenir cet effet en déclarant les symboles *présent* et *absent* d'une sorte différente de celle des autres symboles. Maintenant, les projections sont typées par

$$fst : \Pi(\textit{présent}.\alpha, \varepsilon.\Omega) \to \alpha \qquad\qquad snd : \Pi(\varepsilon.\Omega, \textit{présent}.\alpha) \to \alpha,$$

et tout se combine merveilleusement bien.

Quand les étiquettes sont un peu plus nombreuses, mais toujours dénombrables, les enregistrement peuvent toujours être considérés comme de grandes boîtes filiformes ayant autant de cases qu'il y a d'étiquettes. Les étiquettes sont simplement un moyen syntaxique de décrire les champs significatifs. Par exemple, si les étiquettes sont tous les mots composés d'au plus six lettres, $\{A = V\}$ est simplement une notation pour la boîte représentant

$$\{A = \bullet, V\,;\ \ B = \circ, \Omega\,;\ \ C = \circ, \Omega\,;\ \ \ldots zzzzzz = \circ, \Omega\},$$

dont le type est

$$\Pi(\varepsilon.void, \textit{absent}.\beta_1, \textit{absent}.\beta_2, \ldots \textit{absent}.\beta_{642544811}),$$

dont la taille ne doit pas nous effrayer pour l'instant. Nous montrerons dans une prochaine partie comment nous pouvons, comme pour les poupées russes, ranger toutes les cases vides dans une seule.

4.1.2 Une formulation dans le système $\mathcal{K}$-ML

Nous donnons dans ce paragraphe une formulation des idées précédentes. Rappelons qu'il s'agit d'étudier le typage des expressions et non leur évaluation. C'est-à-dire que nous ne cherchons pas à justifier la sémantique intuitive utilisée dans l'approche précédente, mais simplement à définir de façon précise le système de typage et étudier ses propriétés syntaxiques. Nous supposerons donc données un certain nombre de primitives permettant d'effectuer les opérations élémentaires suivantes sur les enregistrements :

- construire un enregistrement vide (défini nulle part),
- étendre un enregistrement par un seul champ. Si ce champ existait dans l'enregistrement de départ, il est masqué par le nouveau champ.
- lire un champ d'un enregistrement,
- copier un enregistrement en effaçant un champ.

Nous ne résoudrons pas le cas suivant,

- étendre un enregistrement par tous les champs pris dans un second enregistrement. Comme dans l'extension simple, les champs du second enregistrement cachent ceux du premier s'ils existaient déjà,

mais le mentionnerons seulement dans des commentaires. Nous considérerons que ces opérations sont respectivement implantées par les primitives *null*, new^ℓ, $extract^\ell$ et $forget^\ell$ où ℓ désigne le champ correspondant à l'opération et parcourt donc l'ensemble $\mathcal{L}$. L'ensemble $\mathcal{L}$ est supposé fini et nous le prendrons égal à un segment $[1, L]$ de $I\!N$ pour simplifier.

Nous rappelons que le langage de typage de ML est l'algèbre libre $\mathcal{T}_0(\mathcal{C}_0, \mathcal{V}_0)$ simplement graduée. Pour le noyau de ML, l'ensemble $\mathcal{C}_0$ est réduit au seul élément $\rightarrow$. Soit $\mathcal{K}$ l'ensemble formé des trois sortes *usual*, *field* et *flag*. Soit $\mathcal{C}$ l'ensemble des symboles signés sur $\mathcal{K}$ suivants :

$$\begin{array}{lcl} \rightarrow & : & usual \otimes usual \Rightarrow usual \\ \Pi & : & \underbrace{field \otimes \ldots field}_{Card\,(\mathcal{L})} \Rightarrow usual \\ . & : & flag \otimes usual \Rightarrow field \\ présent & : & flag \\ absent & : & flag \end{array}$$

complété par les symboles

$$f \in \mathcal{C}_0 \setminus \{\rightarrow\} : \underbrace{usual \otimes \ldots usual}_{\varrho(f)} \Rightarrow usual$$

Soit $\mathcal{V}$ un $\mathcal{K}$-ensemble de variables tel que $\varsigma^{(-1)}(usual)$ soit égal à $\mathcal{V}_0$. Notre langage de types sera l'algèbre libre $\mathcal{T}(\mathcal{C}, \mathcal{V})$ signée sur $\mathcal{K}$. Le langage $\mathcal{T}_0$ peut être plongé dans l'ensemble des types de $\mathcal{T}$ de sorte *usual* de façon évidente.

Notation Bien que le lecteur puisse retrouver les sortes à partir des utilisations des variables, nous l'aiderons en notant ε et ε' les variables de la sorte *flag*, et ϕ et ψ les variables de la sorte *field*. Nous garderons les notations α, β et γ pour des variables de la sorte *usual* et σ, τ et ρ pour des types non nécessairement variables quelle que soit leur sorte.

Nous conservons les primitives du langage ML avec leur type, et nous les étendons avec les primitives

$$\begin{array}{lcl} null & : & \Pi\,(absent.\beta_1, \ldots absent.\beta_l) \\ extract^\ell & : & \Pi\,(\varphi_1, \ldots, \varphi_{\ell-1}, présent.\alpha\,, \varphi_{\ell+1} \ldots \varphi_l) \rightarrow \alpha \\ forget^\ell & : & \Pi\,(\varphi_1, \ldots \varphi_l) \rightarrow \Pi\,(\varphi_1, \ldots, \varphi_{\ell-1}, absent.\alpha\,, \varphi_{\ell+1} \ldots \varphi_l) \\ new^\ell & : & \Pi\,(\varphi_1, \ldots \varphi_l) \rightarrow \alpha \rightarrow \Pi\,(\varphi_1, \ldots, \varphi_{\ell-1}, \varepsilon.\alpha\,, \varphi_{\ell+1} \ldots \varphi_l) \end{array}$$

L'ensemble des déclarations de types ci-dessus appartient au langage $\mathcal{K}$-ML. Nous avons montré dans le chapitre précédent que le langage $\mathcal{K}$-ML est une extension naturelle de ML, et que son système de typage est décidable et unitaire. Nous venons de montrer que les objets enregistrements avec un nombre fini de champs peuvent se coder dans le langage $\mathcal{K}$-ML.

Nous étudierons plus loin les propriétés de ce codage. Nous allons d'abord l'étendre à des enregistrements dont l'ensemble des étiquettes sera potentiellement infini. Même dans les cas où cet ensemble serait fini mais grand (plus d'une dizaine d'étiquettes) ce codage resterait intéressant pour son efficacité.

4.2 Extension à des grands enregistrements

Si la solution précédente est très simple, il est clair qu'elle n'est pas convenable pour un ensemble d'étiquettes très grand. Car le prix d'un enregistrement dépend de la taille totale de l'ensemble des étiquettes et non de la taille de l'enregistrement lui-même. Ceci peut très bien convenir pour des enregistrements utilisés localement, dont le nombre d'étiquettes peut s'écrire avec un seul chiffre, mais ce n'est certainement plus le cas pour des enregistrements globaux, pour lesquels le nombre de champs effectivement utilisés dans une application raisonnable dépassera très vite la centaine.

Dans toute application, le nombre d'étiquettes utilisées sera toujours fini, mais pour une application modulaire, on ne connaît pas toujours à l'écriture d'un module les étiquettes qui seront utilisées dans les autres modules. En pratique, il est donc intéressant de savoir raisonner sur un ensemble d'étiquettes potentiellement infini. C'est bien entendu d'un point de vue théorique la seule façon d'éviter un méta-raisonnement consistant à montrer que tout calcul fait dans un système avec un petit ensemble d'étiquettes aurait pu être fait avec un ensemble d'étiquettes plus grand, et de montrer que le résultat dans le dernier cas se déduit simplement de celui dans le cas précédent. La bonne solution consiste donc à internaliser cette idée, c'est-à-dire à se placer dans un système où les calculs seront faits sur toutes les étiquettes à la fois.

Nous allons reprendre le discours ci-dessus, en essayant de décrire les intuitions et les motivations pour l'étude formelle des algèbres touffues qui sera faite dans les parties suivantes.

Reprenons l'exemple précédent de deux enregistrements

$$X = \{a = \bullet, 1\,;\ \ b = \bullet, V\},$$
$$Y = \{a = \bullet, 2\,;\ \ b = \circ, \Omega\},$$

que l'on type par

$$X : \Pi\,(a : \varepsilon_a.num\,;\ \ b : \varepsilon_b.void),$$
$$Y : \Pi\,(a : \varepsilon'_a.num\,;\ \ b : absent.\alpha'_b),$$

dans le cas où $\mathcal{L}$ est égal à $\{a, b\}$. Si nous ajoutons l'étiquette c à $\mathcal{L}$, nous écrirons

$$X = \{a = \bullet, 1\,;\ \ b = \bullet, V\,;\ \ c = \circ, \Omega\},$$
$$Y = \{a = \bullet, 2\,;\ \ b = \circ, \Omega\,;\ \ c = \circ, \Omega\}.$$

que nous typerons par

$$X : \{a : \varepsilon_a.num = b : \varepsilon_b.void\,;\ \ c = absent.\alpha_c\},$$
$$Y : \{a : \varepsilon'_a.num = b : absent.\alpha'_b\,;\ \ c = absent.\alpha'_c\}.$$

Si nous ajoutons à nouveau une étiquette, d par exemple, nous écrirons

$$X = \{a = \bullet, 1\,;\ \ b = \bullet, V\,;\ \ c = \circ, \Omega\,;\ \ d = \circ, \Omega\},$$
$$Y = \{a = \bullet, 2\,;\ \ b = \circ, \Omega\,;\ \ c = \circ, \Omega\,;\ \ d = \circ, \Omega\}.$$

Nous pouvons noter plus simplement

$$X = \{a = 1\,;\ \ b = V\},$$
$$Y = \{a = 2\},$$

quel que soit l'ensemble $\mathcal{L}$ considéré. Il suffit de lire l'écriture $\{(\ell = e_\ell)_{\ell \in I}\}$ comme l'enregistrement

$$\left\{ \left(\ell = \begin{cases} \bullet, e_\ell & \text{si } \ell \in I \\ \circ, \Omega & \text{sinon} \end{cases} \right)_{\ell \in \mathcal{L}} \right\}.$$

Il est un peu plus délicat de donner un type à cet objet. Dans le cas où $\mathcal{L}$ est égal à $\{a, b, c, d\}$ nous obtenons

$$X : \Pi\,(a : \varepsilon_a.num\,;\ \ b : \varepsilon_b.void\,;\ \ c : absent.\alpha_c\,;\ \ d : absent.\alpha_d),$$
$$Y : \Pi\,(a : \varepsilon'_a.num\,;\ \ b : absent.\alpha'_b\,;\ \ c : absent.\alpha'_c\,;\ \ d : absent.\alpha'_d).$$

De nouvelles variables ont dû être créées par rapport au cas où $\mathcal{L}$ ne contenait pas d. Aussi, on écrira

$$X : \Pi\,(a : \varepsilon_a.num\,;\ \ b : \varepsilon_b.void\,;\ \ absent.\alpha_\infty),$$
$$Y : \Pi\,(a : \varepsilon'_a.num\,;\ \ b : absent.\alpha'_b\,;\ \ absent.\alpha'_\infty).$$

où l'expression $\{(\ell : \sigma_\ell)_{\ell\in I}; \tau_\infty\}$ devra être lue comme

$$\Pi\left(\ell : \left(\begin{cases}\sigma_\ell & \text{si } \ell \in I\\ \tau_\ell & \text{sinon, où } \tau_\ell \text{ est une copie de } \tau_\infty\end{cases}\right)_{\ell\in\mathcal{L}}\right)$$

Intéressons-nous maintenant au mélange de X avec Y, par exemple dans l'expression

Z `= if true then` X `else` Y`.`

Nous devons nous placer dans un ensemble $\mathcal{L}$ contenant au moins toutes les étiquettes de X et de Y. Si nous prenons $\{a, b, c\}$ pour $\mathcal{L}$, nous obtenons, par la substitution

$$\begin{cases}\varepsilon_a \mapsto \varepsilon''_a\\ \varepsilon'_a \mapsto \varepsilon''_a\\ \varepsilon_b \mapsto absent\\ \alpha'_b \mapsto void\\ \alpha_c \mapsto \alpha''_c\\ \alpha'_c \mapsto \alpha''_c\end{cases}$$

un objet Z de type

$$Z : \Pi\,(a : \varepsilon''_a.num\,;\ \ b : absent.void\,;\ \ c : absent.\alpha''_c).$$

Si nous prenons l'ensemble d'étiquettes $\{a, b, c, d\}$, nous pouvons soit d'abord extraire le champ d de chacun des enregistrements X et Y, puis typer avec les étiquettes $\{a, b, c, d\}$ ou bien typer seulement avec les étiquettes $\{a, b, c\}$, puis extraire le champ d du résultat. Dans les deux cas, nous obtenons le même résultat,

$$Z : \Pi\,(a : \varepsilon''_a.num\,;\ \ b : absent.void\,;\ \ c : absent.\alpha''_c\,;\ \ d : absent.\alpha''_d).$$

On peut résumer cette commutation par la formule *toutes les étiquettes qui n'apparaissent pas se comportent de la même façon*. Il suffit d'effectuer ce calcul une seule fois, nous le ferons sur le *modèle* des types des champs absents. A partir de

$$X : \Pi\,(a : \varepsilon_a.num\,;\ \ b : \varepsilon_b.void\,;\ \ absent.\alpha_\infty),$$
$$Y : \Pi\,(a : \varepsilon_a.num\,;\ \ absent.\alpha'_\infty),$$

nous extrayons le type $absent.\alpha'_b$ du champ b de Y, afin que les types de X et Y aient le même ensemble de champs, puis nous unifions X et Y champ à champ ainsi que leurs modèles. Nous obtenons

$$Z : \Pi\,(a : \varepsilon_a.num\,;\ \ b : absent.void\,;\ \ absent.\alpha''_\infty).$$

Ce calcul résume les deux calculs précédents.

Nous avons évité de parler de renommages dans le discours précédent. C'est en fait une difficulté que nous avons cachée. Un modèle ressemble un peu à un type générique $\forall\vec{\alpha}_\infty.\tau$, dont on pourrait prendre des instances différentes sur chaque champ. Mais cela ne convient pas tout à fait, car les variables de modèle α_∞, α'_∞, etc., peuvent apparaître dans plusieurs modèles à la fois, elles désignent alors un même morceau de modèle. L'essentiel des parties suivantes est de donner un sens précis aux modèles.

4.3 Algèbre touffue

4.3.1 Apparition et évolution des premières touffes

Afin de bien comprendre le mécanisme précédent, et aussi parce que ce premier résultat a un intérêt propre, nous allons simplifier l'étude des types enregistrements avec un ensemble infini d'étiquettes, et simplement nous intéresser à des tuples infinis ultimement réguliers, c'est-à-dire qu'à partir d'un certain rang, toutes les composantes seront α-convertibles. Cette simplification revient à supposer que l'ensemble des étiquettes est totalement ordonné et qu'une étiquette ne peut apparaître que si toutes les étiquettes plus petites sont déjà apparues. Nous reviendrons au cas général dans la prochaine partie.

Il ressort de l'étude intuitive menée précédemment que les objets qui nous intéressent sont des tuples infinis, mais tels qu'à partir d'un certain rang tous les champs se ressemblent. Ressemblance signifie ici que les deux types sont égaux modulo le renommage de toutes leurs variables.

Le dernier élément d'un tuple représentera le modèle. Il doit être possible d'extraire de ce modèle (nous parlerons maintenant de touffe), à la fois un champ (nous dirons maintenant brin) supplémentaire et une touffe plus petite représentant les brins restants. Cette opération est le clonage. Dans l'approche intuitive précédente, nous avons considéré le clonage de façon atomique. Soit @ un symbole liant les brins extraits d'une touffe. La figure 4.1 décrit l'opération de clonage macroscopique.

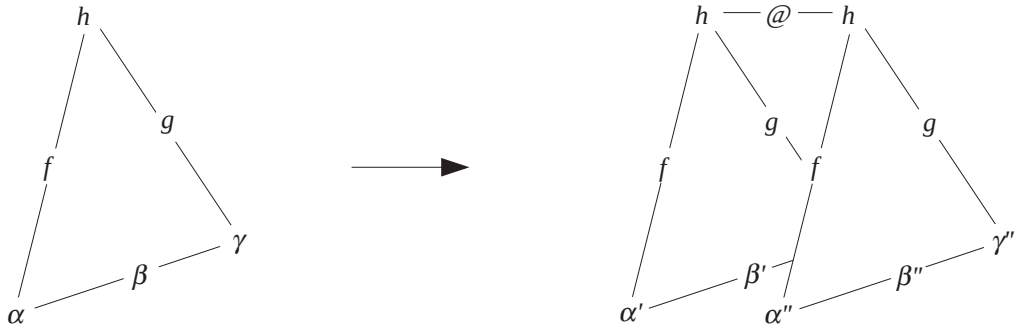


Figure 4.1: Vue macroscopique du clonage

Cette approche pose deux problèmes majeurs. D'une part elle rend difficile l'expression du partage. D'autre part elle interdit toute opération à l'intérieur des touffes, ce qui est apparu très gênant dans une extension où l'ajout de relations d'inclusion de types nécessitait la gestion de pointeurs arrière. Il est donc souhaitable de rendre l'opération de clonage microscopique. Pour les variables, le clonage peut être exprimé directement par l'opération de substitution (figure 4.2).

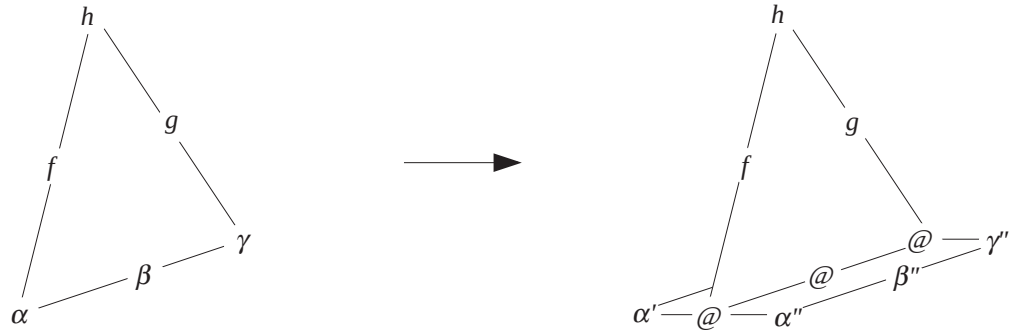


Figure 4.2: Clonage microscopique des variables par substitutions

Son évolution microscopique passe naturellement par les étapes de la figure 4.3.
Pour que cette opération soit permise, il suffit de poser en axiomes les équations suivantes

$$f(\alpha_1 \mathbin{@} \beta_1, \ldots \alpha_p \mathbin{@} \beta_p)_{i\in[1,p]} \doteq (f\,(\alpha_1, \ldots \alpha_p)) \mathbin{@} (f\,(\beta_1, \ldots \beta_p)) \qquad (f \in \mathcal{C})$$

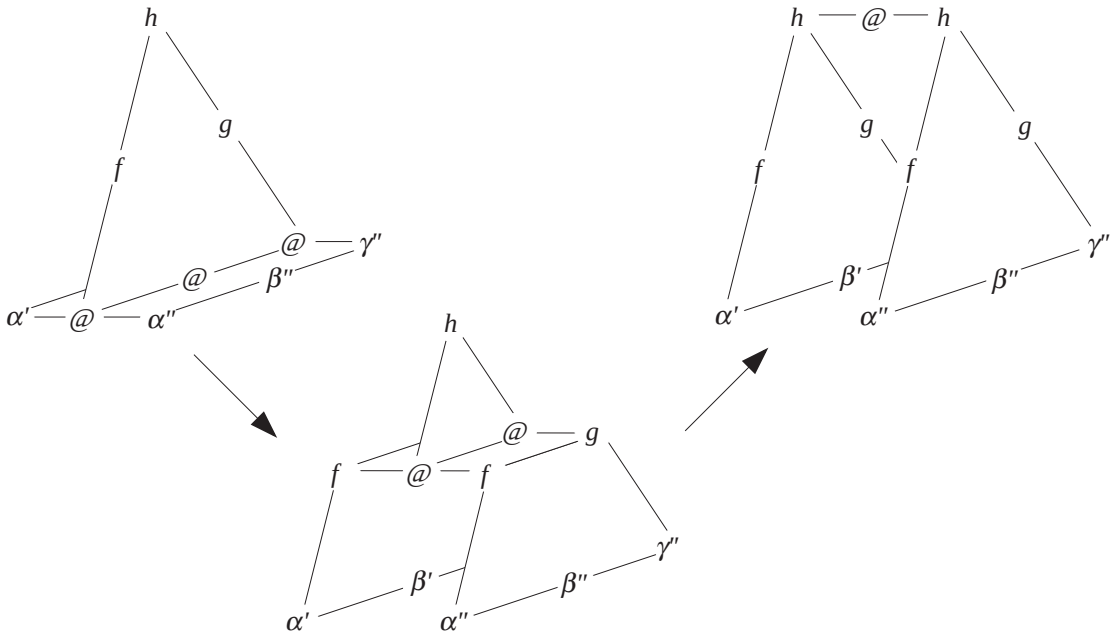


Figure 4.3: Propagation microscopique du clonage par des équations

comme axiomes. Cependant, l'introduction hâtive du symbole lieur @ fait apparaître deux sortes de termes parasites :

- Les termes

  $$(\sigma \mathbin{@} \tau) \mathbin{@} (\rho \mathbin{@} \pi)$$

  sont indésirables, car nous ne voulons extraire d'une touffe uniquement des brins et non des paquets de brins. Les seules termes désirables sont ceux de la forme

  $$\sigma \mathbin{@} \tau \mathbin{@} \underline{\rho}$$

  où les termes σ et τ ne sont pas clonables.

- Nous voulons également éliminer les touffes-rejets, qui se contiendraient strictement :

  $$\underline{\alpha} = \sigma \mathbin{@} \underline{\alpha}$$

  A priori, il est suffisant de refuser les termes rationnels pour empêcher les rejets. Cependant, nous voulons éliminer les rejets, sans pour autant interdire globalement les termes rationnels. En fait, cette solution n'en serait une qu'en apparence car elle laisserait des rejets potentiels qui empêcheraient la terminaison des simplifications. Nous reviendrons sur ce point ci-dessous dans la partie 4.3.4.

Nous résolvons ces deux problèmes simultanément en contrôlant la formation des termes par des sortes. Afin de ne pas les confondre avec les sortes déjà présentes, nous parlerons de puissances.

- Nous déclarons les termes non clonables d'une puissance ϵ, où ϵ est une nouvelle constante.

- Les termes clonables seront de puissances différentes. Pour empêcher les rejets, il suffit que leur puissance compte le nombre de brins qui ont été extraits. Nous leur affecterons donc des puissances entières.

Nous reprenons dans la partie suivante la construction de la théorie touffue avant d'en effectuer son étude détaillée.

4.3.2 Construction de la théorie touffue

Nous présentons l'algèbre touffue, puis nous justifierons ce choix par des exemples avant d'en entreprendre l'étude. Nous considérons initialement un alphabet $\mathcal{C}$ et un ensemble de variables $\mathcal{V}$ signés sur un ensemble $\mathcal{K}$, et l'algèbre $\mathcal{T}(\mathcal{V},\mathcal{C})$.

Nous distinguerons les modèles des autres termes grâce aux sortes. Nous utiliserons comme langage de sortes un ensemble $\mathcal{K}'$ égal au produit cartésien de $\mathcal{K}$ par un ensemble $\bar{\mathcal{P}}$ dont les éléments seront appelés *puissances*. Afin d'éviter de confondre des modèles n'ayant pas le même nombre de brins extraits nous les forcerons à être de puissances différentes. La puissance d'un modèle sera donc un entier n de $\mathbb{N}$, indiquant le nombre de brins extraits du modèle. Nous étendons l'ensemble $\mathbb{N}$ en $\overline{\mathbb{N}}$ en lui ajoutant un symbole ϵ qui sera la puissance des termes qui ne sont pas des modèles. Nous étendons l'ordre $<$ de $\mathbb{N}$ en un ordre sur $\overline{\mathbb{N}}$ en considérant que ϵ est la plus petite puissance. Nous prenons $\bar{\mathcal{P}}$ égal à $\overline{\mathbb{N}}$. Les puissances distinctes de ϵ seront dites *effectives*.

Nous introduisons une infinité de « copies »de l'algèbre $\mathcal{T}$ de la façon suivante. Pour chaque puissance n,

- Nous notons ι^n le couple (ι,n) de $\mathcal{K}\times\bar{\mathcal{P}}$. Par extension on note $((\iota_i)_{i\in[1,p]}\Rightarrow\kappa)^n$ le tuple $(\iota_i^n)_{i\in[1,p]}\Rightarrow\kappa^n$.
- à chaque symbole f de $\mathcal{C}$ de signature ζ, nous associons les *symboles touffus* f^n de signatures ζ^n.
- Nous introduisons une copie $\mathcal{V}^n$ de $\mathcal{V}$ de variables de puissance n.

Nous relions les différentes copies en définissant pour chaque sorte ι de $\mathcal{K}$,

- pour chaque puissance n de $\mathbb{N}$, un *symbole lieur* $@_\iota^n$ de signature $\iota^\epsilon\otimes\iota^{n+1}\Rightarrow\iota^n$ que nous infixerons,
- un *symbole passif* Π_ι de signature $\iota^0\Rightarrow\iota^\epsilon$.

Nous notons respectivement $\mathcal{K}'$, $\mathcal{C}'$ et $\mathcal{V}'$ les réunions de toutes les sortes, symboles et variables. Nous désignerons par f, g et h les symboles touffus et par f', g' et h' des symboles quelconques de $\mathcal{C}'$.

Exemple 4.2 On peut former les termes non sortés

$$f^1\left(a^1,\Pi\left(a^\epsilon\ @^0\ f^1a^1\right)\ @^1\ \alpha\right)$$

ou bien

$$a^\epsilon\ @^0\ f^1\left(\alpha\ @^1\ b^2,\beta\ @^1\ g^2b^2\right)$$

Il faut bien sûr lire $f^n\sigma\ @^n\ g^{n+1}\tau$ comme $(f^0\sigma)\ @^n\ (g^{n+1}\tau)$ et $f^n(\sigma\ @^n\ \tau,\rho)$ comme $f^n((\sigma\ @^n\ \tau),\rho)$.

Pour chaque symbole $f:\pi\Rightarrow\kappa$, de $\mathcal{C}$ d'arité p et pour chaque puissance n nous introduisons l'axiome

$$f^n(\alpha_i\ @^n_{\pi/i}\ \beta_i)_{i\in[1,p]}\doteq\left(f^\epsilon\ (\alpha_i)_{i\in[1,p]}\right)\ @^n_\kappa\ \left(f^{n+1}\ (\beta_i)_{i\in[1,p]}\right)\qquad(n\uparrow)_f$$

permettant de distribuer le symbole touffu f^n sur le symbole lieur $@_\iota^n$. Nous noterons $(n\uparrow)_f$ l'axiome ci-dessus et $(n\downarrow)_f$ l'axiome symétrique[3].

Définition 4.1 L'*algèbre touffue* sur $\mathcal{T}$ est l'algèbre libre $\mathcal{T}'(\mathcal{C}',\mathcal{V}')$ signée sur $\mathcal{K}'$ munie des précédents axiomes. □

Exemple 4.3 L'intérêt de ces axiomes est de permettre la suite d'opérations suivantes. Soit σ et σ' les deux termes $\Pi(\alpha\ @^0\ \alpha')$ et $\Pi(f^0(b^0,\beta))$ que nous voulons unifier. Nous pouvons appliquer la substitution $(\beta\mapsto\gamma\ @^0\ \gamma')$ à σ' et obtenir le terme

$$\Pi\left(f^0\left(b^0,\gamma\ @^0\ \gamma'\right)\right)$$

[3] L'orientation vers le haut remonte les lieurs des feuilles vers la racine.

qui se transforme par les axiomes de distributivité en

$$\Pi\left(f^{\epsilon}\left(b^{\epsilon},\gamma\right)@^{0}f^{1}\left(b^{1},\gamma'\right)\right)$$

qui est unifié avec σ par la substitution

$$\begin{cases}\alpha\mapsto f^{\epsilon}(b^{\epsilon},\gamma)\\ \alpha'\mapsto f^{1}(b^{1},\gamma')\end{cases}$$

4.3.3 Etude de la théorie touffue

Nous montrons ci-dessous que la théorie touffue est syntaxique.

Lemme 4.1 *La théorie touffue est résolvante.*

Démonstration Nous n'avons pas

$$\rightarrow\xrightarrow[(\bullet\bullet^{*})]{}\subset\xrightarrow[(\bullet\bullet^{*})]{*}\xrightarrow{\delta}$$

car pour le diagramme minimal obtenu par superposition d'axiomes :

$$\begin{array}{ccc}&(f^{\epsilon}\alpha_i)\ @^{n}_{\kappa}\ (f^{n+2}(\beta_i\ @^{n+1}_{\pi_{/i}}\ \gamma_i))&\\ {\scriptstyle(\epsilon)}\swarrow\nearrow & & \nwarrow\searrow{\scriptstyle(2)}\\ f^{n+1}(\alpha_i\ @^{n}_{\pi_{/i}}\ (\beta_i\ @^{n+1}_{\pi_{/i}}\ \gamma_i)) & & (f^{\epsilon}\alpha_i)\ @^{n}_{\kappa}\ ((f^{\epsilon}\beta_i)\ @^{n+1}_{\kappa}\ (f^{n+3}\gamma_i))\end{array}$$

aucune réduction du membre droit ne peut avoir lieu à la racine. Les diagrammes minimaux ci-dessus ne se produisent qu'en enchaînant des axiomes ayant des orientations opposées. Nous pourrions étudier la réduction de diagrammes plus forts, comme nous le ferons dans les théories suivantes, mais nous pouvons aussi faire disparaître ces diagrammes minimaux en privilégiant certaines orientations, c'est-à-dire en remplaçant les axiomes par un système de réécriture. Nous disposerons alors d'une famille de systèmes de réécriture canoniques qui nous permettront de montrer que tous les doublets sont résolvants.

Si l'on choisi la même orientation pour tous les axiomes de même puissance, on élimine la première famille de paires critiques. Pour éliminer la seconde, il suffit que l'on choisisse l'orientation $(\downarrow)$ pour l'axiome $(n)_f$ seulement si l'on choisit également l'orientation $(\downarrow)$ pour l'axiome $(n+1)_f$. Nous obtenons ainsi une famille d'orientations localement confluentes $\mathcal{R}_n$ formée des règles

$$\{({}^{n}\uparrow)_f \mid k<n, f\in\mathcal{C}\}\cup\{({}^{n}\downarrow)_f \mid k\geq n, f\in\mathcal{C}\}.$$

Remarquons que la hauteur des termes est conservée par A-égalité. Soit m_n l'ordre produit (m'_n, m''_n) où m'_n compte les symboles de puissance plus grande ou égale à n dans l'ordre lexicographique des puissances décroissantes et m_n compte les symboles de puissance effective strictement plus petite que n dans l'ordre lexicographique des puissances croissantes. Notons $\mathcal{T}_n$ l'ensemble des termes construits avec des symboles de puissance au plus n. L'orientation $\mathcal{R}_n$ est stable sur tous les ensembles $\mathcal{T}_k$ pour $k\geq n$ et décroissante pour l'ordre m_n. Or cet ordre est bien fondé sur les ensembles $\mathcal{T}_n$. Remarquons que l'orientation $\mathcal{R}_\infty$ composée de toutes les règles $(\uparrow)$ n'est stable sur aucun des ensembles $\mathcal{T}_k$ s'il existe des symboles d'arité nulle. Elle permettrait la dérivation infinie :

$$f^0\xrightarrow{*}f^{\epsilon}\ @^0\ldots f^{n+1}\rightarrow f^{\epsilon}\ @^0\ldots f^{n+1}\ @^n\ f^{n+2}\rightarrow\ldots$$

Nous obtenons la résolvance par la proposition 2.32. Il suffit d'utiliser les orientations canoniques

- $\mathcal{R}_{n+1}$ pour les doublets de symboles identiques $\{f^n, f^n\}$,
- $\mathcal{R}_0$ pour tous les autres doublets de symboles identiques,
- $\mathcal{R}_n$ pour les doublets $\{f^{n+1}, @^n\}$ et $\{f^{n+1}, g^{n+1}\}$,
- et une orientation $\mathcal{R}_k$ quelconque pour tous les autres doublets.

Tous les doublets sont donc résolvants. ■

Théorème 8 *La théorie touffue est syntaxique et l'unification y est unitaire.*

En effet, la présentation touffue est résolvante et pour tout couple (f', g') de symboles homogènes, il existe au plus un axiome de $f'(\mathcal{T}) \times g'(\mathcal{T})$ donc la théorie touffue est unitaire.

Soit (σ, τ) un couple de termes de $f'(\mathcal{T}) \times g'(\mathcal{T})$ et W une protection de σ et τ. Si g' est un symbole f^n de $\mathcal{C}$ d'arité p et de signature $\pi \Rightarrow \kappa$ et f' le symbole lieur $@^n_\kappa$, alors la généralisation par les axiomes de l'équation $\sigma \doteq \tau$ en dehors de W est le système :

$$(\tau_{/1} \doteq f^\epsilon(\alpha_i)_{i\in[1,p]}) \sqcap (\tau_{/2} \doteq f^{n+2}(\beta_i)_{i\in[1,p]}) \sqcap \prod_{i\in[1,p]} (\sigma_{/i} \doteq \alpha_i \mathbin{@^n_{\pi_{/i}}} \beta_i)$$

pour des variables $\alpha_1, \ldots \alpha_p$ et β_1, β_2 prise en dehors de W. Si f' et g' sont égaux d'arité p, c'est simplement

$$\prod_{i\in[1,p]} \sigma_{/i} \doteq \tau_{/i},$$

sinon, il y a collision.

Terminaison Soit m l'ordre lexicographique (m', m'', m''') où m' compte les symboles f de puissance effective dans l'ordre lexicographique des puissances croissantes et m'' compte ceux de puissance ϵ. Enfin m''' compte le nombre de variables. La généralisation et la fusion sont décroissantes pour l'ordre m, et laissent invariants les ensembles $\mathcal{T}_n$. L'ordre m est bien fondé sur les ensembles $\mathcal{T}_n$. Ce qui assure la terminaison du processus de fusion-généralisation.

La théorie touffue est stricte car l'A-égalité conserve la hauteur des termes. Nous avons donc un algorithme d'unification pour cette théorie.

4.3.4 Attention aux fausses touffes !

L'introduction d'une famille de copies peut paraître luxueuse, alors que deux copies sembleraient suffire. Effectivement, à toutes les puissances strictement positives correspondent des ensembles de symboles et d'équations isomorphes. Il est donc naturel d'examiner le système plus simple suivant. Nous limitons $\mathcal{K}'$ à $\mathcal{K}^\epsilon \cup \mathcal{K}^0$, et $\mathcal{C}'$ à

$$\mathcal{C}^\epsilon \cup \mathcal{C}^0 \cup \bigcup_{\iota\in\mathcal{K}} \{@_\iota, \Pi_\iota\}$$

où $@_\iota$ a maintenant pour signature $\iota^\epsilon \otimes \iota^0 \Rightarrow \iota^0$. On muni $\mathcal{T}'$ des axiomes

$$f^0(\alpha_i \mathbin{@_{\pi_{/i}}} \beta^0_i)_{i\in[1,p]} \doteq \left(f^\epsilon\,(\alpha^\epsilon_i)_{i\in[1,p]}\right) @_\kappa \left(f^0\,(\beta^0_i)_{i\in[1,p]}\right) \qquad ()_f$$

Malheureusement, on ne peut plus contrôler la terminaison de la réécriture comme ci-dessus, ou du processus de mutation à supposer que la théorie reste syntaxique. On ne pourra plus comme dans les parties suivantes mesurer les termes par une taille restant invariante par les axiomes. Les symboles f devraient être de poids nul.

Les sortes permettent de contrôler le nombre de brins, et interdisent de mélanger deux touffes de sortes différents, par exemple la touffe $\alpha^\epsilon @ \alpha^0$ ne peut pas substituer la touffe α^0.

4.3.5 Purification des touffes

Nous prenons $\mathcal{K}''$ égal à $\mathcal{K}$, et $\mathcal{C}''$ égal à

$$\mathcal{C} \cup \bigcup_{\iota\in\mathcal{K}} \{@_\iota, \Pi_\iota\}$$

où $@_\iota$ et Π_ι ont pour signatures $\iota \otimes \iota \Rightarrow \iota$ et $\iota \Rightarrow \iota$. L'algèbre $\mathcal{T}''(\mathcal{K}'', \mathcal{C}'')$ est l'algèbre pure associée à $\mathcal{T}'$. Nous justifions cette terminologie par la propriété suivante.

Soit φ l'application de $\mathcal{T}'$ dans $\mathcal{T}''$ qui oublie toutes les puissances, c'est-à-dire qui identifie toutes les copies. Alors il est facile de montrer par récurrence sur la taille du terme considéré qu'à tout terme pur σ'' de $\mathcal{T}''$ et toute sorte ι^ϵ de $\mathcal{K}^\epsilon$, on peut associer au plus un terme σ' de $\mathcal{T}'$ de sorte ι^ϵ tel que $\sigma'' = \varphi(\sigma')$. Cela montre qu'il suffit de préciser la sorte d'un terme pour pouvoir omettre les puissances de ses symboles.

Exemple 4.4 Le terme

$$f\,(a, \Pi\,(a \mathbin{@} fa) \mathbin{@} \alpha)$$

de puissance 1 ne peut être que le premier terme de l'exemple 4.2.

4.3.6 Une intuition touffue

Intuitivement, une touffe est un tuple infini mais ultimement régulier. Les éléments ou brins de ces touffes sont totalement ordonnés et peuvent être extraits un par un mais dans l'ordre. Certains brins sont privilégiés, mais la plupart sont indiscernables jusqu'au moment où ils sont extraits de la touffe. Nous allons maintenant donner une théorie dans laquelle il sera possible de nommer les brins et de les extraire dans un ordre quelconque.

4.4 Algèbre touffue à brins étiquetés

Nous reprenons l'essentiel des idées de la partie précédente, en les généralisant au cas où les brins sont nommés. Mais les démonstrations ne peuvent pas être étendues, et nous devrons utiliser une méthode plus puissante pour montrer que la théorie est syntaxique.

Nommer les brins nous permettra de les extraire dans un ordre quelconque. Il ne sera plus suffisant de compter les brins extraits comme dans l'algèbre simplement touffue ; il faudra les conserver explicitement. Les puissances entières devront donc être remplacées par des ensembles d'étiquettes.

Soit $\mathcal{L}$ un ensemble dénombrable d'étiquettes, et $\mathcal{P}$ l'ensemble des parties finies de $\mathcal{L}$. Nous désignerons par a, b et c des éléments de $\mathcal{L}$ et par P, Q et R ceux de $\mathcal{P}$. Pour toute étiquette a et tout ensemble d'étiquettes P ne contenant pas a, nous notons $a.P$ l'ensemble $\{a\} \cup P$, et nous notons simplement $a.b.P$ pour $a.(b.P)$. Nous considérerons l'ordre d'inclusion $\subset$ sur $\mathcal{P}$. Nous prenons pour $\bar{\mathcal{P}}$ la complétion de $\mathcal{P}$ avec la puissance ϵ que nous prendrons minimale pour l'ordre $\subset$ étendu, et de cardinal nul. Nous noterons $\bar{\mathcal{P}}$ l'ensemble $\mathcal{P}$ étendu et désignerons par $\bar{P}$, $\bar{Q}$ et $\bar{R}$ ses éléments.

4.4.1 Construction de l'algèbre touffue à brins étiquetés

Nous considérons initialement un alphabet $\mathcal{C}$ et un ensemble de variables $\mathcal{V}$ signés sur un ensemble $\mathcal{K}$, et l'algèbre $\mathcal{T}(\mathcal{V},\mathcal{C})$. Pour toute puissance $\bar{P}$,

- Nous notons $\iota^{\bar{P}}$ le couple $(\iota, \bar{P})$ de $\mathcal{K} \times \bar{\mathcal{P}}$.
- à chaque symbole f de $\mathcal{C}$ de signature ζ, nous associons un symbole $f^{\bar{P}}$ de signature $\zeta^{\bar{P}}$.
- Une copie $\mathcal{V}^{\bar{P}}$ de $\mathcal{V}$ de puissance $\bar{P}$.

Pour chaque sorte ι de $\mathcal{K}$,

- nous créons un symbole lieur $@_\iota^{a,P}$ de signature $\iota^\epsilon \otimes \iota^{a.P} \Rightarrow \iota^P$ pour chaque étiquette a et chaque puissance effective P.
- Un symbole passif Π_ι de signature $\iota^\emptyset \Rightarrow \iota^\epsilon$.

Nous introduisons

- pour chaque symbole $f : \pi \Rightarrow \kappa$ d'arité p, pour chaque étiquette a et chaque puissance effective P, l'axiome

$$f^P(\alpha_i \mathbin{@}^{a,P}_{\pi/i} \beta_i)_{i\in[1,p]} \doteq \left(f^\epsilon\,(\alpha_i)_{i\in[1,p]}\right) \mathbin{@}^{a,P}_\kappa \left(f^{a.P}\,(\beta_i)_{i\in[1,p]}\right) \qquad (a{\uparrow}P)\,f^{\pi\Rightarrow\kappa}$$

  permettant de distribuer le symbole touffu f^P sur le symbole lieur $@_\iota^{a,P}$.

- pour chaque sorte ι, chaque couple d'étiquettes (a, b) et chaque puissance effective P, l'axiome

$$\alpha \ @_\iota^{a,P} \ (\beta \ @_\iota^{b,a.P} \ \gamma) \doteq \beta \ @_\iota^{b,P} \ (\alpha \ @_\iota^{a,b.P} \ \gamma) \qquad (a{\uparrow}b{\downarrow}P)\,{}_{@}^{\iota}$$

  permettant d'exprimer la commutativité à gauche des symboles lieurs.

Nous notons respectivement $\mathcal{K}'$, $\mathcal{C}'$ et $\mathcal{V}'$ les réunions de toutes les sortes, symboles et variables.

Définition 4.2 L'*algèbre touffue à brins étiquetés* sur $\mathcal{T}$ est l'algèbre libre $\mathcal{T}'(\mathcal{C}', \mathcal{V}')$ signée sur $\mathcal{K}'$, munie des axiomes précédents. □

Remarque 4.5 Les symboles passifs n'interviennent dans aucune équation. L'algèbre touffue simple ou à brins étiquetés se généralise en ajoutant un nombre quelconque de symboles passifs de signature de la forme[4] $\pi \Rightarrow \iota^\epsilon$ où π et ι sont quelconques.

On peut en effet les ajouter dans la définition de l'algèbre touffue, ou bien les considérer comme une extension de la théorie touffue pour laquelle on pourra appliquer les résultats de C. Kirchner [36] sur le mélange de théories, dans ce cas trivial.

Nous utiliserons cette algèbre étendue dans l'implantation de l'annexe C où l'utilisateur pourra définir de nouveaux symboles passifs.

4.4.2 Etude de la théorie touffue à brins étiquetés

Lemme 4.2 *La théorie touffue à brins étiquetés est résolvante.*

Démonstration Nous ne pouvons pas utiliser la même méthode que précédemment car il n'est pas possible d'obtenir un système de ré-écriture confluent en privilégiant certaines orientations d'axiomes. Nous allons utiliser le lemme 2.38 et son corollaire 2.39.

Ni la hauteur, ni la taille ne sont conservées. Nous considérerons les ensembles $\mathcal{T}_n$ de termes formés uniquement avec la réunion de l'ensemble $\mathcal{C}^\epsilon$ des symboles de puissance ϵ et de l'ensemble

$$\bigcup_{Card\,P \leq n} \{f^P, @^{a,P}\}$$

des symboles de puissance P de cardinal plus petit que n.

Sur ces ensembles, nous définissons la taille Θ^n d'un terme en affectant aux symboles f^P d'arité q un poids $2 * (n - Card\,P) + q$, aux symboles f^ϵ d'arité q un poids $1 + q$ et un poids 1 pour tous les autres symboles en particulier les symboles lieurs. La taille Θ^n est conservée par A-égalité pour tout entier n.

Cela entraîne d'une part que la condition h_3 est vérifiée pour les ensembles $\mathcal{T}_n$, d'autre part qu'il suffit de vérifier les conditions h_1 et h_2 dans $\mathcal{T}$. La condition h_1 est satisfaite car les axiomes sont linéaires et leurs occurrences non variables sont de longueur au plus 1. Nous vérifions que

$$\xrightarrow[R,(\epsilon)]{} \xrightarrow[S_k,(k)]{k \in I\!N} \xrightarrow[T,(\epsilon)]{} \subset_{\mathcal{T}} \xrightarrow[(\bullet\bullet^*)]{*} \longrightarrow \xrightarrow[(\bullet\bullet^*)]{*}$$

par cas sur les axiomes utilisés.

Si S est la suite vide, alors R et T sont symétriques et s'annulent. Considérons les cas où S n'est pas vide.

Si $R = (a{\uparrow}P)_f$, alors S_2 est un axiome $(b{\uparrow}P)_f$ et T l'axiome $(a{\uparrow}b{\downarrow}P)_{@}$. Nous avons alors le diagramme ci-dessous en omettant les sortes des symboles lieurs

$$\begin{array}{ccc}
 & f^P(\alpha_i \ @^{a,P} \ (\beta_i \ @^{b,a.P} \ \gamma_i)) & \\
\swarrow^{(\epsilon)} & & \searrow^{(i)^{i\in[1,p]}} \\
(f^\epsilon\alpha_i) \ @^{a,P} \ (f^{a.P}(\beta_i \ @^{b,a.P} \ \gamma_i)) & & f^P(\beta_i \ @^{b,P} \ (\alpha_i \ @^{a,b.P} \ \gamma_i)) \\
\updownarrow{\scriptstyle (\underline{2})} & & \updownarrow{\scriptstyle (\epsilon)} \\
(f^\epsilon\alpha_i) \ @^{a,P} \ ((f^\epsilon\beta_i) \ @^{b,a.P} \ (f^{b.a.P}\gamma_i)) & & (f^\epsilon\beta_i) \ @^{b,P} \ (f^{b.P}(\alpha_i \ @^{a,b.P} \ \gamma_i)) \\
\searrow^{(\epsilon)} & & \swarrow^{(\underline{2})} \\
 & (f^\epsilon\beta_i) \ @^{b,P} \ ((f^\epsilon\alpha_i) \ @^{a,b.P} \ (f^{b.a.P}\gamma_i)) &
\end{array}$$

[4] La restriction sur la puissance de ι n'interviendra que pour l'obtention de formes canoniques dans la partie suivante.

Si $R = (a{\downarrow}P)_{\,f}$, alors T est nécessairement une règle $(b{\uparrow}P)_{\,f}$. Si a et b étaient identiques, S serait vide. Il faut donc au moins une application d'un axiome @ à chaque sous terme direct pour changer les symboles $@^{b,a.P}$ en $@^{a,b.P}$. Le diagramme est le même que précédemment mais en le lisant en diagonale.

Si $R = (a{\uparrow}b{\downarrow}P)_{\,@}$, alors si T est une règle $()_{\,f}$ on se retrouve dans le premier cas déjà traité, sinon T est une règle $(b{\uparrow}c{\downarrow}P)_{\,@}$. Comme (1) est une occurrence passive de T, on peut donc repousser S_1 à l'extérieur par :

$$\xrightarrow[(\underline{1})]{} \xrightarrow[T,(\epsilon)]{} \subset \rightarrow \xrightarrow[(\underline{21})]{}$$

Si c et a sont identiques, S est vide. Il faut donc au moins appliquer l'axiome $(a{\downarrow}c{\uparrow}b.P)_{\,@}$ à l'occurrence (2). On a le diagramme suivant :

$$\begin{array}{ccc}
 & \alpha\ @^{a,P}\ (\beta\ @^{b,a.P}\ (\gamma\ @^{c,b.a.P}\ \delta)) & \\
{\scriptstyle(\epsilon)}\swarrow & & \searrow{\scriptstyle(\underline{2})} \\
\beta\ @^{b,P}\ (\alpha\ @^{a,b.P}\ (\gamma\ @^{c,a.b.P}\ \delta)) & & \alpha\ @^{a,P}\ (\gamma\ @^{c,a.P}\ (\beta\ @^{b,c.a.P}\ \delta)) \\
{\scriptstyle(\underline{2})}\updownarrow & & \updownarrow{\scriptstyle(\epsilon)} \\
\beta\ @^{b,P}\ (\gamma\ @^{c,b.P}\ (\alpha\ @^{a,c.b.P}\ \delta)) & & \gamma\ @^{c,P}\ (\alpha\ @^{a,c.P}\ (\beta\ @^{b,a.c.P}\ \delta)) \\
{\scriptstyle(\epsilon)}\searrow & & \swarrow{\scriptstyle(\underline{2})} \\
 & \gamma\ @^{c,P}\ (\beta\ @^{b,c.P}\ (\alpha\ @^{a,b.c.P}\ \delta)) &
\end{array}$$

Nous avons envisagé tous les motifs minimaux de h_2. ■

Théorème 9 *La théorie touffue à brins étiquetés est syntaxique et l'unification y est unitaire.*

La présentation touffue à brins étiquettés est résolvante et pour tout couple (f', g') de symboles homogènes, il existe au plus un axiome de $f'(\mathcal{T}) \times g'(\mathcal{T})$ donc la théorie est unitaire.

Soit (σ, τ) un couple de termes de $f'(\mathcal{T}) \times g'(\mathcal{T})$ et W une protection de σ et τ.

Si g' est un symbole f^P de $\mathcal{C}$ d'arité p et de signature $\pi \Rightarrow \kappa$ et f' un symbole lieur $@_{\kappa}^{a,P}$, alors la généralisation par les axiomes de l'équation $\sigma \doteq \tau$ en dehors de W est le système :

$$(\tau_{/1} \doteq f^{\epsilon}(\alpha_i)_{i\in[1,p]} \sqcap (\tau_{/2} \doteq f^{a.P}(\beta_i)_{i\in[1,p]}) \sqcap \prod_{i\in[1,p]} (\sigma_{/i} \doteq \alpha_i\ @_{\pi_{/i}}^{a,P}\ \beta_i)$$

pour des variables $\alpha_1, \ldots \alpha_p$ et β_1, β_2 prises en dehors de W.

Si f' est un symbole lieur $@_{\iota}^{a,P}$ et g' un symbole lieur $@_{\iota}^{b,P}$ la généralisation par les axiomes de l'équation $\sigma \doteq \tau$ en dehors de W est le système :

$$\sigma_{/1} \doteq \alpha \sqcap \tau_{/1} \doteq \beta \sqcap \sigma_{/2} \doteq (\beta\ @_{\iota}^{b,a.P}\ \gamma) \sqcap \tau_{/2} \doteq (\alpha\ @_{\iota}^{b,a.P}\ \gamma)$$

pour des variables β, γ et δ prises en dehors de W. Si f' et g' sont égaux d'arité p, c'est simplement

$$\prod_{i\in[1,p]} (\sigma_{/i} \doteq \tau_{/i}),$$

sinon il y a collision.

La généralisation et la fusion sont stables dans les ensembles $\mathcal{T}_n$ et y font décroître la taille Θ^n, ce qui assure la terminaison du processus. La théorie est stricte car l'A-égalité conserve la taille Θ^n. Nous avons donc un algorithme d'unification pour la théorie touffue à brins étiquetés. ■

4.4.3 Touffes pures

Comme précédemment, on peut définir l'algèbre pure associée en conservant l'ensemble des sortes initiales et en se limitant aux symboles

$$\mathcal{C} \cup \bigcup_{\iota\in\mathcal{K}} (\{\Pi_\iota\} \cup \{@_\iota^a \mid a \in \mathcal{L}\})$$

où $@_\iota$ et Π_ι ont pour signatures $\iota \otimes \iota \Rightarrow \iota$ et $\iota \Rightarrow \iota$. En gardant les même notations que dans la partie précédente, et si φ oublie les puissances des symboles lieurs en ne conservant que leur étiquette, c'est-à-dire $\varphi(@_\iota^{a,P}) = @_\iota^a$, on peut encore montrer qu'à tout terme pur σ'' de $\mathcal{T}''$ et toute sorte ι de $\mathcal{K}$, on peut associer au plus un terme σ' de $\mathcal{T}'$ de sorte ι tel que $\sigma'' = \varphi(\sigma')$. Il est donc toujours suffisant de préciser la sorte d'un terme pour pouvoir omettre les puissances des symboles.

4.4.4 Une intuition récursive

Nous savons maintenant nommer les brins et les extraire dans n'importe quel ordre. Mais il reste surprenant qu'un brin primitif, c'est-à-dire construit directement, soit de nature différente d'un brin extrait. En effet, seul un brin primitif peut contenir un symbole Π. Dit autrement, les brins extraits ne contiennent que des symboles de $\mathcal{C}$. On peut donc envisager une nouvelle genèse en prenant comme base l'algèbre touffue pour obtenir des touffes de touffes ! Les brins de première génération pourraient comporter des touffes. Mais bien sûr cette propriété dégénère. Nous présentons dans l'annexe A l'algèbre ω-touffue, où l'on peut imbriquer les touffes de façon arbitraire. Pour simplifier, nous considérerons des brins non étiquetés.

4.5 Formes canoniques

Les algèbres touffues étudiées précédemment ont certainement d'autres applications que le typage des objets enregistrements. Au point qu'elle semblent trop puissantes pour cette simple application. Elles ont permis de délocaliser l'information sur les différents champs significatifs de la racine vers les brins. L'extraction d'un brin d'une touffe est merveilleusement effectuée par l'opération de substitution. En revanche, il devient difficile de retrouver les différents champs significatifs d'une touffe à partir de sa forme syntaxique. Les équations qui ont permis de calculer sur les champs ont maintenant l'effet pervers de fournir trop de présentations équivalentes d'un résultat. Nous allons dans cette partie essayer d'obtenir des formes canoniques, moins précises, mais beaucoup plus simples.

Nous montrerons d'abord comment faire remonter l'information des brins vers la racine. Cette opération implique en général des extractions de brins supplémentaires, que nous appellerons expansions, et qui sont en général réalisées par des substitutions, donc irréversibles. Elles affaiblissent la solution principale, mais nous montrerons que cela n'est pas catastrophique car l'expansion possède une propriété de commutation avec l'opération de substitution.

Nous définirons des formes canoniques, et nous montrerons qu'elles peuvent être obtenues par des expansions minimales. Malheureusement, les formes canoniques ne commuteront avec l'opération de sup que dans un cas particulier, ce qui nous amènera à envisager des formes canoniques plus faibles.

Nous nous plaçons dans l'algèbre touffue à brins étiquetés, et tous les résultats se particularisent donc à l'algèbre simplement touffue.

4.5.1 Expansions

Dans cette partie, nous formalisons ce que nous avons appelé l'extraction de brins. Nous étudierons les propriétés de commutation de cette opération avec l'opération de substitution.

Définition 4.3 On appelle *petite substitution* de W, l'une des quatre formes de substitutions élémentaires suivantes :

- $(\alpha \mapsto \beta \ @^{a,P} \ \gamma)$ est une *petite expansion* de W si α est une variable de W et β et γ sont des variables prises en dehors de W,
- $(\alpha \mapsto \beta)$ est un *petit renommage* de W si α est une variable de W et β une variable prise en dehors de W,
- $(\alpha \mapsto \beta)$ est une petite fusion de W si α et β sont deux variables de W,
- $(\alpha \mapsto f(\beta_i)_{i\in[1,p]})$ est une *petite structuration* de W si α est une variable de W et $(\beta_i)_{i\in[1,p]}$ sont des variables toutes deux à deux distinctes et prises en dehors de W,

Nous appelons *composition parfaite* de W toute composition $\mu_1 \dots \mu_p$ telle qu'il existe une suite $(W_i)_{i\in[0,p]}$ d'ensembles de variables satisfaisant

1. $W_0 = W$,
2. μ_i est une petite substitution de W_i,
3. W_i est égal à $W_{i-1}\mu_i$.

Une *expansion* de W est une composition parfaite de W composée uniquement de petites expansions. Une *α-expansion expansion* de W est une composition parfaite de W composée uniquement de petites expansions ou de renommages. Si W est un ensemble de termes, nous remplaçons dans les définitions précédentes W par $\mathcal{V}(W)$. Par abus, et seulement lorsqu'il n'y aura pas ambiguïté, on appellera aussi expansion [respectivement α-expansion] d'un ensemble de termes U, l'image de U par une expansion [respectivement α-expansion] de $\mathcal{V}(U)$. □

Notation Nous désignerons les expansions par les lettres φ, ψ et χ et les *petites* substitutions par l'accent $\breve{}$. La notation

$$\mu : V \to T \iff D(\mu) \subset V \wedge V\mu \subset W.$$

n'est pas adaptée pour décrire des compositions parfaites. Aussi nous noterons

$$V \xrightarrow{\mu} W \iff D(\mu) \subset V \wedge \mathcal{V}(V\mu) = W,$$

Nous ajoutons la convention supplémentaire que si μ est un renommage ou une expansion, la notation

$$V \xrightarrow{\theta} W$$

signifie que θ est un renommage de V, c'est-à-dire est bijective sur V, et la notation

$$V \xrightarrow{\varphi} W$$

signifie que φ est un expansion de V. Nous pouvons composer ces déclarations de la façon suivante

$$V \xrightarrow{\mu} W \xrightarrow{\nu} W'.$$

Dans les diagrammes, nous écrirons en haut ou à droite les substitutions universellement quantifiées (sur un certain ensemble) et en bas ou à droite, les substitutions existentiellement quantifiées. Ainsi l'énoncé de la propriété 4.9

$$\begin{array}{ccc} V & \xrightarrow{\mu} & W \\ \varphi\downarrow & & \downarrow\psi \\ W' & \xrightarrow[\nu]{} & . \end{array}$$

se lit *pour toute substitution μ de V dans W et toute expansion φ de V dans W', il existe une substitution ν de W' et une expansion ψ de W telle que $\mu\psi$ et $\varphi\nu$ soient égales (A-égales dans la propriété 4.9 car on précise que le diagramme est à considérer* modulo A-égalité*).*

Les petites substitutions permettent de ramener l'étude des propriétés de commutation à des cas très simples, grâce au lemme de décomposition suivant.

Lemme 4.3 *Toute substitution μ de domaine inclus dans W peut s'écrire comme une composition parfaite de W.*

Démonstration Nous le montrons d'abord pour une substitution élémentaire $(\alpha \mapsto \sigma)$, par récurrence sur le nombre de symboles dans σ. Si σ est une variable, alors $(\alpha \mapsto \sigma)$ est une petite substitution de taille nulle. Si σ contient n symboles, soit f le symbole de tête de σ, p son arité, et $(\alpha_i)_{i\in[1,p]}$ des variables toutes distinctes et prises en dehors de W. La substitution μ peut s'écrire

$$W \xrightarrow{(\alpha \mapsto f(\alpha_i)_{i\in[1,p]})} W_1 \xrightarrow{(\alpha_1 \mapsto \sigma_{/1})} \cdots W_{p+1} \xrightarrow{(\alpha_p \mapsto \sigma_{/p})}$$

Les sous-termes directs $(\sigma_{/i})_{i\in[1,p]}$ contiennent tous moins de n symboles. On peut remplacer chacune des substitutions élémentaires $(\alpha_i \mapsto \sigma_{/i})$ par une composition parfaite de W_p de petites substitutions.

Nous le montrons maintenant dans le cas général par récurrence sur le nombre de variables dans le domaine de la substitution μ. C'est vrai pour un domaine vide. Supposons que la propriété soit

vraie pour n. Privilégions une variable α du domaine de μ et formons la restriction ν de μ à $\overline{\mathcal{V}}(\{\alpha\})$. Soit β une variable prise en dehors de W. La substitution μ est égale à $(\alpha \mapsto \beta)\nu(\beta \mapsto \alpha\mu)$ sur W. Par l'hypothèse de récurrence, la substitution ν peut s'écrire

$$W \cup \{\beta\} \xrightarrow{\breve{\nu}_1} W_1 \ldots \xrightarrow{\breve{\nu}_p} W_p$$

Comme β n'est pas dans le domaine de μ, β est dans W_p. Par le résultat ci-dessus, la substitution élémentaire $(\beta \mapsto \alpha\mu)$ peut s'écrire

$$W_p \xrightarrow{\breve{\nu}_{p+1}} W_{p+1} \ldots \xrightarrow{\breve{\nu}_{p+q}} W_{p+q}$$

■

La décomposition précédente n'est bien sûr pas unique, mais le lemme 4.6 montrera qu'il n'y a pas tant de liberté que cela.

Lemme 4.4 *Une substitution μ est une expansion de W si et seulement si*

1. *l'image de μ ne contient que des symboles lieurs @,*
2. *l'image de μ ne contient que des variables prises dans $D(\mu)$ ou en dehors de W,*
3. *les images de deux variables distinctes ne contiennent pas de variables en commun.*

Remarque 4.6 De façon plus concrète, une expansion φ est de la forme

$$(\alpha_i \mapsto \alpha_{i0}\ @^{a_{i1},P_i} \ldots \alpha_{i(p-1)}\ @^{a_{ip},a_{i(p-1)}.\cdots.a_{i1}.P_i}\ \alpha_{ip})_{\alpha_i \in D(\varphi)}$$

où toutes les variables α_{ij} sont prises en dehors de $W \setminus D(\varphi)$ et distinctes deux à deux.

Démonstration Il est immédiat de construire une décomposition en petites expansions de la solution générale ci-dessus. Inversement, on montre que la composition parfaite d'une substitution de cette forme avec une petite expansion est encore de cette forme. ■

Remarque 4.7 L'ensemble des expansions de W est fermé par A-égalité. Cela découle sans difficulté du lemme précédent. L'A-égalité de deux expansions peut toujours se prouver en n'utilisant que des axiomes de commutativité gauche.

Lemme 4.5 *Si θ est un renommage de V et φ une expansion de $V\mu$, alors il existe une expansion ψ de V et un renommage η de $V\psi$ tels que $\varphi\mu$ et $\psi\nu$ soient égales,*

$$\begin{array}{ccc} W' & \xrightarrow{\varphi} & W \\ {\scriptstyle\theta}\uparrow & & \uparrow{\scriptstyle\eta} \\ V & \xrightarrow[\psi]{} & W'' \end{array}$$

et réciproquement en inversant le rôle des renommages et des expansions,

$$\begin{array}{ccc} W' & \xrightarrow{\theta} & W \\ {\scriptstyle\varphi}\uparrow & & \uparrow{\scriptstyle\psi} \\ V & \xrightarrow[\eta]{} & W'' \end{array}$$

Démonstration Montrons le premier diagramme. Notons V' l'ensemble $V \cup W \cup W'$. On peut étendre la substitution θ en une substitution θ' de telle façon que $\theta'(I(\theta))$ soit égal à $D(\theta)$. Notons η la restriction de θ' à W.

La substitution ψ égale à $\theta\varphi\eta^{-1}$ est la restriction à V de la substitution $\theta'\varphi\theta'^{-1}$, or θ' est un renommage de toutes les variables de φ. Les conditions du lemme 4.4 ne sont évidemment pas altérées par ce renommage, et ψ est donc une expansion. Le diagramme est vérifié par construction.

Le diagramme réciproque se montre de la même façon. ■

Lemme 4.6 *Une composition parfaite de petites substitutions qui est une α-expansion ne contient que des renommages ou des expansions.*

Démonstration Nous montrons la proposition contraposé. Soit μ la composition parfaite

$$W_0 \xrightarrow{\breve{\mu}_1} W_1 \ldots \xrightarrow{\breve{\mu}_p} W_p$$

de W_0. Si cette suite contient une structuration $f(\alpha_i)_{i\in[1,p]}$, alors $I(\mu)$ contient le symbole f. Sinon elle contient une fusion $(\alpha \mapsto \beta)$. Si α et β ont des antécédents distincts α' et β' de W_0, alors les images de α' et β' contiennent au moins une variable en commun. Sinon, α et β ont un même antécédent et son image par μ est un terme ayant plusieurs occurrences d'une même variable. Dans chacun des trois cas ci-dessus, μ ne peut pas être une expansion. ■

Corollaire 4.7 *Deux substitutions qui se composent parfaitement en une α-expansion sont des α-expansions.*

Lemme 4.8 *Pour toute petite substitution $\breve{\mu}$ et toute petite expansion $\breve{\varphi}$ de W, il existe une expansion ψ de $\mathcal{V}(W\breve{\mu})$ et une substitution ν telles que $\breve{\varphi}\mu$ et $\breve{\mu}\varphi$ soient A-égales sur W, ce que l'on peut représenter par le diagramme*

$$\begin{array}{ccc} W & \xrightarrow{\breve{\mu}} & . \\ \breve{\varphi}\downarrow & & \downarrow\psi \\ . & \xrightarrow[\nu]{} & . \end{array}$$

modulo A-égalité.

Démonstration Nous le montrons par cas sur la petite substitution $\breve{\mu}$. La substitution $\breve{\varphi}$ est de la forme $(\alpha \mapsto \alpha' \,@^{a,P}\, \alpha'')$ où α' et α'' sont prises en dehors de W, car les contraintes de sortes les empêchent d'être égales à α.

- Si $\breve{\mu}$ est un renommage $(\alpha \mapsto \beta)$, alors $(\beta \mapsto \alpha\, @^{a,P} \alpha'')$ pour ν et l'identité pour ψ conviennent.
- Si $\breve{\mu}$ est une fusion $(\alpha \mapsto \beta)$, alors $(\beta \mapsto \alpha' \,@^{a,P}\, \alpha'')$ convient à la fois pour ψ et ν. Sinon,
- Si l'image de $\breve{\mu}$ ne rencontre ni W ni l'image de $\breve{\varphi}$, alors on envisage les trois cas suivants :
  - Si le domaine de $\breve{\mu}$ n'est pas la variable α, alors $\breve{\varphi}\breve{\mu}$ et $\breve{\mu}\breve{\varphi}$ sont égales, sinon,
  - Si $\breve{\mu}$ est une expansion $(\alpha \mapsto \beta' \,@^{a,P}\, \beta'')$, il suffit de prendre l'identité pour ψ et le renommage $(\alpha' \mapsto \beta', \alpha'' \mapsto \beta'')$ pour ν.
  - Si $\breve{\mu}$ est une expansion $(\alpha \mapsto \beta' \,@^{a,Q}\, \beta'')$, soit γ une variable prise en dehors de W et distincte de α', α'', β' et β''. On prendra l'expansion $(\alpha'' \mapsto \beta' \,@^{b,a.P}\, \gamma)$ pour ν et l'expansion $(\beta'' \mapsto \alpha' \,@^{a,b.P}\, \gamma)$ pour ψ. On conclut par l'axiome $(a{\uparrow}b{\downarrow}P)_{@}$.
  - Si $\breve{\mu}$ est une structuration $(\alpha \mapsto f(\beta_i)_{i\in[1,p]})$, on prendra pour ψ la substitution

    $$(\beta_i \mapsto \gamma'_i \,@\, \gamma''_i)_{i\in[1,p]},$$

    et pour ν la substitution

    $$(\alpha' \mapsto f(\gamma'_i)_{i\in[1,p]}, \alpha'' \mapsto f(\gamma''_i)_{i\in[1,p]}),$$

    où toutes les variables γ'_i et γ''_i sont prises en dehors de W et des images de $\breve{\varphi}$ et de $\breve{\mu}$. On conclut par l'axiome $(a{\uparrow}P)_{f}$.
- Sinon, $\breve{\mu}$ est soit une petite expansion, soit une petite structuration. Il existe un renommage θ tel que $\breve{\mu}\theta$ soit une petite expansion ou une petite structuration dont l'image ne rencontre pas W ni l'image de $\breve{\varphi}$. On est donc ramené au cas précédent qui permet de conclure à l'existence d'une expansion χ et d'une substitution ξ telles que les compositions parfaites $\varphi\xi$ et $\mu\theta\chi$ soient A-égales. En utilisant le lemme 4.9, on peut ré-écrire $\theta\chi$ en une composition parfaite $\psi\eta$. Il suffit de prendre $\xi\eta^{-1}$ pour ν.

■

Remarque 4.8 Dans le cas où $\breve{\mu}$ est une petite expansion, ν est une α-expansion.

Proposition 4.9 *Pour toute substitution μ et toute expansion φ en dehors de W, il existe une expansion ψ de $\mathcal{V}(W\mu)$ et une substitution ν telles que $\varphi\nu =_A^W \mu\psi$, ce que l'on peut représenter à nouveau par le diagramme*

$$\begin{array}{ccc} W & \xrightarrow{\mu} & . \\ {\scriptstyle\varphi}\downarrow & & \downarrow{\scriptstyle\psi} \\ W' & \xrightarrow[\nu]{} & . \end{array}$$

modulo A-égalité.

Démonstration Les substitutions φ et μ peuvent s'écrire comme composition de petites substitutions. Il suffit alors de composer le lemme précédent. Plus précisément, on montre le lemme par récurrence sur les nombres p et q de petites substitutions composant φ et μ. C'est vrai si p ou q est nul. Supposons que la propriété soit vraie pour (p, q).

Pour une décomposition $(p + 1, q)$, on peut écrire φ sous la forme $\varphi'\breve{\chi}$ où $\breve{\chi}$ est petite. Par hypothèse de récurrence, il existe une expansion ψ' de W et une substitution μ' de $W\varphi$ telles que $\psi\mu'$ et $\mu\psi'$ soient A-égales. Notons W' l'ensemble des variables de $W\varphi'$. En utilisant le lemme précédent, il existe une expansion χ' de W' et une substitution ν telles que $\breve{\chi}\nu$ et $\mu\chi''$ soient A-égales. Il suffit de prendre l'expansion $\psi'\chi'$ de $\mathcal{V}(W\mu)$ pour ψ. On a ainsi obtenu le diagramme

$$\begin{array}{ccc} W & \xrightarrow{\mu} & . \\ {\scriptstyle\varphi'}\downarrow & & \downarrow{\scriptstyle\psi'} \\ . & \xrightarrow[\mu']{} & . \\ \downarrow{\scriptstyle\breve{\chi}} & & \downarrow{\scriptstyle\chi'} \\ . & \xrightarrow[\nu]{} & . \end{array}$$

Pour une décomposition $(p, q+1)$, on peut écrire μ sous la forme $\mu'\breve{\xi}$ où $\breve{\xi}$ est une petite substitution et μ' une composition de q petites substitutions. Puis on procède comme précédemment en racontant l'histoire du diagramme suivant.

$$\begin{array}{ccccc} W & \xrightarrow{\mu'} & W' & \xrightarrow[\breve{\xi}]{} & . \\ {\scriptstyle\varphi}\downarrow & & \downarrow{\scriptstyle\varphi'} & & \downarrow{\scriptstyle\psi} \\ . & \xrightarrow[\nu']{} & . & \xrightarrow[\breve{\xi}']{} & . \end{array}$$

■

Remarque 4.9 Si μ est une α-expansion, ν est aussi une α-expansion.

Corollaire 4.10 *Si un unificande U admet une solution μ en dehors de W, si φ est une expansion de U, il existe une solution ν de $U\varphi$ et une expansion ψ de $U\mu$ telle que $\mu\psi$ soit A-égal à $\varphi\nu$.*

Nous allons montrer que si μ est un unificateur principal de $\mathcal{M}$, alors on peut prendre un unificateur principal de $U\varphi$ pour ν.

Théorème 10 *Soit un unificande U, $\varphi\theta$ une α-expansion de U, et U' l'image par $\varphi\theta$ de U. Si μ et ν sont des A-unificateurs principaux de U et U', alors il existe une α-expansion $\psi\eta$ de $U'\nu$ telle que $\mu\theta\psi$ et $\varphi\theta\nu$ soient A-égales, ce que l'on résume par le diagramme*

$$\begin{array}{ccc} U & \xrightarrow{\mu} & . \\ {\scriptstyle\varphi\theta}\downarrow & & \downarrow{\scriptstyle\psi\eta} \\ U' & \xrightarrow[\nu]{} & . \end{array}$$

modulo A-égalité.

Démonstration Par la propriété 4.9, nous savons qu'il existe une substitution μ' et une expansion φ' telles que $\mu\varphi'$ et $\varphi\mu'$ soient A-égales. La substitution $\varphi\theta\nu$ est un unificateur de U, donc est A-plus petite que μ par une substitution ξ. D'autre part, la substitution $\theta^{-1}\mu'$ est un unificateur de U' donc est A-plus petite que la solution principale ν par une substitution ξ'. La composition $\xi\xi'$ est A-égale à l'expansion φ'. La substitution ξ est donc une α-expansion. En résumé,

$$\begin{array}{ccccc} U & \xrightarrow{\mu} & . & = & . \\ {\scriptstyle\varphi}\downarrow & & \downarrow{\scriptstyle\varphi'} & & \Big\downarrow{\scriptstyle\xi} \\ . & \xrightarrow[\mu']{} & . & & \\ {\scriptstyle\theta}\downarrow & & \uparrow{\scriptstyle\xi'} & & \\ U' & \xrightarrow{\nu} & . & = & . \end{array}$$

modulo A-égalité. ■

4.5.2 Termes canoniques

Dans cette partie nous définissons des termes canoniques[5], et étudions plus précisément leur structure. Nous supposons fixé un ordre total $<$ sur $\mathcal{L}$.

Définition 4.4 Un terme σ est *canonique* si tout symbole lieur est le fils d'un symbole Π où d'un autre symbole lieur.

$$\forall ux \in \mathcal{O}(\sigma),\ (\mathrm{Top}\,(\sigma_{/ux}) = @^{a,P} \Longrightarrow \mathrm{Top}\,(\sigma_{/u}) \in \{\Pi, @\} \wedge (\forall b \in P,\ b < a))$$

□

Exemple 4.10 Le terme

$$\big(\Pi(\alpha\ @^{a,P}\ \alpha')\big)\ @^{a,P}\ \beta'\ @^{a,b.P}\ \beta''$$

est canonique[6], mais

$$f\big((\alpha\ @^{a,P}\ \alpha'),\ (\beta\ @^{a,P}\ \beta'\ @^{a,b.P}\ \beta'')\big)$$

ne l'est pas.

Notation Si $\mathcal{C}$ est un ensemble de symboles, on note $\mathcal{C}(\sigma)$ l'ensemble des symboles apparaissant à la fois dans $\mathcal{C}$ et σ. Cette notation est analogue à celle utilisée pour les variables. On note $\mathcal{C}_{\not\Pi}$ l'ensemble $\mathcal{C} \setminus \{\Pi\}$.

On note $\mathcal{T}^{\bar{P}}_{\not\Pi}$ l'ensemble $\mathcal{T}(\mathcal{C}_{\not\Pi}, \mathcal{V}^{\epsilon} \cup \mathcal{V}^{\bar{P}})$ des termes construits avec des variables de puissances ϵ ou égales à $\bar{P}$ et des symboles distincts de Π. Si $\mathcal{T}_0$ est un ensemble de termes, on note $\Pi(\mathcal{T}_0)$ l'ensemble

$$\{\Pi(\sigma) \mid \sigma \in \mathcal{T}_0 \wedge \sigma : \emptyset\}.$$

[5] Nous énoncerons les définitions et propriétés avec un seul symbole passif Π, mais elles s'étendent sans difficulté au cas d'un ensemble quelconque de symboles passifs.

[6] Si j ien sûr.

Lemme 4.11 *Soit P et Q deux puissances effectives telles que A soit plus grande que Q. Tout terme de $\mathcal{T}_{\mathcal{U}}^{P}$ de puissance Q, est A-égal à un terme canonique de la forme*

$$\sigma_0 \ @^{a_1,Q} \ \ldots \sigma_{p-1} \ @^{a_p,a_{p-1}.\cdots a_1.Q} \ \sigma_p$$

où $(a_i)_{i\in[1,p]}$ est la suite ordonnée des étiquettes de $P \setminus Q$.

Démonstration Nous montrons le lemme par récurrence sur la taille $\Theta^{Card(P)+1}$ des termes. Un terme de taille nulle est une variable de puissance Q, il est dans $\mathcal{T}_{\mathcal{U}}^{Q}$. Soit σ un terme de $\mathcal{T}_{\mathcal{U}}^{P}$ de taille non nulle de puissance Q, strictement plus petite que P.

S'il est de la forme $\tau \ @^{a_0,Q} \ \rho$, le terme ρ est de plus petite taille que σ, et de puissance $a.Q$. Par récurrence, il est donc A-égal à un terme canonique de la forme

$$\sigma_1 \ @^{a_1,a_0.Q} \ \ldots \sigma_{p-1} \ @^{a_p,a_{p-1}.\cdots a_1.a_0Q} \ \sigma_p$$

où la suite $(a_i)_{i\in[1,q]}$ présente les étiquettes de $P \setminus (a_0.Q)$ dans l'ordre. Nous pouvons par les axiomes $(a{\uparrow}a_i{\downarrow}R)_{@}$ réarranger ρ et placer a_0 après a_k de telle sorte que $a_k < a_0 < a_k + 1$ (avec des cas particuliers évidents aux extrémités).

Sinon, σ est de la forme $f^Q(\sigma_j)_{j\in[1,q]}$. Si q est nul, alors σ est A-égal à

$$f^{\epsilon} \ @^{a_1,Q} \ \ldots f^{\epsilon} \ @^{a_p,a_{p-1}.\cdots a_1.Q} \ f^P$$

Sinon, par hypothèse de récurrence, les termes σ_i sont A-égaux à des termes canoniques de la forme

$$\sigma_{i0} \ @^{a_1,Q} \ \ldots \sigma_{i(p-1)} \ @^{a_p,a_{p-1}.\cdots a_1.Q} \ \sigma_{ip}$$

En utilisant les axiomes $(a_j{\uparrow}a_{j-1}.\cdots a_1.Q)_f$, pour j dans $[1,p]$ dans l'ordre des j croissants, on obtient le terme

$$(f^{\epsilon}(\sigma_{j0})_j) \ @^{a_1,Q} \ \ldots (f^{\epsilon}(\sigma_{j(p-1)})_j) \ @^{a_p,a_{p-1}.\cdots a_1.Q} \ (f^P\sigma_{ip})$$

qui satisfait les conditions du lemme. ■

Lemme 4.12 *L'ensemble des termes de A-canoniques ne contenant pas le symbole Π est exactement égal à la réunion de tous les ensembles $\mathcal{T}_{\mathcal{U}}^{\bar{P}}$.*

Démonstration Un terme canonique de puissance ϵ est dans tous les ensembles $\mathcal{T}_{\mathcal{U}}^{\bar{P}}$. Un terme canonique de puissance effective Q qui ne contient pas le symbole Π est de la forme

$$\sigma_0 \ @^{a_1,Q} \ \ldots \sigma_{p-1} \ @^{a_p,a_{p-1}.\cdots a_1.Q} \ \sigma_p$$

où p est éventuellement nul. Pour chaque i de $[0,p-1]$, le terme σ_i ne contient que des symboles f de puissance ϵ, ses variables sont donc dans $\mathcal{V}^{\epsilon}$. Si P est la puissance $a_p.\cdots a_1Q$, Le terme σ_p, de puissance P, ne contient que des symboles f de puissance P donc que des variables de puissance P. Un terme A-égal à un terme canonique de puissance P a donc également ses variables dans $\mathcal{V}^{\epsilon} \cap \mathcal{V}^{P}$. Ce qui montre l'inclusion directe. L'inclusion réciproque est contenue dans le lemme précédent. ■

4.5.3 Décompositions canoniques

De façon générale, on peut décomposer un terme en sous-termes. Cette décomposition revient à marquer certains nœuds d'un terme comme des frontières. Nous utiliserons une décomposition stable par A-égalité, en choisissant les frontières aux nœuds Π, qui ne sont jamais chevauchés par des applications d'axiomes.

Définition 4.5 L'ensemble des *termes frontières* est défini comme le plus petit ensemble noté $\mathcal{T}_f$ vérifiant :

> Si σ est un terme frontière, et si μ_f est une $\mathcal{K}$-application de domaine fini de $\mathcal{V}$ dans $\mathcal{T}_f$, alors $\sigma[\mu_f]$ est un terme frontière.

On appelle frontière toute application finie de $\mathcal{V}$ dans $\mathcal{T}_f$. On notera σ_f, τ_f et ρ_f les termes frontières et μ_f, ν_f et ξ_f les frontières. Nous utilisons les mêmes notations D et I que pour les termes et substitutions, pour désigner le domaine et l'image d'une frontière. On notera simplement σ au lieu de $\sigma[]$. On dira que $\sigma[\alpha \mapsto \tau]$ est un terme *mono-frontière* . □

Définition 4.6 On définit récursivement

- une opération de valuation ϵ_f de $\mathcal{T}_f$ dans $\mathcal{T}$ par

$$\epsilon_f(\sigma[\mu_f]) = \sigma(\alpha \mapsto \epsilon_f(\alpha\mu_f))_{\alpha \in D(\sigma_f)}$$

  Deux termes frontières qui s'évaluent sur le même terme sont dits ϵ_f-égaux.

- l'A-égalité de deux termes frontières par

$$\sigma[\mu_f] =_A \tau[\nu_f] \iff \sigma =_A \tau \wedge D(\mu_f) = D(\nu_f) \wedge \forall \alpha \in D(\mu_f),\ \alpha[\mu_f] = \alpha[\nu_f]$$

- l'ensemble des *variables frontalières* d'un terme frontière par

$$V(\sigma[\mu_f]) = D(\mu_f) \cup V(I(\mu_f))$$

  Un terme frontière est dit *en dehors* de W si $V(\sigma_f)$ ne rencontre pas W.

- l'ensemble des termes d'un terme frontière par

$$T(\sigma[\mu_f]) = \{\sigma\} \cup T(I(\mu_f))$$

- l'ensemble $\mathcal{T}_f^1$ des *termes frontières linéaires*

$$\sigma[\mu_f] \in \mathcal{T}_f^1 \iff \forall \alpha \in D(\mu_f),\ \wedge \begin{cases} \exists! u \in \mathcal{O}(\sigma),\ \sigma_{/u} = \alpha \\ \alpha\mu_f \in \mathcal{T}_f^1 \end{cases}$$

□

Définition 4.7 Si $\sigma[\mu_f]$ est un terme frontière et μ une substitution qui n'intercepte pas $V(\mu_f)$, on définit récursivement la substitution de $\sigma[\mu_f]$ par μ par

$$\sigma[\mu_f]\nu = \sigma\nu[\mu_f\nu]$$

□

Proposition 4.13 *Pour tout terme frontière σ_f, et toute substitution μ n'interceptant pas $V(\sigma_f)$, l'évaluation de la substitution de σ_f par μ est égale à la substitution de l'évaluation de σ_f par μ.*

Démonstration Par récurrence,

$$\tau\mu(\alpha \mapsto \epsilon_f(\alpha\nu_f\mu))_{\alpha \in D(\mu)},$$

est égal à

$$\tau\mu(\alpha \mapsto \epsilon_f(\alpha\nu_f)\mu)_{\alpha \in D(\mu)},$$

qui est lui-même égal à

$$\tau(\alpha \mapsto \epsilon_f(\alpha\nu_f))_{\alpha \in D(\mu)}\mu.$$

■

Définition 4.8 Une Π-*décomposition* est un terme frontière linéaire $\sigma[\mu_f]$, tel que $T(\mu_f)$ soit inclus dans $\Pi(\mathcal{T}_{\not\Pi})$, c'est-à-dire composé uniquement de termes de la forme $\Pi(\tau)$, où τ ne contient pas le symbole Π. Une Π-*décomposition* de σ est une Π-décomposition qui s'évalue en σ. □

Notation Pour tout terme σ nous notons $\mathcal{O}_\Pi(\sigma)$ (respectivement $\mathcal{O}_{\not\Pi}(\sigma)$) l'ensemble des occurrences de σ auxquelles figure (respectivement ne figure pas) un symbole Π.

Proposition 4.14 *Si $\sigma[\nu_f]$ est une Π-décomposition de τ, alors $\mathcal{O}_{\not\Pi}(\sigma)$ est le plus grand sous-ensemble de $\mathcal{O}_{\not\Pi}(\tau)$ contenant tous ses préfixes. De plus, pour toute occurrence u de $\mathcal{O}_\Pi(\sigma)$, le terme frontière $\sigma_{/u}\mu_f$ est une Π-décomposition de $\tau_{/u}$.*

Démonstration La démonstration est immédiate. ■

Proposition 4.15 *Si deux termes frontières $\sigma[\mu_f]$ et $\tau[\nu_f]$ sont des Π-décompositions s'évaluant sur un même terme ρ, alors il existe un renommage θ des variables de $D(\mu_f)$ sur les variables de $D(\nu_f)$ tel que $\sigma\theta$ et τ soient égaux et μ_f et $\theta\nu_f$ s'évaluent sur une même substitution.*

Démonstration Soit ρ le terme sur lequel s'évaluent σ_f et τ_f et O est le plus grand sous-ensemble de $\mathcal{O}_{\not\Pi}(\rho)$ contenant tous ses préfixes. Les termes σ et τ sont égaux à toutes les occurrences de O. Les ensembles d'occurrences $\mathcal{O}_\Pi(\sigma)$ et $\mathcal{O}_\Pi(\tau)$ sont donc égaux. Aux occurrences de cet ensemble, σ a des variables toutes distinctes, et τ a également des variables toutes distinctes. On peut donc construire un renommage θ tel que $\sigma\theta$ et τ soient égaux. Il est ensuite immédiat de vérifier que μ_f et $\theta\nu_f$ s'évaluent sur une même substitution. ■

Remarque 4.11 Cette propriété peut s'appliquer récursivement, et montre ainsi que la Π-décomposition est définie à des renommages près des variables frontières. Si l'on se restreint à des termes σ_f tels qu'une variable de $V(\sigma_f)$ n'apparaisse qu'une seule fois dans $T(\sigma_f)$, que l'on pourrait qualifier de super-linéaires, alors les renommages θ introduits récursivement sont tous de domaines disjoints et d'images disjointes, et se composent en un renommage η. On peut alors définir le renommage frontière de σ par η récursivement par

$$renom\,(\sigma[\mu_f], \eta) = \sigma\eta(\eta^{-1}(\alpha) \mapsto renom\,(\alpha\mu_f, \theta))_{\alpha \in D(\mu_f)}$$

Une Π-décomposition super-linéaire est définie à un renommage près.

Proposition 4.16 *Si deux termes frontières $\sigma[\mu_f]$ et $\tau[\nu_f]$ sont des Π-décompositions s'évaluant sur deux termes A-égaux, alors il existe un renommage θ des variables de $D(\mu_f)$ sur les variables de $D(\nu_f)$ tel que $\sigma\theta$ et τ soient A-égaux et μ_f et $\theta\nu_f$ s'évaluent sur des substitutions A-égales.*

Démonstration Il suffit de le montrer pour une application d'axiome à une occurrence u de $\sigma[\mu_f]$. Si u n'est pas une occurrence de σ, alors σ et τ sont égaux, et on peut conclure par la propriété précédente. Sinon on vérifie que l'axiome peut être également appliqué à σ, ce qui donne un terme σ'. Il est immédiat que $\sigma'[\mu_f]$ est une Π-décomposition. Elle est A-égale à $\tau[\nu_f]$, et on peut conclure par la propriété précédente. ■

Remarque 4.12 Comme précédemment, on peut appliquer cette propriété récursivement. Deux Π-décompositions super-linéaires s'évaluant sur deux termes A-égaux et sont A-égales à un renommage près.

Proposition 4.17 *Si σ_f est une Π-décomposition de τ et si ν est une substitution n'introduisant pas de symbole Π, et n'interceptant pas $V(\sigma_f)$, alors $\sigma_f\nu$ est une Π-décomposition de $\tau\nu$.*

Démonstration L'ensemble des termes composant $\sigma_f\nu$ est égal à l'ensemble des substitutions par ν des termes composant σ_f. ■

Lemme 4.18 *Si σ_f est une Π-décomposition d'un terme A-canonique, alors pour tout terme τ de σ_f il n'existe qu'une seule puissance effective possible pour les variables de τ.*

Démonstration Soit ρ la valuation de σ_f. Il existe un terme canonique ρ' A-égal à ρ. Soit σ'_f une décomposition de ce terme. Soit τ un terme quelconque de $V(\sigma_f)$. Il est A-égal par un renommage η à un terme τ' de $T(\sigma'_f)$.

Ce terme contient au plus le symbole Π en tête. Lui ou son sous-terme direct est un terme canonique ne contenant pas de symbole Π et est donc dans un ensemble $\mathcal{T}^P_{\not\Pi}$ pour une certaine puissance P. Cette puissance est la seule puissance possible autre que la puissance ϵ pour les variables de τ', et donc également de τ qui lui est A-égal à un renommage près. ■

4.5.4 Majorants canoniques

Nous faisons maintenant le lien entre les expansions et les termes canoniques. Plus précisément, les majorants canoniques seront les plus petits termes canoniques obtenus par expansion. Le lemme ci-dessous est technique et sera utilisé dans la preuve du lemme suivant.

Lemme 4.19 *Pour toute partition $(W_i)_{i\in[1,p]}$ d'ensembles de variables, il existe une A-plus petite expansion[7] φ vérifiant*

$$\forall i \in [1,\ p], \exists P_i \in I\!N,\ W_i\varphi \subset \mathcal{T}_{I\!H}^{P_i}$$

Soit W la réunion $\bigcup_{i\in[1,p]} W_i$, et P_i la puissance maximale des variables de W_i. Une solution est de prendre pour φ une substitution de domaine W, associant à toute variable α de W_i le terme

$$\alpha_0\ @^{a_1,Q}\ \ldots\ \alpha_{(q-1)}\ @^{a_q,a_{q-1}.\cdots a_1.Q}\ \alpha_q$$

où Q est la puissance de α, la suite $(a_i)_{i\in[1,q]}$ présente les étiquettes de $P_i \setminus Q$ dans un ordre croissant, et toutes les variables de $(\alpha_j)_{j\in[0,q]}$ sont prises en dehors de W et distinctes deux à deux.

Démonstration Il est immédiat que la solution proposée convient. Pour tout autre solution ψ, il est facile de construire une solution χ qui lui soit A-égale par composition avec φ. Les puissances R_i telles que $W_i\nu$ soit inclus dans $\mathcal{T}_{I\!H}^{R_i}$ majorent les puissances P_i car une expansion[7] remplace toujours une variable par un terme contenant une variable de puissance au moins égale. L'expansion ψ associe donc à toute variable α de W un terme σ de la forme

$$\sigma_0\ @^{b_1,Q}\ \ldots\ \sigma_{(k-1)}\ @^{b_k,b_{k-1}.\cdots b_1.Q}\ \sigma_k$$

où la suite $(b_i)_{i\in[1,k]}$ présente les étiquettes de $R_i \setminus P$. Il découle des inclusions $Q \subset P_i \subset R_i$ que la suite $(a_j)_{j\in[1,q]}$ est extraite de la suite $(b_k)_{j\in[1,k]}$. Par les axiomes de commutativité à gauche, nous pouvons réarranger le terme σ de telle façon que la suite $(a_i)_{j\in[1,q]}$ soit un préfixe de la suite $(b_i)_{i\in[1,k]}$. Il suffit de prendre pour χ la somme de toutes les substitutions $(\alpha_j \mapsto \sigma_j)$ pour j variant dans $[1, q-1]$ et de la substitution

$$(\alpha_q \mapsto \sigma_q\ \ldots\ @^{b_k,b_{k-1}.\cdots b_1.Q}\ \sigma_k),$$

α variant dans $D(\psi)$. ■

Lemme 4.20 *Pour tout terme σ de $\mathcal{T}$, il existe une plus petite expansion φ telle que $\sigma\varphi$ soit A-canonique.*

Démonstration Comme une expansion φ n'introduit pas de symbole Π, une Π-décomposition de $\sigma\varphi$ est obtenue par substitution par φ d'une Π-décomposition σ_f de σ. Par le corollaire précédent, il est nécessaire que toutes les variables d'un même terme de σ_f soit affaiblies à la même puissance.

Soit $\sim$ la relation d'équivalence définie sur l'ensemble $(\mathcal{V} \setminus \mathcal{V}^\epsilon)(\sigma)$ par

$$\alpha \sim \beta \iff \exists \tau \in T(\sigma_f),\ \alpha \in (\mathcal{V} \setminus \mathcal{V}^\epsilon)(\tau) \wedge \beta \in (\mathcal{V} \setminus \mathcal{V}^\epsilon)(\tau)$$

La recherche d'une expansion minimale se ramène donc au lemme précédent pour la partition engendrée par $\sim$. ■

Définition 4.9 L'expansion φ est appelé une *majoration canonique* de σ, et $\sigma\varphi$ est appelé un *majorant canonique* de σ. □

Les majorations canoniques sont des expansions. On peut donc leur appliquer le corollaire 4.10

[7] On peut remplacer expansion par substitution s'il l'on est assuré qu'une substitution ne peut remplacer une variable que par un terme ayant au moins une variable de puissance supérieure ou égale. Nous utiliserons cette remarque ci-après dans un cas particulier où ces conditions seront vérifiées.

4.5.5 Un cas d'échec

Si U est unificande, on peut essayer de fermer le diagramme

$$\begin{array}{ccccc} U & \xrightarrow{\mu} & . & \xrightarrow{\psi} & . \\ {\scriptstyle\varphi}\downarrow & & & & \Vert ? \\ U'' & \xrightarrow{\nu} & . & \xrightarrow{\chi} & . \end{array}$$

où μ et ν sont des unificateurs principaux et les expansions sont des majorations canoniques[8]. Ce n'est pas possible en général comme le montre l'exemple suivant dans la théorie simplement touffue :

$$\begin{array}{ccc} f(a \mathbin{@} \alpha, \beta) \doteq f(a, \beta) & \xrightarrow[\mathcal{T}^0_{\text{И}}]{(\alpha \mapsto a)} & f(a, \beta) \doteq f(a, \beta) \\ (\beta \mapsto \gamma \mathbin{@} \delta)\downarrow \mathcal{T}^1_{\text{И}} & & \Vert \\ f(a \mathbin{@} \alpha, \gamma \mathbin{@} \delta) \doteq f(a, \gamma \mathbin{@} \delta) & \xrightarrow[\mathcal{T}^1_{\text{И}}]{(\alpha \mapsto a)} & f(a, \gamma \mathbin{@} \delta) \doteq f(a, \gamma \mathbin{@} \delta) \end{array}$$

Ce contre-exemple remet en cause l'espoir de munir l'ensemble des termes A-canoniques d'une opération de sup du moins comme majoration de l'opération de sup dans $\mathcal{T}$. Il est en effet immédiat d'étendre le diagramme précédent pour montrer que cette opération ne serait pas associative.

Le problème provient des axiomes $(a{\downarrow}P)_{\,c}$ pour les constantes c. Ces axiomes ne font pas intervenir de variables, et nous parlerons d'axiomes constants. Si l'on voit les algèbres touffues comme des structures de données paresseuses, une expansion revient à demander un calcul supplémentaire, qui sera mis en mémoire dans la structure. Ce calcul s'exprime en général par le remplacement d'une variable de puissance P par la liaison d'une variable de puissance ϵ avec une variable de puissance $a.P$. C'est une opération de substitution irréversible. Mais un axiome constant permet également une expansion sans qu'il y ait substitution de variable. L'axiome est alors réversible, et l'interprétation précédente de l'expansion comme mise en mémoire d'un calcul supplémentaire n'a plus de sens.

Nous montrons dans la partie suivante qu'en l'absence d'axiome constant, les résultats précédents peuvent être affinés, et que l'ensemble des termes A-canoniques possède une opération de sup. S'il existe des axiomes constants $(a{\downarrow}P)_{\,c}$, on peut toujours forcer l'arité de c et se ramener au cas précédent. Les variables supplémentaires rendent irréversible l'opération d'expansion. Une autre solution est de chercher une forme canonique en autorisant l'opération inverse de l'expansion.

4.5.6 Une opération de sup en l'absence d'axiome constant

Hypothèse Nous supposons qu'aucun symbole f d'arité nulle n'intervient dans un axiome $(a{\downarrow}P)_{\,f}$.

Le lemme 4.19 s'énonce alors pour des substitutions quelconques. En effet, il n'était que nécessaire de se limiter à des substitutions envoyant toute variable de puissance P sur un terme ayant au moins des variables de puissances supérieures où égales à P.

Lemme 4.21 *Pour toute partition $(W_i)_{i\in[1,p]}$ d'ensembles de variables protégées par W, il existe une plus petite substitution φ vérifiant*

$$\forall i \in [1,\ p], \exists P_i \in I\!N,\ W_i\varphi \subset \mathcal{T}^{P_i}_{\text{И}}$$

Le lemme 4.20 devient

Lemme 4.22 *Une majoration de σ est plus petite que toute substitution μ telle $\sigma\mu$ soit A-canonique.*

[8] A la différence du diagramme de théorème 10, qui lui se referme !

Démonstration Dans un premier temps, nous supposons que la substitution n'introduit pas de variables Π. Une Π-décomposition de $\sigma\mu$ est obtenue par substitution par μ d'une Π-décomposition de σ. On peut donc raisonner comme dans le lemme 4.20 mais en utilisant le lemme plus puissant ci-dessus et montrer que la même expansion convient.

Si μ introduit des symboles Π, alors on peut construire une substitution ν plus petite que μ

$$\alpha\nu = \alpha\mu[\alpha_w/w]_{w\in\mathcal{O}_\Pi(\alpha\mu)},$$

où toutes les variables α_w introduites sont distinctes et prises en dehors de σ et de μ. ll est immédiat que ν satisfait les conditions du lemme. Elle est par construction plus petite que μ. ■

On en déduit le théorème suivant.

Théorème 11 *Soit U un unificande, μ une solution principale de U et φ une majoration canonique de U. Si ν est une solution principale de $U\varphi$, ψ une majoration canonique de $U\mu$ et χ une majoration canonique de $U\varphi\nu$, alors $\mu\psi$ et $\varphi\nu\chi$ sont A-égales sur U à un renommage près.*

Démonstration L'existence de ν est assurée par le corollaire 4.10. La substitution $\mu\psi$ est une majoration de U, elle est donc plus grande par une substitution ξ que φ. La substitution ξ unifie $U\varphi$, elle est donc plus grande par une substitution ξ' que ν. La substitution ξ' est une majoration de $U\varphi\nu$, elle est donc plus grande par une substitution ξ'' que χ. En résumé,

$$\begin{array}{ccc}
U & \xrightarrow{\mu} & . \\
{\scriptstyle\varphi}\downarrow & & {\scriptstyle\psi}\downarrow \\
. & \xrightarrow[\xi]{} & . \\
{\scriptstyle\nu}\downarrow & & \| \\
. & \xrightarrow[\xi']{} & . \\
{\scriptstyle\chi}\downarrow & & \| \\
. & \xrightarrow[\xi'']{} & .
\end{array}$$

modulo A-égalité, ce qui prouve

$$\varphi\nu\chi <_A \mu\psi.$$

Réciproquement, $\varphi\nu$ est une solution de U donc plus petite par une substitution ξ que μ. Comme $\xi\chi$ est une majoration de $U\mu$, elle est plus petite par une substitution ξ' que ψ. En résumé,

$$\begin{array}{ccccc}
U & \xrightarrow{\mu} & . & \xrightarrow{\psi} & . \\
{\scriptstyle\varphi}\downarrow & & \downarrow{\scriptstyle\xi} & & .\downarrow{\scriptstyle\xi'} \\
. & \xrightarrow{\nu} & . & \xrightarrow{\chi} & .
\end{array}$$

modulo A-égalité, ce qui prouve

$$\mu\psi <_A \varphi\nu\chi.$$

■

Sous l'hypothèse qu'il n'y ait pas d'axiome constant, il existe une opération d'unification sur les termes A-canoniques compatible avec l'unification dans $\mathcal{T}$. Nous en obtenons un algorithme par composition de l'algorithme d'unification dans $\mathcal{T}$ avec un algorithme de calcul de la majoration canonique. Il serait facile d'introduire l'expansion dans les systèmes de multi-équations comme une opération d'A-extension.

4.5.7 Formes canoniques

Nous reprenons le cas d'échec ci-dessus. Les majorants canoniques ne commutent pas avec le sup. Pourtant, le théorème 10 montre que les expansions commutent[9] avec le sup. Pour remédier au contre-exemple ci-dessus, une solution consiste à autoriser l'opération inverse de l'expansion.

Le premier but de cette partie est de montrer qu'il existe pour tout terme σ un plus petit terme τ s'expansant en σ. Nous appellerons contraction de σ, un terme τ dont σ soit une expansion. Nous obtiendrons le résultat en montrant que la relation de contraction est confluente. Nous avons pour cela besoin d'examiner plus en détail les relations entre les termes A-égaux. Nous avons montré dans la partie précédente que les axiomes ne pouvaient pas traverser les symboles Π. Nous allons montrer que l'ordre d'apparition des symboles touffus est conservé par A-égalité.

La définition suivante est technique, on pourra s'aider de l'exemple de la figure 4.4.

Définition 4.10 Soit $\mathcal{R}$ la propriété définie sur $\mathcal{T} \times \mathcal{V} \times \mathcal{P} \times \mathcal{V}$ par $\mathcal{R}(\sigma, \alpha, a, \beta)$ si :

1. α et β apparaissent exactement une fois dans σ.
2. Il existe trois chemins u, v et w appelés respectivement point de rencontre, chemin droit et chemin gauche, et un symbole $@^{a,P}$ tels que
   - Les chemins $(u(@^{a,P}, 1)v)$ et $(u(@^{a,P}, 2)w)$ soient dans σ et les variables α et β se trouvent au bout de ces chemins respectifs.
   - Les chemins v et w ne contiennent pas le symbole Π.
   - Le sous-chemin extrait de w en ne gardant que les directions formées de symboles touffus est égal au chemin v.

□

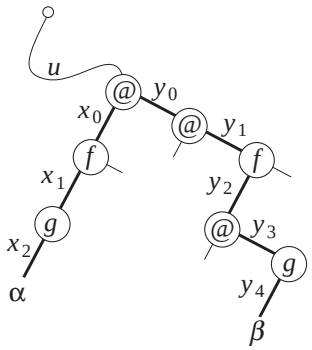


Figure 4.4: Point de rencontre

Proposition 4.23 *Soit σ un terme et α et β deux variables tels que $\mathcal{R}(\sigma, \alpha, a, \beta)$. Si τ est un terme A-égal à σ alors $\mathcal{R}(\tau, \alpha, a, \beta)$.*

Démonstration Supposons $\mathcal{R}(\sigma, \alpha, a, \beta)$. Considérons l'application d'un axiome à une occurrence u'. Les variables sont conservées par A-égalité, elles apparaissent donc exactement une fois dans τ. Nous nous contentons de donner le point de rencontre, le chemin droit et le chemin gauche du terme τ en fonction du terme σ.

- Si u' est disjoint de u ou s'il est préfixe de u et tel que le suffixe le rendant égal à u soit de longueur au moins 2, le terme $\sigma_{/u}$ n'est pas modifié. Le point de rencontre est modifié par l'application de l'axiome, mais le symbole au point de rencontre et les chemins droits et gauches sont inchangés.

[9] Il ne s'agit pas de la commutation des substitutions mais de la commutation du diagramme ; en particulier toutes les substitutions sont différentes.

- Si u est de la forme $u'z$, et l'axiome est $(a{\uparrow}P)_{\,f}$, alors le chemin droit et le chemin gauche dans τ sont obtenus par l'ajout du préfixe (f,z) à chacun des chemins correspondant dans σ. L'étiquette du symbole au point de rencontre est inchangée.

- Si u est de la forme $u'z$, et l'axiome est $(b{\uparrow}a{\downarrow}R)_{\,@}$, alors le chemin droit est inchangé et le chemin gauche dans τ est obtenu par l'ajout du préfixe $(@^{a,R},2)$ au chemin gauche dans σ. L'étiquette du symbole au point de rencontre est inchangée.

- Si u' est égale à u, il s'agit en fait d'une transformation inverse d'un des deux cas précédents.

- Si u' est suffixe de u et préfixe de $u(@^{a,P},2)w$, le chemin droit est inchangé et le chemin gauche est obtenu par échange des directions j et $j+1$ (et réajustement des sortes des symboles de ces directions).

- L'application d'un axiome à une occurrence suffixe de u et préfixe de $u(@^{a,P},1)v$ n'est pas possible.

- La seule possibilité restante est une application d'axiome à une occurrence du sous-terme $\sigma_{/u}$ qui ne se trouve pas sur les chemins gauches et droits. Elle ne modifie pas ceux-ci, ni le point de rencontre.

■

Lemme 4.24 *Soit σ un terme et α et β deux variables. Si $\mathcal{R}(\sigma,\alpha,a,\beta)$, alors σ est A-égal à un terme contenant le sous-terme*[10] *$\alpha\ @^{a,P}\ \beta$.*

Démonstration Nous le montrons par récurrence sur la longueur du chemin droit. Soit u le point de rencontre dans σ. Si le symbole à l'occurrence $u2$ est un symbole $@^{b,a.P}$, on peut appliquer l'axiome $(a{\uparrow}b{\downarrow}P)_{\,@}$. Si le symbole à l'occurrence $u2$ est un symbole de $\mathcal{C}$, c'est nécessairement le même symbole f qu'à l'occurrence $u1$. On peut appliquer un axiome $(a{\downarrow}P)_{\,f}$. Dans les deux cas, on obtient un terme τ satisfaisant la propriété $\mathcal{R}(\tau,\alpha,a,\beta)$ et dont le chemin droit est égal au chemin droit de σ moins la première direction. Sinon, $\sigma_{/u2}$ est une variable, c'est donc β. Nécessairement $\sigma_{/u1}$ est également une variable et ce ne peut être que α. ■

Lemme 4.25 *Soit α, α', β, β' et β'' cinq variables distinctes, σ un terme contenant β.*
Si $\mathcal{R}(\sigma(\beta\mapsto\beta'\ @^{a,Q}\ \beta''),\alpha,a,\alpha')$, alors $\mathcal{R}(\sigma,\alpha,a,\alpha')$.

Démonstration Le sous-terme $\beta'\ @^{a,Q}\ \beta''$ est nécessairement en dehors des chemins de σ pour le couple (α,α'). Ces chemins ne sont donc pas altérés par le renommage. ■

Lemme 4.26 *Soit α, β, β' et β'' quatre variables distinctes, σ un terme contenant β.*
Si $\mathcal{R}(\sigma(\beta\mapsto\beta'\ @^{a,Q}\ \beta''),\alpha,a,\beta'')$, alors $\mathcal{R}(\sigma,\alpha,a,\beta)$.

Démonstration Le chemin droit de σ est celui de $\sigma(\beta\mapsto\beta'\ @^{a,Q}\ \beta'')$ duquel on a enlevé le dernier élément, le point de rencontre et le chemin gauche sont inchangés. ■

Lemme 4.27 *Soit σ un terme et $(\alpha_i)_{i\in[1,p]}$ une suite de p variables et $(\beta_i)_{i\in[1,p]}$ une suite de p variables toutes disjointes. Si $\mathcal{R}(\sigma,\alpha_i,a_i,\beta_i)$ pour tout i de $[1,p]$, alors σ est A-égal à un terme contenant tous les sous-termes $(\alpha_i\ @^{a_i,P_i}\ \beta_i)_{i\in[1,p]}$.*

Démonstration Par récurrence sur p. C'est évident pour p nul. Pour montrer la propriété pour $p+1$, on applique le lemme pour p. Soit τ le terme obtenu en remplaçant tous les termes $(\alpha_i\,@^{a_i,P_i}\,\beta_i)_{i\in[1,p]}$ par des variables $(\gamma_i)_{i\in[1,p]}$ distinctes deux à deux et prises en dehors σ. Par le lemme 4.25 nous avons la propriété $\mathcal{R}(\tau,\alpha_{p+1},a_{p+1},\beta_{p+1})$ et on peut donc appliquer le lemme 4.24 pour faire apparaître le terme $\alpha_{p+1}@^{a_{p+1},P_{p+1}}\beta_{p+1}$. Le terme obtenu ρ est A-égal à τ, donc le terme $\rho(\gamma_i\mapsto\alpha_i@^{a_i,P_i}\beta_i)_{i\in[1,p]}$ est A-égal à σ et a la forme désirée. ■

[10] P est déterminé de façon unique par la puissance de β.

Lemme 4.28 *Soit σ et τ deux termes. S'il existe une α-expansion $\varphi\theta$ de σ et une α-expansion $\psi\eta$ de τ telles que $\sigma\varphi\theta$ et $\tau\psi\eta$ soient A-égales, alors il existe une terme ρ et deux α-expansions $\varphi'\theta'$ et $\psi'\eta'$ de ρ telles que $\varphi'\theta'\varphi\theta$ et $\psi'\eta'\psi\eta$ soient A-égales.*

$$\begin{array}{ccc} . & \xleftarrow{\varphi\theta} & \sigma \\ \psi\eta\uparrow & & \uparrow\varphi'\theta' \\ \tau & \xleftarrow[\psi'\eta']{} & \rho \end{array}$$

modulo A-égalité.

Démonstration Nous le montrons d'abord pour des petites expansions $(\alpha \mapsto \alpha' \ @^{a,P} \ \alpha'')$ et $(\beta \mapsto \beta' \ @^{b,Q} \ \beta'')$. Notons σ' et τ' les termes $\sigma(\alpha \mapsto \alpha' \ @^{a,P} \ \alpha'')$ et $\tau(\beta \mapsto \beta' \ @^{b,Q} \ \beta'')$.

Soit n le nombre d'occurrences de α'' dans σ', soit $(\alpha'_i \ @^{a,P} \ \alpha''_i)_{i\in[1,n]}$ une suite de termes tous disjoints et disjoints de τ', isomorphes à $\alpha' @^{a,P} \alpha''$. Soit σ'' le terme obtenu en remplaçant les i-èmes occurrences de γ dans σ par les termes $\alpha'_i \ @^{a,P} \ \alpha''_i$. L'égalité de σ' et τ' se prouve par une suite d'applications d'axiomes conduisant de σ' à τ'. Si nous appliquons cette même suite d'axiomes à σ'', nous obtenons un terme τ'' qui est égal à τ au renommage près des variables α'_i et α''_i en α' et α''. En particulier τ'' est de la forme $\rho(\beta \mapsto \beta' \ @^{b,Q} \ \beta'')$. Pour tout i de $[1,n]$, on a $\mathcal{R}(\sigma'', \alpha'_i, a_i\alpha''_i)$ donc également $\mathcal{R}(\rho(\beta \mapsto \beta' \ @^{b,Q} \ \beta''), \alpha'_i, a_i, \alpha''_i)$, car les deux termes sont A-égaux.

Si α'' et β'' sont distinctes, on est assuré de la propriété $\mathcal{R}(\rho, \alpha'_i, \alpha''_i)$ par le lemme 4.25. Il découle du lemme 4.27 que ρ est A-égal à un terme de la forme $\pi'(\alpha_i \mapsto \alpha'_i \ @^{a,P} \ \alpha''_i)_{i\in[1,n]}$. En remontant toute la chaîne, on obtient que σ' est A-égal au terme $\pi'(\alpha_i \mapsto \alpha)_{i\in[1,n]}\varphi\psi$. Les substitutions φ et ψ commutent, il suffit de prendre $\pi'(\alpha_i \mapsto \alpha)_{i\in[1,n]}$ pour π, et ψ pour φ' et φ pour ψ'.

Si α'' et β'' sont identiques, si α' et α' sont également identiques, il suffit de prendre un renommage pour θ et l'identité pour φ', ψ' et η, sinon on obtient les propriétés $\mathcal{R}(\rho, \alpha'_i, \beta_i)$ par le lemme 4.26. Il découle du lemme 4.27 que ρ est A-égal à un terme de la forme $\pi'(\alpha_i \mapsto \alpha'_i \ @^{a,P} \ \beta_i)_{i\in[1,n]}$. En remontant toute la chaîne, on obtient que σ' est A-égal au terme $\pi'(\alpha_i \mapsto \alpha)_{i\in[1,n]}(\alpha \mapsto \alpha' \ @^{a,P} \ \beta' \ @^{b,a.Q} \ \beta'')$. Il suffit de prendre $\pi'(\alpha \mapsto \alpha_i)_{i\in[1,n]}$ pour π, $(\alpha \mapsto \alpha' \ @^{a,P} \ \beta)$ pour ψ' et $(\alpha \mapsto \beta' \ @^{b,P} \ \alpha)$ pour φ'.

Lorsque α et β sont identiques, on se ramène au cas précédent en composant ψ et φ avec des renommages.

Si les expansions ne sont pas petites, elles se décomposent en des suites de petites α-expansions. Il suffit de répéter le processus ci-dessus comme pour la démonstration de la propriété 4.9. ■

Théorème 12 *Pour tout terme σ, il existe un A-plus petit terme τ tel que σ en soit une α-expansion.*

Démonstration Il suffit d'utiliser le lemme précédent. Une contraction fait décroître le nombre de variables. Lorsque celui-ci est minimal, les contractions sont A-égales, à un renommage près. ■

La plus petite contraction d'un terme est "plus petite", ce qui correspond à une représentation plus compacte, mais elle n'est pas vraiment plus simple que le terme de départ, dans le sens où elle n'est pas plus canonique. Le calcul d'une plus petite contraction est évidemment décidable, car les contractions potentielles sont formées de variables de σ et sont donc en nombre fini.

Définition 4.11 On appelle forme canonique d'un terme σ, un majorant canonique d'une plus petite contraction de σ. □

Théorème 13 *Soit U un unificande et U' son minorant canonique. L'unificande U admet des solutions si et seulement si U' admet des solutions. Si μ est un unificateur principal de U et ν un unificateur principal de U', alors $U\mu$ et $U'\nu$ ont un même unificande canonique.*

Démonstration Ce théorème découle directement du théorème 10. ■

Remarque 4.13 Ce résultat a d'abord un intérêt théorique. Il permet également de préciser la structure des algèbres touffues. En pratique nous nous contenterons des majorants canoniques. Le calcul du minorant canonique semble coûteux, bien que nous n'ayons pas vraiment cherché un algorithme efficace.

4.6 Une solution générale

4.6.1 Présentation

Nous développerons cette partie dans l'algèbre touffue à brins étiquetés. Nous reprenons la formulation dans $\mathcal{K}$-ML, en particulier la même algèbre de types, mais sur laquelle nous construisons la théorie touffue. Les sortes de base sont $\{\mathit{field}, \mathit{flag}, \mathit{usual}\}$. Nous en prenons une infinité de « copies ». Les symboles sont :

$$\begin{array}{lcl}
\rightarrow & : & \mathit{usual}^{\bar{P}} \otimes \mathit{usual}^{\bar{P}} \Rightarrow \mathit{usual}^{\bar{P}} \\
\Pi & : & \mathit{field}^{\emptyset} \Rightarrow \mathit{usual}^{0} \\
@ & : & \mathit{field}^{a} \otimes \mathit{field}^{a.P} \Rightarrow \mathit{field}^{P} \\
. & : & \mathit{flag}^{P} \otimes \mathit{usual}^{P} \Rightarrow \mathit{field}^{P} \\
\mathit{présent} & : & \mathit{flag}^{P} \\
\mathit{absent} & : & \mathit{flag}^{P} \\
f \in \mathcal{C}_0 \setminus \{\rightarrow\} & : & \underbrace{\mathit{usual}^{\bar{P}} \otimes \ldots \mathit{usual}^{\bar{P}}}_{\varrho(f)} \Rightarrow \mathit{usual}^{\bar{P}}
\end{array}$$

Nous appellerons ΠML le langage $\mathcal{K}$-A-ML défini au chapitre 3 muni des primitives ci-dessus et de la théorie touffue à brins étiquetés. Nous nous dispenserons de l'écriture des puissances des symboles, celles-ci pouvant être synthétisées. Nous noterons $a : \alpha\,;\ \beta$ au lieu de $\alpha\ @^a\ \beta$.

Par exemple $a : \alpha\,;\ (b : \beta\,;\ \gamma)$ ou plus simplement $a : \alpha\,;\ b : \beta\,;\ \gamma$ pour $\alpha\ @^a\ \beta\ @^b\ \gamma$

Les primitives du langage ML sont de type de la sorte *usual*. Les primitives sur les objets enregistrements sont :

$$\begin{array}{lcl}
\mathit{null} & : & \Pi\ (\mathit{absent}.\alpha) \\
\mathit{extract}^a & : & \Pi\ (a : \mathit{présent}.\alpha\,;\ \varphi) \rightarrow \alpha \\
\mathit{forget}^a & : & \Pi\ (a : \varphi\,;\ \psi) \rightarrow \Pi\ (a : \mathit{absent}.\alpha\,;\ \psi) \\
\mathit{new}^a & : & \Pi\ (a : \varphi\,;\ \psi) \rightarrow \alpha \rightarrow \Pi\ (a : \varepsilon.\alpha\,;\ \psi)
\end{array}$$

Nous étendons la syntaxe du langage par

$$\begin{array}{lcl}
\{\} & \equiv & \mathit{null} \\
\{r \text{ with } a = x\} & \equiv & \mathit{new}^a\ r\ x \\
\{a_1 = x_1\,;\ \ldots a_n = x_n\} & \equiv & \{\{a_1 = x_1\,;\ \ldots a_{n-1} = x_{n-1}\} \text{ with } a_n = x_n\} \\
r.a & \equiv & \mathit{extract}^a\ r \\
\{r \text{ without } a\} & \equiv & \mathit{forget}^a\ r
\end{array}$$

Le plongement du système ΠML dans le système $\mathcal{K}$-A-ML pour la théorie touffue permet d'énoncer le théorème suivant qui est le résultat principal de cette thèse.

Théorème 14 *Le typage dans le système* ΠML *est décidable. Toute expression typable possède un type principal.*

Remarque 4.14 Nous suggérons dans l'implantation l'utilisation des majorants canoniques pour montrer à l'utilisateur une information à la fois partielle, pertinente et lisible. Mais nous n'avons pas implanté l'agorithme.

En revanche le coût des minorants canoniques ne justifie pas leur utilisation. D'un point de vue théorique, il serait intéressant d'inclure l'α-expansion dans la relation d'instance. Cette extension semble ne poser aucune difficulté. Le type principal serait alors défini à α-expansion près.

4.6.2 Quelques exemples

Tous les types des exemples de ce chapitre ont été générés automatiquement à partir de l'implantation décrite dans l'annexe C. Nous commençons par les exemples très simples, puis nous montrerons les limites du système de typage.

Un enregistrement très simple,

```
> let car = {speed = 200; old = true};;
car : {speed : 'u.num; old : 'v.bool; abs.'a}
```

peut être étendu avec un nouveau champ par :

```
> let mine = {car with registration = "172 RK 92"};;
mine :
  {registration : 'u.string; speed : 'v.num; old : 'w.bool; abs.'a}
```

Cela n'exclut pas le cas où le champ étendu est déjà présent :

```
> let sidecar = {car with old = "yes"};;
sidecar : {old : 'u.string; speed : 'v.num; abs.'a}
```

Nous utiliserons la fonction

```
> let choice x y = if true then x else y;;
choice : 'a -> 'a -> 'a
```

pour contraindre les types de deux objets à être compatibles. Cela schématise par exemple le cas où les deux objets `x` et `y` sont deux éléments d'une même liste. L'enregistrement `car` précédent peut donc se mélanger avec un enregistrement n'ayant qu'un seul champ

```
> choice car {old = false};;
it : {speed : abs.num; old : 'u.bool; abs.'a}
```

Le synthétiseur de types a gardé la trace des types qui ont traversé un champ. Cela évite l'étrange comportement du langage Amber [8] qui réussit à typer la phrase

```
> choice (choice {speed = 20}  {speed = 10; old = true}) {speed = 5; old = false};;
it : {speed : 'u.num; old : abs.bool; abs.'a}
```

mais échoue avec la phrase

```
> choice {speed = 20} (choice {speed = 10; old = true} {speed = 5; old = false});;
it : {speed : 'u.num; old : abs.bool; abs.'a}
```

ce qui est plutôt surprenant ! Le fait de ne pas pouvoir oublier la trace des valeurs ayant séjourné dans les champs peut sembler une sûreté supplémentaire qui empêche l'utilisateur de mélanger des champs qui n'ont aucun point commun. C'est en fait une importante faiblesse du système de typage que nous discuterons plus loin.

La construction `without` permet d'oublier un champ explicitement,

```
> {car without old};;
it : {old : abs.'a; speed : 'u.num; abs.'b}
```

A la différence d'un oubli implicite, la trace du type ayant séjourné dans le champ correspondant est effacée, ce qui permet donc de contourner le problème ci-dessus. Ce n'est pas une véritable solution, car l'utilisateur a dû indiquer explicitement le champ à effacer au synthétiseur, ce qui n'est plus vraiment de la synthèse ! On peut voir les propriétés isolantes du `without` sur l'exemple qui suit. Nous rappelons que la construction `fun x -> M` n'est qu'une autre écriture de $\lambda x.M$.

```
> fun car bus -> choice {car without old} {bus without old};;
it : {old : 'p; 'q} -> {old : 'r; 'q} -> {old : abs.'a; 'q}
```

Dans cet oubli explicite le champ `old` du résultat est absent et ne se souvient plus des types des valeurs y ayant séjourné. Une fonction d'extraction est de la forme

```
> let speed car = car.speed;;
speed : {speed : pre.'a; 'p} -> 'a
```

ou par filtrage

```
> fun {speed = v; old = very} -> if very then 50 else v;;
it : {speed : pre.num; old : pre.bool; 'p} -> num
```

La seconde faiblesse du système de typage est due à la non généricité des valeurs liées par un lambda. Par exemple nous échouons avec l'expression

```
> fun car -> if car.old then {speed = 50} else car;;
Typechecking error: collision between pre and abs
```

bien que nous puissions typer le programme

```
> let car = {speed = 200; old = true} in if car.old then {speed = 50} else car;;
it : {speed : 'u.num; old : abs.bool; abs.'a}
```

Cette restriction n'est pas due à la façon de typer les enregistrements, mais elle est inhérente au polymorphisme de ML qui s'arrête aux frontières de l'abstraction. Nous pouvons reproduire cette situation dans le noyau du langage ; le programme

```
#fun f -> if f 1 then f 1 else f();;
```

n'est pas typable, alors que le programme très voisin

```
#let f _ = true in if f 1 then f 1 else f();;
```

l'est, bien sûr. Ce contre-exemple pourrait être contourné en remplaçant l'abstraction par une liaison, mais cela n'est évidemment pas toujours possible.

Voici un exemple qui illustre la possiblité de mettre des annotations sur des structures de données et de calculer la présence de ces annotations par le typage. Nous le réalisons dans l'environnement enrichi « prelude ».

```
> type tree ('u) = Leaf of num
>                | Nod of {left: pre.tree ('u); right: pre.tree ('u);
>                          annot: 'u.num; abs.unit}
> ;;
New constructors declared:
Nod : {left : pre.tree ('u); right : pre.tree ('u); annot :
       'u.num; abs.unit}
      -> tree ('u)
Leaf : num -> tree ('u)
```

La variable `'u` indique la présence de l'annotation `annot`. Par exemple elle est absente dans la structure

```
> let virgin = 'Nod {left = 'Leaf 1; right = 'Leaf 2 };;
virgin : tree (abs)
```

La fonction suivante annote une structure.

```
> let rec annotation =
>  function
>     Leaf n -> 'Leaf n, n
>   | Nod {left = r; right = s} ->
>      let (r,p) = annotation r in
>      let (s,q) = annotation s in
>        'Nod {left = r; right = s; annot = p+q}, p+q
> ;;
annotation : tree ('u) -> tree ('v) * num
>
> let annotate = fst o annotation
> ;;
annotate : tree ('u) -> tree ('v)
```

Elle transforme donc la structure `virgin` en une structure annotée.

```
> let consumed = annotate virgin;;
consumed : tree ('u)
```

La fonction `read` définie par

```
> let read =
>  function
>     'Leaf n -> n
>   | 'Nod r -> r.annot
```

```
> ;;
read : tree (pre) -> num
```

peut être appliquée à `consumed` mais pas à `virgin`.

```
> read virgin;;
Typechecking error: collision between pre and abs
> read consumed;;
it : num
```

Cependant, la fonction

```
> let rec left =
>  function
>    'Leaf n -> n
>  | 'Nod r -> left (r.left)
> ;;
left : tree ('u) -> num
```

peut être appliquée indifféremment à `virgin` ou `consumed`.

```
> left virgin;;
it : num
> left consumed;;
it : num
```

4.6.3 Une solution à la carte

Nous avons introduit deux formes de polymorphisme. Nous créons l'enregistrement

$$X = \{a = 1\,;\ \ b = true\}$$

avec beaucoup de polymorphisme. D'une part chaque champ présent est typé par une variable de la sorte *flag*, d'autre part le modèle des champs absents possède une variable α de la sorte *usual*,

$$X : \Pi\ (a : \varepsilon.num\,;\ \ b : \varepsilon'.bool\,;\ \ absent.\alpha).$$

La fonction d'extraction du champ a

$$\lambda r.(r.a) : \Pi\ (a : présent.\alpha\,;\ \ \varphi) \to \alpha$$

est également polymorphe, sur la valeur du champ α qu'elle extrait, mais aussi sur le modèle φ. Ainsi l'extraction du champ a à l'enregistrement précédent réussit doublement. En effet on peut utiliser le polymorphisme de l'enregistrement qui permet d'oublier des champs ou bien le polymorphisme de la fonction d'extraction qui permet d'extraire un champ en présence éventuellement d'autres champs.

Le polymorphisme des enregistrements est inhabituel, car un enregistrement est systématiquement un objet polymorphe. En général le polymorphisme est réservé aux fonctions et les valeurs construites sont monomorphes. On pourra cependant comparer le phénomène précédent avec la construction d'une liste vide qui fabrique un objet de type $list(\alpha)$ en ML.

Il est facile de donner une version du typage des objets enregistrements où ce phénomène ne se produit pas. Les enregistrements seront monomorphes, la contrepartie est qu'il ne sera plus possible de mélanger deux enregistrements définis sur des ensembles de champs différents. Une solution immédiate est d'affaiblir les types précédents, mais on peut également simplifier la structure des types et se dispenser de la sorte *flag*.

$$\begin{array}{lcl}
\to & : & usual^{\bar{P}} \otimes usual^{\bar{P}} \Rightarrow usual^{\bar{P}} \\
\Pi & : & field^{\emptyset} \Rightarrow usual^{0} \\
@ & : & field^{a} \otimes field^{a.P} \Rightarrow field^{P} \\
présent & : & usual^{P} \otimes field^{P} \\
absent & : & field^{P} \\
f \in \mathcal{C}_0 \setminus \{\to\} & : & \underbrace{usual^{\bar{P}} \otimes \ldots usual^{\bar{P}}}_{\varrho(f)} \Rightarrow usual^{\bar{P}}
\end{array}$$

On typera les primitives de la façon suivante :

$$\begin{array}{lcl} \mathit{null} & : & \Pi\,(\mathit{absent}) \\ \mathit{extract}^a & : & \Pi\,(a : \mathit{présent}(\alpha)\,;\ \varphi) \to \alpha \\ \mathit{forget}^a & : & \Pi\,(a : \varphi\,;\ \psi) \to \Pi\,(a : \mathit{absent}\,;\ \psi) \\ \mathit{new}^a & : & \Pi\,(a : \varphi\,;\ \psi) \to \alpha \to \Pi\,(a : \mathit{présent}(\alpha)\,;\ \psi) \end{array}$$

Si nous appelons ΠML′ le langage $\mathcal{K}$-A-ML muni des primitives ci-dessus, alors le langage ΠML′ est une restriction du langage ΠML, c'est-à-dire que toute expression typable dans ΠML′ est typable dans ΠML. Cela se montrerait en introduisant le langage intermédiaire ΠML″ obtenu en conservant la signature de ΠML, mais en remplaçant toutes les occurrences de termes $\mathit{absent}.\alpha$ par $\mathit{absent}.c$ dans les déclarations des primitives où c est une constante arbitraire. Les déclarations obtenues sont moins polymorphes que celles de ΠML, il est donc immédiat que $\mathcal{K}$-ML″ est une restriction de $\mathcal{K}$-ML. Si c est une constante qui n'appartient pas au langage des types de $\mathcal{K}$-ML″, alors les constantes c et *absent* n'apparaissent que simultanément et seulement dans l'expression $\mathit{absent}.c$; on peut alors remplacer toutes les occurrences de cette expression par une nouvelle constante absent'.

Dans ce langage les enregistrements ne sont plus polymorphes. La déclaration suivante dans le système ΠML :

```
> let speedy = {speed = 200};;
speedy : {speed : 'u.num; abs.'a}
```

devient dans le système ΠML′ :

```
> let racecar = {speed = 300};;
racecar : {speed : Pre (num); Abs}
```

Soit l'enregistrement

```
> let car = {speed = 300; old = false};;
car : {speed : Pre (num); Abs}
```

et la fonction d'extraction du champ `speed` définie par

```
> let speed car = car.speed;;
speed : {speed : Pre ('a); 'p} -> 'a
```

Nous pouvons encore extraire les champs de `racecar` et `car` avec la même fonction `speed`,

```
> speed racecar, speed car;;
it : num * num
```

gràce au polymorphisme de la fonction `speed`. En revanche nous ne pouvons plus effectuer l'opération

```
> fun speed -> speed racecar, speed car;;
it : ({speed : Pre (num); Abs} -> 'a) -> 'a * 'a
```

les enregistrements n'étant plus polymorphes.

D'autres variantes du système ΠML sont possibles. Comme expliqué ci-dessus, l'effet précédent pouvait être simplement obtenu en affaiblissant le type des primitives new^a comme suit :

$$\mathit{new}^a \ : \ \Pi\,(a : \varphi\,;\ \psi) \to \alpha \to \Pi\,(a : \mathit{présent}.\alpha\,;\ \psi)$$

Avec ce système, l'expression `x` ci-dessus se typerait par :

```
> let racecar = {speed = 300};;
racecar : {speed : abs.num; abs.'a}
```

Le codage permet également le renommage des champs. Une primitive effectuant cette opération serait typée par

$$\mathit{rename}^{a \leftarrow b} \ : \ \Pi\,(a : \varphi\,;\ b : \psi\,;\ \chi) \to \Pi\,(a : \mathit{absent}.\alpha\,;\ b : \varphi\,;\ \chi)$$

le champ renommé n'étant alors plus accessible. En fait l'opération plus élémentaire serait

$$exchange^{a \leftrightarrow b} \ : \quad \Pi\,(a : \varphi\,;\ \ b : \psi\,;\ \ \chi) \to \Pi\,(a : \psi\,;\ \ b : \varphi\,;\ \ \chi)$$

qui permet de retrouver $rename^{a \leftarrow b}$ comme la composition

$$forget^{a} \circ exchange^{a \leftrightarrow b}.$$

Le lecteur imaginera bien d'autres primitives très utiles. Bien sûr, une des difficultés pour toutes ces primitives non classiques reste de les compiler efficacement.

Conclusion

Sur les outils utilisés

Un outil universel

Les unificandes permettent de bien exprimer le partage, de décrire les simplifications de façon microscopique, et d'en séparer le contrôle. En présentant un problème d'unification comme un problème général, nous avons pu utiliser les unificandes dans des situations très différentes.

En particulier, nous avons ajouté aux multi-équations hiérarchisées des contraintes (degrés de propagation et de réalisation) qui ont permis un calcul incrémental de l'équilibrage d'un système hiérarchisé. Puis nous avons mélangé ces multi-équations aux relations de typage, et extrait des règles de typage de (ML_h) un ensemble de simplifications de ces unificandes conduisant à un algorithme efficace.

Des algèbres encore touffues

Les algèbres touffues ont été introduites par nécessité, mais nous utilisons très peu de leur puissance dans le codage des objets enregistrements. La généralisation aux algèbres ω-touffues à brins étiquetées était naturelle sans que l'on n'en connaisse cependant d'application réelle.

Les théories touffues sont des exemples non usuels de théories syntaxiques. Les équations de commutativité gauche produisent en général des disjonctions. Leur restriction par les sortes conduit à une théorie dans laquelle l'unification est unitaire. Les équations de commutativité médiane, seules, conduisent à une théorie non syntaxique ; dans la théorie touffue à brins étiquetés, leurs restrictions draconienne par des sortes permet d'avoir une théorie syntaxique dans laquelle l'unification est unitaire.

Nous pensons que les théories touffues ont d'autres applications que le simple codage des types enregistrements. Les équations touffues ont permis d'effectuer le clonage des modèles des types des enregistrements. Auraient-elles une application dans le typage de ML en considérant les types génériques comme des touffes ? Les systèmes de types conjonctifs n'ont pas la propriété de type principal ; il faut, pour l'obtenir, leur ajouter une opération d'expansion qui ressemble beaucoup au clonage. Cette opération peut-elle être entièrement capturée par l'ajout d'équations touffues ? Ces questions restent à approfondir.

Sur les extensions du langage

Davantage de polymorphisme ?

La solution proposée est une extension naturelle de ML, mais elle est limitée par la puissance de son polymorphisme. L'amélioration de cette solution passe par l'augmentation du polymorphisme de ML. Une façon de permettre à l'inclusion de traverser les lambda-abstractions est d'ajouter des relations d'inclusion structurelle de types. L'inclusion est alors obtenue par la contravariance des relations d'inclusion par rapport aux types flèches. En étendant l'inclusion structurelle à de l'inclusion plus générale, on pourrait utiliser le codage $\Pi\mathrm{ML}'$, plus naturel, qui permettrait d'oublier le type des champs en même temps que leur présence.

Il reste donc à montrer que l'inclusion peut cohabiter avec les théories touffues. D'un point de vue pratique, il faudra également s'assurer de l'efficacité des algorithmes de synthèse de types obtenus, notamment dans le cas où les arbres rationnels sont admis.

Une autre façon d'augmenter le polymorphisme de ML est de lui ajouter des types conjonctifs restreints de telle sorte que la synthèse reste décidable ... et efficace ! On pourra également étudier un système avec des types d'ordre supérieur qui seraient partiellement synthétisés.

Un peu de récursion

Dans tout ce travail, nous n'avons considéré que des arbres finis. Or le cas des arbres rationnels est très important pour les applications à l'héritage où l'on utilise abondamment les constructions récursives [61], mais aussi pour appliquer la théorie touffue au typage des objets sommes pour lesquels la récursion est essentielle [52]. Les termes récursifs ont été étudiés dans la théorie vide [27, 42, 15], mais en général ils se mélangent mal à une théorie équationnelle. Les théories touffues sont si régulières que nous espérons pouvoir les étendre avec des arbres rationnels. L'ajout des types récursifs à ML peut poser d'autres problèmes que celui de la synthèse de types qu'il conviendra d'étudier.

Et si on sortait les sortes ?

Dans le codage des enregistrements, nous utilisons abondamment les sortes. Elles permettent de séparer les variables de drapeaux des variables de types usuelles. Les puissances, qui sont une autre forme de sortes empêchent l'apparition de touffes dégénérées qui se contiendraient elles-mêmes et assurent la terminaison du processus de simplification. Dans le système hiérarchique, les degrés, qui auraient pu être formalisés par des sortes ordonnées, éliminent toutes les substitutions qui n'obéissent pas à la hiérarchie. D'autres extensions récentes de ML sont également basées sur des termes sortés : les classes de types de Haskell en effectuent implicitement un codage [2]. Le typage du langage Machiavelli présenté dans [48] repose sur une restriction des domaines des substitutions.

Ces exemples sont-ils encourageants car ils montrent que l'ajout de structure aux expressions de types permet un typage plus précis, ou désespérants car ils ne font qu'exploiter au maximum la structure de premier ordre, sans jamais en augmenter la puissance ?

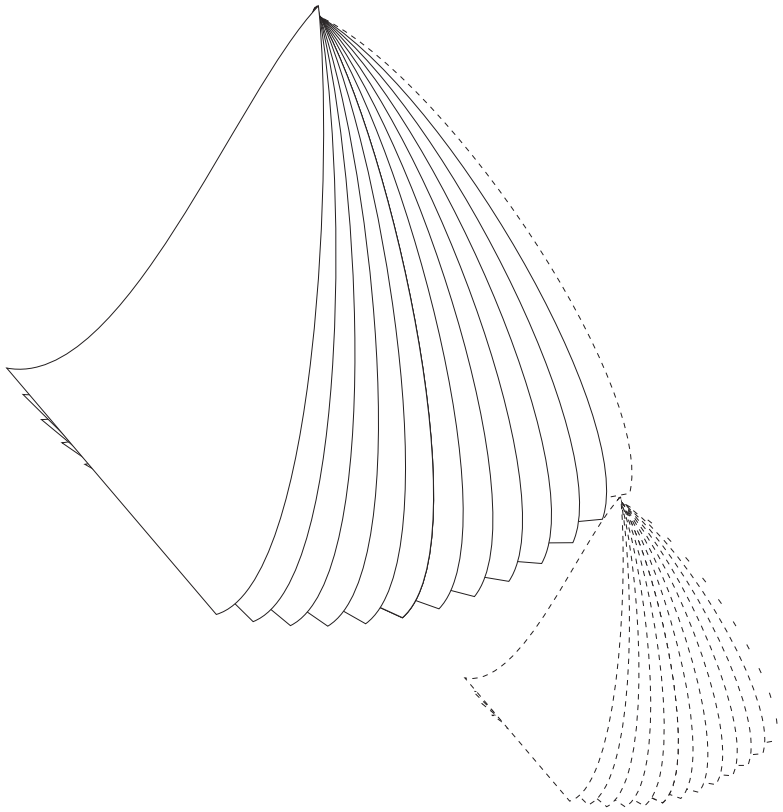
Sur l'héritage

Nous proposons une solution simple et efficace au codage des objets enregistrements et nous pensons l'étendre très prochainement avec des types récursifs. Ce langage sera une formulation complète des résultats énoncés dans [52] et pourra servir de base à un codage des classes avec héritage [61]. D'autres solutions à l'héritage ont également été proposées [49]. Nous espérons que le langage ML en permettant des possibilités de programmation objet sans pour autant exiger des déclarations de types retrouvera l'attrait que des langages plus puissants risquaient de lui voler.

Car ML a montré que son polymorphisme était bien souvent suffisant, et il existera toujours des applications où la convivialité d'un langage avec synthèse des structures de données primera sur la puissance permise par un typage explicite.

Annexe A

Algèbres ω-touffues



Cette annexe décrit la construction d'une algèbre ω-touffue simple. On montre que la théorie associée est syntaxique. On décrira la construction d'une algèbre ω-touffue à brins étiquetés, mais on ne montrera pas qu'elle est syntaxique. Nous ne voyons pas de problème particulier à cette démonstration sinon des difficultés de notations et la multiplicité des combinaisons. Un outil automatique pourrait être une aide précieuse.

Cette partie n'a pas à ce jour d'application pratique, mais sa présentation montre la généralité des algèbres touffues. En effet dans ces algèbres, les touffes pourront également contenir des touffes et cela récursivement à une profondeur arbitraire, ce qui éliminera la distinction entre les brins primitifs et les brins extraits. La multiplicité des indices ne doit pas tromper : Ceux-ci pourraient être calculés de façon automatique, et l'ensemble des termes purs est toujours très simple. Comme dans les deux théories précédentes, les indices interdisent la formations de cycles à l'intérieur des touffes et assurent la terminaison.

A.1 Construction de l'algèbre ω-touffue simple

Nous reprenons l'ensemble $\overline{\mathbb{N}}$ des entiers complétés par ϵ. Soit $\mathcal{W}$ l'ensemble des suites finies d'éléments de $\overline{\mathbb{N}}$. Nous les étendons en des applications de $\overline{\mathbb{N}}^{\mathbb{N}}$ en les complétant par une infinité de zéros. Nous identifions la suite vide avec ϵ. Nous notons $|u|$ la longueur de la suite u c'est-à-dire le plus grand entier k tel que u_k soit non nul. On utilisera l'ordre partiel $\leq$ sur ces mots, produit infini de l'ordre sur $\overline{\mathbb{N}}$, défini par

$$u \leq v \iff (\forall k \in \mathbb{N},\ u_k < v_k)$$

Nous noterons $\hat{n}$ le successeur de n pour n dans $\mathbb{N}$.

Notation La suite u dans laquelle les i_k-èmes éléments sont remplacés par x_k pour k dans $[1, q]$, est notée

$$\begin{vmatrix} i_1 & x_1 \\ & \vdots \\ i_q & x_q \end{vmatrix}^u$$

On considérera également des suites marquées, c'est-à-dire munies de fonctions partielles de domaine une partie finie de $\mathbb{N}$ dans un ensemble de marques a priori arbitraire. Par exemple la suite u marquée par

$$\begin{cases} i_1 \mapsto m_1 \\ \vdots \\ i_q \mapsto m_q \end{cases}$$

sera représentée par

$$\begin{vmatrix} m_1 & i_1 \\ & \vdots \\ m_q & i_q \end{vmatrix}$$

Un élément i marqué par $*$ peut également être redéfini à une valeur x. On écrira alors

$$\left|{*i}\ \ x\right|$$

au lieu de

$$\left|{*i}\right|$$

Nous considérons initialement un alphabet $\mathcal{C}$ et un ensemble de variables $\mathcal{V}$ signés sur un ensemble $\mathcal{K}$, et l'algèbre $\mathcal{T}(\mathcal{V}, \mathcal{C})$. Nous prenons $\mathcal{W}$ pour ensemble de puissances. Pour chaque mot u de $\mathcal{W}$,

- Nous notons ι^u le couple (ι, u).
- à chaque symbole f de $\mathcal{C}$ de signature ζ, nous associons un symbole[1] $\lfloor\rfloor_f^u$ de signature ζ^u.
- Une copie $\mathcal{V}^u$ de $\mathcal{V}$ de puissance (u).

Et pour chaque sorte ι de $\mathcal{K}$,

- nous créons la famille des symboles lieurs $(*i)\,{}_{@}^{u}$ de signature

$$(i\ 0)^u \otimes (i\ \hat{u}_i)^u \Rightarrow (i)^u,$$

- et un symbole passif $(*i)\,{}_{\Pi}^{u}$ de signature $(i\ 1)^u \Rightarrow (i\ 0)^u$.

Nous notons respectivement $\mathcal{K}'$, $\mathcal{C}'$ et $\mathcal{V}'$ les réunions de toutes les sortes, symboles et variables.

L'algèbre *ω-touffue* sur $\mathcal{T}$ est l'algèbre libre $\mathcal{T}'(\mathcal{C}', \mathcal{V}')$ signée sur $\mathcal{K}'$ munie des axiomes de distributivité

$$\lfloor i\ x\rfloor_f^u \left(\alpha_i \left\lceil {*i}\ x\right\rceil^u \beta_i\right)_{i\in[1,p]} \doteq \left(\lfloor i\ 0\rfloor_f^u(\alpha_i)_{i\in[1,p]}\right) \left\lceil {*i}\ x\right\rceil^u \left(\lfloor i\ \hat{x}\rfloor_f^u(\beta_i)_{i\in[1,p]}\right) \qquad (\uparrow i)\,{}_{f}^{u}$$

et des axiomes de commutativité médiane

$$\left(\alpha \begin{bmatrix} *i & x \\ j & 0 \end{bmatrix}^u \beta\right) \begin{bmatrix} i & x \\ *j & y \end{bmatrix}^u \left(\gamma \begin{bmatrix} *i & x \\ j & \hat{y} \end{bmatrix}^u \delta\right) \doteq \left(\alpha \begin{bmatrix} i & 0 \\ *j & y \end{bmatrix}^u \gamma\right) \begin{bmatrix} *i & x \\ j & y \end{bmatrix}^u \left(\beta \begin{bmatrix} i & \hat{x} \\ *j & y \end{bmatrix}^u \delta\right) \qquad (\uparrow i, \downarrow j)\,{}_{@}^{u}$$

[1] Nous noterons $\lfloor\rfloor$ pour les symboles touffus et $\lceil\rceil$ pour les symboles lieurs, cela permettra de placer les indices à l'intérieur des symboles.

Exemple A.1 Le terme

$$(*i)\,\underset{\Pi}{\overset{u}{}}\left(\alpha\,\left\lceil {*i \atop 1}\right\rceil^{u}\,(*j)\,\underset{\Pi}{\overset{u}{}}\left(\beta\,\left\lceil {i\ \ 1 \atop *j\ \ 1}\right\rceil^{u}\,\gamma\right)\right)$$

est un terme ω-touffu.

A.2 Etude de la théorie ω-touffue

Lemme A.1 *La théorie ω-touffue est résolvante.*

Démonstration Nous utilisons à nouveau le lemme 2.38 et son corollaire 2.39. Il suffit d'adapter le schéma de la démonstration de la partie précédente. Nous considérerons les ensembles $\mathcal{T}^u$ des termes formés uniquement avec l'ensemble

$$\bigcup_{\substack{v\leq u\\ i<|v|}} \{\lfloor\rfloor_f^v, \lceil *i\rceil^v\}$$

des symboles de puissance (v) plus petit que (u).

Sur ces ensembles, nous définissons la taille Θ^u d'un terme en affectant un poids

$$p + 2 * \sum_{v_k\neq 0} u_k - v_k + Card\,\{k\in[1,|u|] \mid v_k = 0\}$$

aux symboles $\lfloor\rfloor_f^u$ d'arité p et un poids 1 à tous les autres symboles en particulier les symboles @. La taille Θ^u est conservée par A-égalité pour tout mot u.

Les conditions h_1 et h_3 sont satisfaites. Nous vérifions que

$$\xrightarrow[R,(\epsilon)]{}\left(\xrightarrow[S,(k)]{\delta}\right)^{k\in N}\xrightarrow[T,(\epsilon)]{}\subset_H \xrightarrow[(\bullet\bullet^*)]{*}\xrightarrow[(\epsilon)]{\delta}\xrightarrow[(\bullet\bullet^*)]{*}$$

par cas sur les axiomes utilisés.

Pour alléger l'écriture, nous omettrons un certain nombre de paramètres définis par défaut. La suite de base dans toutes les notations est par défaut u sauf éventuellement pour les éléments de rangs i, j et k ; nous préciserons alors leurs valeurs, si elles sont modifiées, toujours dans cet ordre. Le symbole en indice d'une suite non marquée est par défaut f, celui d'une suite marquée est @, les symboles en indices des axiomes sont f pour une seule marque, et @ pour une double marque.

Exemple A.2 $\left\lceil {x \atop *y}\right\rceil$ signifie $\left\lceil {i\ \ x \atop *j\ \ y}\right\rceil_{@}^{u}$ et $(\uparrow i)_f$ est la règle $(\uparrow i)^u$.

Le fait que les symboles @ soient marqués est essentiel, car cela limite énormément les enchaînement possibles d'axiomes.

Si S est vide, la position des marques aux occurrences (ϵ) et $(\bullet)$ force R et T à être inverse l'un de l'autre. Ils s'annulent donc. Considérons les cas où S n'est pas vide.

Si R est un axiome $(\uparrow i)_f$ alors S est formée d'axiomes $(\uparrow j)_f$ à toutes les occurrences $(\bullet)$ et T est un axiome $(\uparrow i, \downarrow j)_{@}$. Nous avons alors le diagramme suivant :

$$\left\lfloor {x \atop y}\right\rfloor\left(\left(\alpha_i\,\left\lceil {*x \atop 0}\right\rceil\,\beta_i\right)\left\lceil {x \atop *y}\right\rceil\left(\gamma_i\,\left\lceil {*x \atop \hat{y}}\right\rceil\,\delta_i\right)\right)$$

$$\overset{(\epsilon)}{\swarrow\!\nearrow}$$

$$\left(\left\lfloor {x \atop 0}\right\rfloor\left(\alpha_i\,\left\lceil {*x \atop 0}\right\rceil\,\beta_i\right)\right)\left\lceil {x \atop *y}\right\rceil\left(\left\lfloor {x \atop \hat{y}}\right\rfloor\left(\gamma_i\,\left\lceil {*x \atop \hat{y}}\right\rceil\,\delta_i\right)\right)$$

$$\updownarrow(\underline{1})(\underline{2})$$

$$\left(\left(\left\lfloor {0 \atop 0}\right\rfloor\alpha_i\right)\left\lceil {*x \atop 0}\right\rceil\left(\left\lfloor {\hat{x} \atop 0}\right\rfloor\beta_i\right)\right)\left\lceil {x \atop *y}\right\rceil\left(\left(\left\lfloor {0 \atop \hat{y}}\right\rfloor\gamma_i\right)\left\lceil {*x \atop \hat{y}}\right\rceil\left(\left\lfloor {\hat{x} \atop \hat{y}}\right\rfloor\delta_i\right)\right)$$

$$\overset{(\epsilon)}{\nwarrow\!\searrow}$$

$$\left(\left(\left\lfloor {0 \atop 0}\right\rfloor\alpha_i\right)\left\lceil {0 \atop *y}\right\rceil\left(\left\lfloor {0 \atop \hat{y}}\right\rfloor\gamma_i\right)\right)\left\lceil {*x \atop y}\right\rceil\left(\left(\left\lfloor {\hat{x} \atop 0}\right\rfloor\beta_i\right)\left\lceil {\hat{x} \atop *y}\right\rceil\left(\left\lfloor {\hat{x} \atop \hat{y}}\right\rfloor\delta_i\right)\right)$$

qui est équivalent à :

$$\left\lfloor \begin{smallmatrix} x \\ y \end{smallmatrix} \right\rfloor \left(\left(\alpha_i \left\lceil \begin{smallmatrix} *x \\ 0 \end{smallmatrix} \right\rceil \beta_i \right) \left\lceil \begin{smallmatrix} x \\ *y \end{smallmatrix} \right\rceil \left(\gamma_i \left\lceil \begin{smallmatrix} *x \\ \hat{y} \end{smallmatrix} \right\rceil \delta_i \right) \right)$$

$$\overset{(i)^{i\in[1,p]}}{\nwarrow\!\!\searrow}$$

$$\left\lfloor \begin{smallmatrix} x \\ y \end{smallmatrix} \right\rfloor \left(\left(\alpha_i \left\lceil \begin{smallmatrix} 0 \\ *y \end{smallmatrix} \right\rceil \gamma_i \right) \left\lceil \begin{smallmatrix} *x \\ y \end{smallmatrix} \right\rceil \left(\beta_i \left\lceil \begin{smallmatrix} \hat{x} \\ *y \end{smallmatrix} \right\rceil \delta_i \right) \right)$$

$$(\epsilon)\updownarrow$$

$$\left(\left\lfloor \begin{smallmatrix} 0 \\ y \end{smallmatrix} \right\rfloor \left(\alpha_i \left\lceil \begin{smallmatrix} 0 \\ *y \end{smallmatrix} \right\rceil \gamma_i \right) \right) \left\lceil \begin{smallmatrix} *x \\ y \end{smallmatrix} \right\rceil \left(\left\lfloor \begin{smallmatrix} \hat{x} \\ y \end{smallmatrix} \right\rfloor \left(\beta_i \left\lceil \begin{smallmatrix} \hat{x} \\ *y \end{smallmatrix} \right\rceil \delta_i \right) \right)$$

$$\swarrow\!\!\nearrow \underline{(1)}\,\underline{(2)}$$

$$\left(\left(\left\lfloor \begin{smallmatrix} 0 \\ 0 \end{smallmatrix} \right\rfloor \alpha_i \right) \left\lceil \begin{smallmatrix} 0 \\ *y \end{smallmatrix} \right\rceil \left(\left\lfloor \begin{smallmatrix} 0 \\ \hat{y} \end{smallmatrix} \right\rfloor \gamma_i \right) \right) \left\lceil \begin{smallmatrix} *x \\ y \end{smallmatrix} \right\rceil \left(\left(\left\lfloor \begin{smallmatrix} \hat{x} \\ 0 \end{smallmatrix} \right\rfloor \beta_i \right) \left\lceil \begin{smallmatrix} \hat{x} \\ *y \end{smallmatrix} \right\rceil \left(\left\lfloor \begin{smallmatrix} \hat{x} \\ \hat{y} \end{smallmatrix} \right\rfloor \delta_i \right) \right)$$

Si R est un axiome $(\downarrow i)_{\,f}$, alors T est nécessairement un axiome $(\uparrow i)_{\,f}$. Il faut au moins une application d'un axiome $(\downarrow i, \uparrow j)_{\,@}$ à toutes les occurrences $(.)$ pour que T ait un sens. Le diagramme est le même que précédemment mais doit être lu en diagonale.

Si R est un axiome $(\uparrow i, \downarrow j)_{\,@}$, alors si T est un axiome (f) on se retrouve par image miroir dans un cas déjà traité. Sinon T est un axiome $(\downarrow j, \uparrow k)_{\,@}$ et S est composé d'axiomes $(\uparrow k', \downarrow j')_{\,@}$ aux occurrences $(\underline{1})$ et $(\underline{2})$. Les contraintes sur les marques imposent que $k = k'$ et $j = j'$ et que chacune des deux occurrences soit transformnée. Nous omettons les feuilles dans le diagramme suivant, ce qui n'introduit aucune ambiguité, car les contraintes de puissances ne permettent de les replacer que d'une seule façon.

$$\left(\left\lceil \begin{smallmatrix} 0 \\ 0 \\ *z \end{smallmatrix} \right\rceil \left\lceil \begin{smallmatrix} 0 \\ *y \\ z \end{smallmatrix} \right\rceil \left\lceil \begin{smallmatrix} 0 \\ \hat{y} \\ *z \end{smallmatrix} \right\rceil \right) \left\lceil \begin{smallmatrix} *x \\ y \\ z \end{smallmatrix} \right\rceil \left(\left\lceil \begin{smallmatrix} \hat{x} \\ 0 \\ *z \end{smallmatrix} \right\rceil \left\lceil \begin{smallmatrix} \hat{x} \\ *y \\ z \end{smallmatrix} \right\rceil \left\lceil \begin{smallmatrix} \hat{x} \\ \hat{y} \\ *z \end{smallmatrix} \right\rceil \right)$$

$$(\epsilon)\swarrow\qquad\qquad \nwarrow\!\!\searrow (\underline{1})(\underline{2})$$

$$\left(\left\lceil \begin{smallmatrix} 0 \\ 0 \\ *z \end{smallmatrix} \right\rceil \left\lceil \begin{smallmatrix} *x \\ 0 \\ z \end{smallmatrix} \right\rceil \left\lceil \begin{smallmatrix} \hat{x} \\ 0 \\ *z \end{smallmatrix} \right\rceil \right) \left\lceil \begin{smallmatrix} x \\ *y \\ z \end{smallmatrix} \right\rceil \left(\left\lceil \begin{smallmatrix} 0 \\ \hat{y} \\ *z \end{smallmatrix} \right\rceil \left\lceil \begin{smallmatrix} *x \\ \hat{y} \\ z \end{smallmatrix} \right\rceil \left\lceil \begin{smallmatrix} \hat{x} \\ \hat{y} \\ *z \end{smallmatrix} \right\rceil \right) \qquad \left(\left\lceil \begin{smallmatrix} 0 \\ *y \\ 0 \end{smallmatrix} \right\rceil \left\lceil \begin{smallmatrix} 0 \\ y \\ *z \end{smallmatrix} \right\rceil \left\lceil \begin{smallmatrix} 0 \\ *y \\ \hat{z} \end{smallmatrix} \right\rceil \right) \left\lceil \begin{smallmatrix} *x \\ y \\ z \end{smallmatrix} \right\rceil \left(\left\lceil \begin{smallmatrix} \hat{x} \\ *y \\ 0 \end{smallmatrix} \right\rceil \left\lceil \begin{smallmatrix} \hat{x} \\ y \\ *z \end{smallmatrix} \right\rceil \left\lceil \begin{smallmatrix} \hat{x} \\ *y \\ \hat{z} \end{smallmatrix} \right\rceil \right)$$

$$(\underline{1})(\underline{2})\updownarrow \qquad\qquad \updownarrow(\epsilon)$$

$$\left(\left\lceil \begin{smallmatrix} *x \\ 0 \\ 0 \end{smallmatrix} \right\rceil \left\lceil \begin{smallmatrix} x \\ 0 \\ *z \end{smallmatrix} \right\rceil \left\lceil \begin{smallmatrix} *x \\ 0 \\ \hat{z} \end{smallmatrix} \right\rceil \right) \left\lceil \begin{smallmatrix} x \\ *y \\ z \end{smallmatrix} \right\rceil \left(\left\lceil \begin{smallmatrix} *x \\ \hat{y} \\ 0 \end{smallmatrix} \right\rceil \left\lceil \begin{smallmatrix} x \\ \hat{y} \\ *z \end{smallmatrix} \right\rceil \left\lceil \begin{smallmatrix} *x \\ \hat{y} \\ \hat{z} \end{smallmatrix} \right\rceil \right) \qquad \left(\left\lceil \begin{smallmatrix} 0 \\ *y \\ 0 \end{smallmatrix} \right\rceil \left\lceil \begin{smallmatrix} *x \\ y \\ 0 \end{smallmatrix} \right\rceil \left\lceil \begin{smallmatrix} \hat{x} \\ *y \\ 0 \end{smallmatrix} \right\rceil \right) \left\lceil \begin{smallmatrix} x \\ y \\ *z \end{smallmatrix} \right\rceil \left(\left\lceil \begin{smallmatrix} 0 \\ *y \\ \hat{z} \end{smallmatrix} \right\rceil \left\lceil \begin{smallmatrix} *x \\ y \\ \hat{z} \end{smallmatrix} \right\rceil \left\lceil \begin{smallmatrix} \hat{x} \\ *y \\ \hat{z} \end{smallmatrix} \right\rceil \right)$$

$$(\epsilon)\nwarrow\!\!\searrow \qquad\qquad \swarrow\!\!\nearrow (\underline{2})$$

$$\left(\left\lceil \begin{smallmatrix} *x \\ 0 \\ 0 \end{smallmatrix} \right\rceil \left\lceil \begin{smallmatrix} x \\ *y \\ 0 \end{smallmatrix} \right\rceil \left\lceil \begin{smallmatrix} *x \\ \hat{y} \\ 0 \end{smallmatrix} \right\rceil \right) \left\lceil \begin{smallmatrix} x \\ y \\ *z \end{smallmatrix} \right\rceil \left(\left\lceil \begin{smallmatrix} *x \\ 0 \\ \hat{z} \end{smallmatrix} \right\rceil \left\lceil \begin{smallmatrix} x \\ *y \\ \hat{z} \end{smallmatrix} \right\rceil \left\lceil \begin{smallmatrix} *x \\ \hat{y} \\ \hat{z} \end{smallmatrix} \right\rceil \right)$$

Nous avons envisagé tous les diagrammes minimaux de h_2. ■

Comme pour tous les couples (f', h') de symboles homogènes, les ensembles $A(f', g')$ sont au plus des singletons, on peut donc énoncer le théorème suivant.

Théorème 15 *La théorie ω-touffue est syntaxique.*

La généralisation et la fusion sont stables dans les ensembles H^u et y font décroître la taille Θ^u. Ce qui assure la terminaison du processus de fusion-généralisation. La théorie est également stricte et l'unification y est unitaire, donc il existe un algorithme d'unification pour la théorie ω-touffue.

L'algèbre ω-touffue revient à introduire dans les termes de premier ordre des tuples infinis réguliers à l'infini avec un niveau d'imbrication quelconque. Du point de vue des termes autorisés le contructeur de ces tuples Π se comporte donc comme un symbole ordinaire. Nous donnons ci-dessous les équations de l'álgèbre ω-touffue à brins étiquettés qui consistent à superposer les deux algèbres précédentes.

A.3 Algèbre ω-touffue à brins étiquettés

Soit $\mathcal{L}$ un ensemble dénombrable d'étiquettes. Nous prendrons maintenant pour $\mathcal{W}$ l'ensemble des suites finies à valeur dans $\bar{\mathcal{P}}$ que nous complétons en une application $\bar{\mathcal{P}}^N$ en ajoutant une infinité

de ϵ. Nous identifions la suite vide avec ϵ. La taille d'une suite est la somme des cardinaux de ses éléments. L'ordre partiel $\leq$ sur ces mots, est le produit infini de l'ordre d'inclusion sur les ensembles, c'est-à-dire $u \leq v$ si et seulement si $u_k \subset v_k$ pour tout entier k.

Nous considérons initialement un alphabet $\mathcal{C}$ et un ensemble de variables $\mathcal{V}$ signés sur un ensemble $\mathcal{K}$, et l'algèbre $\mathcal{T}(\mathcal{V},\mathcal{C})$. Pour chaque puissaance de $\mathcal{W}$,

- à chaque symbole f de $\mathcal{C}$ de signature ζ, nous associons un symbole $\lfloor\rfloor_f^u$ de signature ζ^u.
- nous introduisons une copie $\mathcal{V}^u$ de $\mathcal{V}$ de puissance (u).

Et pour chaque sorte ι de $\mathcal{K}$,

- nous créons la famille des symboles lieurs $(a * i)_{@}^{u}$ de signature

$$(i\ 0)^u \otimes (i\ a.u_k)^u \Rightarrow (i\ u_k)^u$$

- un symbole passif $(*i)_{\Pi}^{u}$ de signature $(i\ \emptyset)^u \Rightarrow (i\ 0)^u$.

Nous notons respectivement $\mathcal{K}'$, $\mathcal{C}'$ et $\mathcal{V}'$ les réunions de toutes les sortes, symboles variables et.

Définition A.1 L'algèbre *ω-touffue à brins étiquetés* sur $\mathcal{T}$ est l'algèbre libre $\mathcal{T}'(\mathcal{C}',\mathcal{V}')$ signée sur $\mathcal{K}'$ munie des axiomes

$$\left\lfloor {i\ P} \right\rfloor_f^u \left(\alpha_i \left\lceil {*i\ P} \right\rceil^u \beta_i\right)_{i\in[1,p]} \doteq \left(\left\lfloor {i\ 0} \right\rfloor_f^u (\alpha_i)_{i\in[1,p]}\right) \left\lceil {*i\ P} \right\rceil^u \left(\left\lfloor {i\ a.P} \right\rfloor_f^u (\beta_i)_{i\in[1,p]}\right) \qquad (a{\uparrow}i)_{\,f}$$

$$\alpha \left\lceil {a*i\ P} \right\rceil^u (\beta \left\lceil {b*i\ a.P} \right\rceil^u \gamma) \doteq \beta \left\lceil {b*i\ P} \right\rceil^u (\alpha \left\lceil {a*i\ b.P} \right\rceil^u \gamma) \qquad (a, b * i)_{\,@}$$

$$\left(\alpha \left\lceil \begin{matrix} a*i & P \\ j & 0 \end{matrix} \right\rceil^u \beta\right) \left\lceil \begin{matrix} i & P \\ a*j & Q \end{matrix} \right\rceil^u \left(\gamma \left\lceil \begin{matrix} a*i & P \\ j & b.Q \end{matrix} \right\rceil^u \delta\right) \doteq \left(\alpha \left\lceil \begin{matrix} i & 0 \\ b*j & Q \end{matrix} \right\rceil^u \gamma\right) \left\lceil \begin{matrix} a*i & P \\ j & Q \end{matrix} \right\rceil^u \left(\beta \left\lceil \begin{matrix} i & a.P \\ b*j & Q \end{matrix} \right\rceil^u \delta\right)$$

$$(a{\uparrow}i, b{\downarrow}j)_{\,@}, i \not\equiv j$$

□

Exemple A.3 Nous utilisons la notation simplifiée des types enregistrements et nous laissons les sortes et les puissances implicites. Soit σ le terme $\Pi(\Pi(\alpha))$. On obtient le terme τ en extrayant le brin a de la touffe externe puis le brin b de la touffe extraite :

$$\Pi(a : \Pi(b : \beta_{ab}\,;\ \beta_{b\infty})\,;\ \Pi\beta_{\infty\infty})$$

Inversement, le terme ρ est obtenu en extrayant le brin b de la touffe interne, puis le brin a de la touffe externe :

$$\Pi(a : \Pi(b : \gamma_{ab}\,;\ \gamma_{b\infty})\,;\ \Pi(b : \gamma_{a\infty}\,;\ \gamma_{\infty\infty}))$$

Si l'on forme τ' en extrayant le brin b de $\beta_{\infty\infty}$, on obtient

$$\Pi(a : \Pi(b : \beta_{ab}\,;\ \beta_{b\infty})\,;\ \Pi(b : \beta_{b\infty}\,;\ \beta'_{\infty\infty}))$$

qui est égal par l'équation de distributivité médiane au terme ρ à un renommage près.

Nous ne ferons pas l'étude de cette théorie.

Annexe B

Une maquette simple, rapide et complète

Cette annexe décrit une implantation d'un algorithme de synthèse de types. Elle est indépendante, mais utilise essentiellement le vocabulaire et les résultats des chapitres 2 et 3. Tous les programmes sont exécutables dans le langage CAML, version 2-6, dans leur ordre d'apparition[1]. Ils forment une maquette de synthétiseur de types pour un noyau du langage ML. On trouvera

- Une grammaire pour les expressions de type. Elle est bien sûr écrite en CAML sous la forme d'un analyseur syntaxique réversible. On en obtiendra donc un imprimeur de façon automatique. Cette grammaire produit une représentation superficielle des expressions de type. Elle est accompagnée de fonctions de transformation entre la représentation superficielle des expressions de types et leur représentation profonde.
- Une description de la représentation choisie pour les structures de données et des fonctions immédiates de normalisation de ces structures.
- Le cœur du système, à savoir les fonctions
  - `simplify` qui réduit par collision, décomposition et fusion un système de multi-équations,
  - `check_order` qui vérifie que l'ordre de liaison d'un système de multi-équations est strict,
  - `balance_at` et `balance_all` qui équilibrent un système de multi-équations et effectuent la généralisation,
  - `instance` qui calcule une instance d'un système de multi-équations génériques.
- Une structure de donnée pour les expressions du langage. Une structure enrichie avec des pointeurs arrière des variables vers leur occurrence liante et avec des emplacements réservés pour les annotations de typage. L'environnement de typage sera codé directement dans les expressions enrichies.
- La fonction `type` du chapitre 3 appliquée aux structures de données présentes.
- Une grammaire irréversible pour les expressions du langage. Nous n'aurons donc pas d'autre imprimeur pour ces expressions que celui de leur syntaxe abstraite superficielle.
- Un environnememt d'interaction avec l'utilisateur.

B.1 Représentation des types

B.1.1 Syntaxe abstraite superficielle des expressions de types

La syntaxe abstraite superficielle des expressions de types est définie par le type concret suivant :

[1] Les programmes écrits avec une marge droite décalée et en plus petits caractères sont des illustrations ou des exemples mais ne font pas partie de l'algorithme.

```
type Texpr
    = Tvar of string
    | Tarrow of Texpr * Texpr
    | Tpair of Texpr * Texpr
    | Tsymbol of string
    | Twith of Texpr * (Texpr * Texpr) list;;
```

B.1.2 Grammaire des expressions de types

Nous utiliserons la syntaxe concrète suivante pour écrire des expressions de type.

```
grammar for values Texpr =

rule entry Texpr =
    parse [<hv 0> Type u; \-; Equations E] -> Twith(u,E)
| Type u -> u

and Equations =
    parse [<v 1> [<hov 2> "with"; Equation e];
            ( * (parse \-; [<hov 1> "and"; Equation e] -> e)) E ] -> e::E

and Equation =
    parse -; "'"; Ident v; -; "="; -; Type u -> Tvar v,u

and Type =
    parse [ Type1 u; \-; "->"; \-; Type v ] -> Tarrow(u,v)
| Type1 u -> u

and Type1 =
    parse [ Type1 u; \-; "*"; \-; Type1 v ] -> Tpair(u,v)
        | Type0 u -> u

and Type0 =
    parse Ident c -> Tsymbol c
        | "'"; Ident v -> Tvar v
        | "("; Type u; ")" -> u;;

let parse_Texpr = (Texpr "Texpr").Parse_raw;;
```

B.1.3 Imprimeur des expressions de types

Cette grammaire est inversible et permet de produire automatiquement un imprimeur de types à partir de leur syntaxe abstraite superficielle. L'indentation est obtenue gràce aux annotations d'impression mises à cet effet dans la grammaire.

```
printer Texpr =
delimitor "\""
rule entry Texpr =
    print Twith (u,E) -> [<hv>Type u; \-; Equations E]
        | u -> Type u

and  Equations =
    print e::E
           -> [<v 1>[<hov 2>Literal "with"; Equation e];
                    * (print e -> \-; [<hov 1>Literal "and"; Equation e]) E]

and  Equation =
    print (Tvar v,u)
```

```
          ->  -; Literal "'"; IDENT v;  -; Literal "=";  -; Type u

and  Type =
    print Tarrow (u,v) -> [Type1 u; \-; Literal "->"; \-; Type v]
        | u -> Type1 u

and  Type1 =
    print Tpair (u,v) -> [Type1 u; \-; Literal "*"; \-; Type1 v]
        | u -> Type0 u

and  Type0 =
    print Tsymbol c -> IDENT c
        | Tvar v -> Literal "'"; IDENT v
        | u -> Literal "("; Type u; Literal ")"
;;

let print_Texpr = Texpr;;
```

B.2 Représentation interne des types

B.2.1 Multi-équations, variables, et types : une représentation uniforme

La structure de donnée fondamentale est celle de système de multi-équations dans la théorie hiérarchique. Afin de pouvoir effectuer le calcul des fonctions hiérarchiques de façon incrémentale, nous utiliserons des systèmes contraints. En fait, seulement les systèmes contraints complètement décomposés auront une structure de donnée fixée. Les systèmes non complètement décomposés sont des contraintes d'unifications qui attendent d'être résolues. Ce sont donc des objets éphémères, qui sont maintenus dans diverses structures, paires ou listes essentiellement, ou bien directement dans l'environnement d'exécution.

Les systèmes contraints complètement décomposés contiennent des multi-équations de la forme

$$\alpha_1 \doteq \ldots \doteq \alpha_p \doteq \sigma \downarrow p \uparrow q \tag{1}$$

Afin de garantir qu'une variable soit au moins dans une multi-équation du système, la création d'une variable sera accompagnée de la création d'une multi-équation réduite à cette seule variable. On pourra donc toujours identifier une variable et la multi-équation avec laquelle elle a été créée.

```
type variable == multi_equation
```

La structure d'une multi-équation est

```
and multi_equation = {name: num ; mutable state: state}
```

où le champ `name` identifie totalement la multi-équation. Lors d'une fusion, deux multi-équations sont transformées en une seule. L'une des deux multi-équations devra donc mourir. Le champ `state` indique si une multi-équation est vivante ou morte.

```
and state  = Alive of baby
           | Dead of multi_equation
```

Une multi-équation morte pointe vers la multi-équation qui l'a tuée par fusion. Par contre, la partie variable d'une multi-équation ne meurt jamais. Ume multi-équation morte ne peut plus être interprétée que comme une variable, qui appartient à la multi-équation vivante ayant fusionnée avec elle. Si l'on voulait manipuler des systèmes non complètement décomposés, on devrait remplacer la définition précédente par

```
and state  = Alive of baby
           | Dead of term * multi_equation
```

Nous précisons plus loin la structure des termes. Une multi-équation vivante contient l'information suivante

```
and baby = {term: term; mutable down: num ; mutable up: num
           ; mutable mark: num}
```

Les champs `down` et `up` contiennent les états de propagation et de réalisation. Le champ `mark` sera utilisé pour tracer son chemin lors des visites récursives du système de multi-équations. Le champ `terme` contient le terme σ. Nous nous ramènerons toujours à des termes σ de hauteur un, en généralisant éventuellement les multi-équations.

```
and term   = Var
           | Term of symbol * variable list
```

Rappelons qu'une variable est représentée par la multi-équation, éventuellement morte, avec laquelle elle est née. Un symbole est simplement un identificateur muni d'une information d'impression.

```
and symbol = {nickname: num; print: Texpr list -> Texpr}
```

En fait, une multi-équation vivante doit être interprétée comme le multi-ensemble de toutes les multi-équations mortes (c'est-à-dire les variables) qui pointent vers elles, et du terme qu'elle contient. Par exemple, nous pouvons créer deux multi-équations réduites à des variables a et b par

```
let a = {name = 1; state = Alive {term = Var; down = 3; up = 4; mark=0}};;
let b = {name = 2; state = Alive {term = Var; down = 1; up = 2; mark=0}};;
```

puis les fusionner en la structure

```
{name 1 =
    Dead {name = 2; state = Alive {term = Var; down = 1; up = 4; mark=0}};;
```

qui représente la multi-équation $a \doteq b \downarrow 1 \uparrow 2$.

Les multi-équations peuvent également être utilisées pour représenter les types de la façon suivante. A une multi-équation e on associe le sous-système $\mathcal{N}$ des multi-équations qui lui sont liées. Si $\mathcal{N}$ admet une pré-solution $[\mathcal{N}]$, alors on associe à e le terme $\alpha[\mathcal{N}]$ où α est la variable représentée par e. Nous terminons donc la déclaration précédente par

```
and Type = multi_equation;;
```

Nous résumons ci-dessous la structure de donnée des multi-équations, des variables et des types.

```
type Type     == multi_equation
and variable == multi_equation

and multi_equation
             = {name: num ; mutable state: state}

and state    = Alive of baby
             | Dead of multi_equation

and baby     = {term: term; mutable down: num ; mutable up: num;
                mutable mark: num}

and term     = Var
             | Term of symbol * variable list

and symbol   = {nickname: num; print: Texpr list -> Texpr};;
```

B.2.2 La boîte à outils

Une multi-équation peut avoir été tuée par une multi-équation ayant à son tour été tuée. La fonction `devil` se charge d'éliminer les enchaînements de cadavres. Elle retourne la multi-équation vivante au bout de la chaîne. Elle sera donc utilisée chaque fois que l'on voudra accéder à la multi-équation vivante contenant une variable.

```
let rec devil u =
 match u.state
 with Dead v -> let v = devil v in u.state <- Dead v;v
    | _ -> u;;
```

On utilisera également la fonction `baby` qui vérifie qu'une équation est vivante et rend sa vie, ainsi que l'abréviation `baby_devil`.

```
let baby =
 function
   Alive baby -> baby
 | monster -> raise failure "baby";;

let baby_devil u = baby (devil u).state;;
```

Les symboles utilisés dans cette maquette sont la flèche, la paire, et les symboles d'arité nulle. Dans cette version, il ne sera pas possible d'en définir de nouveaux.

```
let arrow   = {nickname = 0; print = (function u::v::_ -> Tarrow(u,v))}
and pair    = {nickname = 1; print = (function u::v::_ -> Tpair(u,v))}
and unit    = {nickname = 2; print = (function _       -> Tsymbol "unit")}
and num     = {nickname = 3; print = (function _       -> Tsymbol "num")}
and bool    = {nickname = 4; print = (function _       -> Tsymbol "bool")};;
```

Il n'y a jamais qu'un seul système actif à la fois ; chaque transformation le remplace par un nouveau de façon irréversible. Ses multi-équations sont toujours triées en fonction de leur état de réalisation.

```
let active_system = ref ([|[]|]: multi_equation list vect);;

let last_variable = ref 0;;

let new_variable (u, n) =
  let v = {name = incr last_variable;
           state = Alive {term = u; down = n; up = n; mark = 0}} in
  (!active_system).(n) <- v::((!active_system).(n)); v;;
```

Nous utiliserons une grammaire construisant des programmes pour créer de nouvelles multi-équations, des variables ou des types.

```
grammar for programs Type =

rule entry Type =
    parse Term u; "["; caml n; "]" -> new_variable (u, n)
        | Type0 u -> u

and Type0 =
    parse "^"; caml0 u -> u
        | "("; Type u ; ")" -> u

and Term =
    parse -> Var
        | Type0 u; "->"; Type0 v -> Term (arrow,[u;v])
        | Type0 u; "*"; Type0 v -> Term (pair,[u;v])
```

```
        | "'"; caml0 C -> Term(C,[])
        | "#"; caml0 u -> u

and caml0 = parse {parse_caml_expr0()} e -> e
and caml  = parse {parse_caml_expr()} e -> e;;
```

B.2.3 De la représentation superficielle vers la représentation profonde

Un type est représenté de façon interne par une multi-équation dont il est une pré-solution. Afin d'obtenir un système dont chaque multi-équation ne comporte que des petits termes on crée une variable α pour chaque sous-terme σ et on forme l'équation $\alpha \doteq \sigma$. L'état d'une multi-équation-variable a pour valeur initiale le degré de l'ensemble dans lequel la multi-équation est construite.

```
let Texpr_to_Type n t =
 let table = ref [] in
 Texpr_to_Type t
 where rec Texpr_to_Type =
  function Tvar s ->
            (assoc s !table ?
             let u = << [n] >> in table:=(s,u)::!table; u)
         | Tarrow (s,t) ->
            << ^(Texpr_to_Type s) -> ^(Texpr_to_Type t) [n] >>
         | Tpair (s,t) ->
            << ^(Texpr_to_Type s) * ^(Texpr_to_Type t) [n] >>
         | Tsymbol c ->
            << '(match c
                  with "num"    -> num
                     | "bool"   -> bool
                     | "unit"   -> unit
                     | _ -> failwith "Texpr_to_Type") [n] >>
         | Twith(t,E) ->
             do_list
               (fun (v,u) ->
                    let v = devil (Texpr_to_Type v) in
                    v.state <- Dead (Texpr_to_Type u))
               E;
             Texpr_to_Type t;;
```

B.2.4 Retour vers la représentation superficielle

Pour obtenir un imprimeur à partir de la représentation interne des types sous forme de multi-équations, il nous faut donc inverser cette fonction à la main. Il faudra pour cela générer de jolis noms de variables à partir de leurs noms internes.

```
let (reset_namer, namer) =
 let alphabet = ["a"; "b"; "c"; "d"; "e"; "f"; "g"; "h"] in
 let history = ref ([]: (num * string) list) in
 let names = ref alphabet in
 let index = ref 0 in
 (fun () -> names := alphabet; index := 0; history := [];()),
  fun v ->
   (assq v !history ?
    let rec name() =
      match !names
       with h::t -> names:=t; h
          | _    ->
            names := map (fun s -> s^string_of_num !index) alphabet;
```

```
            incr index; name() in
    let n = name() in history:= (v,n)::!history; n);;
```

L'imprimeur doit être capable d'écrire des types récursifs, pour rapporter les cas d'erreurs. Cela permettra également de traiter le cas des arbres rationnels par un simple inversement de drapeau. Nous utilisons les marques pour détecter les cycles.

```
let mark = ref 0;;

exception cycle of multi_equation;;

let Texpr_of_Type U =

  incr mark;
  let E = ref[] in

  let rec Texpr_of_Type U =
   let U = devil U in
   let u = baby U.state  in
   if eq(u.mark, -!mark) then (u.mark <- -u.mark; Tvar(namer U.name))
   if eq(u.mark,  !mark) then Tvar(namer U.name)
   else
     begin match u.term
       with Var -> Tvar(namer U.name)
          | _ ->
             u.mark <- -!mark;
             let t =
                 match u.term
                  with Term(C,L) -> C.print (map Texpr_of_Type L)
                     | _ -> failwith "Texpr_of_Type" in
             begin if u.mark < 0
                then u.mark <- 0; t
                else let v = Tvar(namer U.name) in E:= (v,t)::!E; v
             end if
    end match  in

  let t = Texpr_of_Type U in
  match !E
   with [] -> t
      | L -> Twith(t,L);;

let print_Type = print_Texpr o Texpr_of_Type;;
```

B.3 Coeur du système

B.3.1 Simplification d'un système de multi-équations

L'algorithme `simplify` effectue la décomposition éventuellement en détectant les collisions de symboles, et la fusion. Lors de la fusion de deux équations, l'état de propagation de la multi-équation fusionnée est mis à jour à la valeur minimale des états des multi-équations fusionnées. L'une des deux multi-équations est tuée par l'autre.

```
#pragma infix "minimum";;

let prefix minimum = uncurry min;;

exception collision of multi_equation * multi_equation;;
```

```
let rec simplify (U,V) =

 let U = devil U in
 let V = devil V in

 if U.name = V.name then () else

  let u = baby U.state in
  let v = baby V.state in

  begin match u.term, v.term
   with Var, _ ->
          U.state <- Dead V;
          v.down <- u.down minimum v.down

      | _, Var ->
          V.state <- Dead U;
          u.down <- u.down minimum v.down

      | Term(C,L), Term(D,M) ->
          if not eq (C.nickname, D.nickname) then raise collision (U, V);

          if u.down < v.down
             then V.state <- Dead U
             else U.state <- Dead V;

          do_list simplify (combine (L,M))

 end match;
 ();;
```

Un système de multi-équations complètement décomposé admet des solutions seulement si l'ordre y est strict. La fonction `check_order` vérifie que cet ordre est strict sur le sous-système réalisé au degré $n+1$.

```
let check_order n = check_order
where rec check_order U =
  let u = baby_devil U in
  if u.mark < 0 then raise cycle U
  if u.mark < !mark & u.up > n then
  begin match u.term
   with Var -> 0
      | Term(C,L) ->
          u.mark <- - !mark;
          do_list check_order L ;
          u.mark <- - u.mark
  end match;
  ();;
```

B.3.2 Equilibrage partiel et généralisation

La fonction `balance_all` effectue l'équilibrage du système actif au degré $n+1$. L'ensemble des multi-équations de degré $n+1$ est propagé jusqu'aux multi-équations réalisées à un degré inférieur ou égal à n. Cette propagation est effectuée de l'extérieur vers l'intérieur par la fonction `balance` qui parcourt récursivement les multi-équations du système jusqu'à la rencontre d'une multi-équation déjà propagée, ou réalisée à un degré inférieur ou égal à n. La propagation est effectuée pendant la descente récursive. La réalisation est effectuée à la remontée, donc de l'intérieur vers l'extérieur, respectant ainsi les conditions permettant de ne pas revenir sur la partie déjà visitée.

Au retour de la fonction `balance_at` les multi-équations sont réalisées au degré n. La généralisation peut donc être faite pendant la remontée. Seules seront généralisées les multi-équations qui participent effectivement à l'assertion augmentant le contexte de typage. Une multi-équation généralisée n'a plus besoin de ses variables d'état ; on utilise le champ `down` pour marquer les variables génériques, en lui donnant une valeur négative. Sa valeur absolue indique un majorant du nombre de multi-équations génériques qui lui sont liées, et attribue un nom générique à chaque variable du système qui est en train d'être généralisé. Ces informations seront utiles pour le calcul des instances. Si l'on prend soin de visiter d'abord toutes les multi-équations qui doivent être généralisées, le majorant sera très raisonnable[2].

Pour les multi-équations réalisées au degré $n+1$ et n'ayant pas besoin d'être généralisées, on vérifiera simplement qu'elles ne participent pas à un cycle, ce qui correspondrait à un échec dans le typage. Les cycles ne sont refusés que si l'on n'admet pas les arbres rationnels. Dans le cas où les arbres rationnels seraient autorisés, il suffirait de changer la valeur booleenne de `recursive_types`.

Enfin, on redistribue l'ensemble des multi-équations réalisées à un degré inférieur ou égal à n dans le système actif.

```
let recursive_types = ref false;;

let balance_at  n =
 let generic_name = ref 0 in balance
 where rec balance k U =

  let u = baby_devil U in

  if u.mark < 0 then (if !recursive_types then k else raise cycle U)
  if u.mark < !mark then

  begin do
    begin match u.term
      with Var ->
             u.up <- u.down minimum k

         | Term(C,L) ->
             begin if u.up > n then
               u.mark <- -!mark;
               let balance = balance (u.down minimum k) in
               u.up <- it_list (fun k U -> max k (balance U)) 0 L
             end if

     end match;

     if u.down > u.up then u.down <- u.up;
     u.mark <- !mark;
     if u.up > n then u.down <- decr generic_name;
     u.up
   end do

  else u.up;;

let balance_all n entries fresh N =

  let sorted = vector vect_length N + 1 of [] in

  let sort U =
    match U.state
     with Dead _ -> ()
```

[2] Un majorant trop grossier augmenterait inutilement le calcul de ses instances.

```
        | Alive u -> sorted.(u.down) <- U::sorted.(u.down);() in

  do_list sort fresh;

  incr mark;
  let balance = balance_at n in
  do_vect_i

    (fun i ->

         begin if i <= n
            then
              do_list
                (let balance_i = balance i in
                 fun ({state = Alive u; _} as U) ->
                        balance_i U;
                        N.(u.up) <- U::N.(u.up)
                   | _ -> failwith "balance_all")

            else
              do_list (balance (n+1)) entries;
              let check =  check_order n in
              do_list
                (fun ({state = Alive u; _} as U) ->
                        if u.up > n
                           then !recursive_types or check U
                           else N.(u.up) <- U::N.(u.up);
                        ()
                    | _ -> failwith "balance_all")
     end if)

    sorted;;
```

B.3.3 Calcul d'une instance

La fonction `instance` appliquée à un entier n et un type générique σ calcule une instance de σ dans $\mathcal{T}_n$. L'état de propagation d'une multi-équation générique est négatif et indique un majorant du nombre de multi-équations génériques qui lui sont liées. Si l'état de réalisation d'une multi-équation est positif, alors aucune des multi-équations qui lui sont liées n'est générique, et la multi-équation est donc égale à son instance.

```
type filled = Empty | Filled of multi_equation;;

let instance k U =
  let U = devil U in
  let u = baby U.state in
  if u.down >= 0 then U else
  let table = vector (-u.down) of Empty in
  instance U

 where rec instance V =
  let V = devil V in
  let v = baby V.state in
  if v.up < u.up
     then V
     else
        match table.(-v.down-1)
```

```
        with Filled W -> W
           | Empty ->
             let W = << [k] >> in
             table.(-v.down-1) <- Filled W;
             W.state <- Alive
               {term =
                  begin match v.term
                   with Var ->  Var
                      | Term(C, L) -> Term(C, map instance L)
                  end match;
                down = k; up = k; mark = 0};
             W;;
```

B.4 Expressions du langage

B.4.1 Représentation superficielle

La syntaxe abstraite superficielle des expressions du langage est définie par le type concret suivant :

```
type M
    = Mexpr of Mexpr
    | Mdecl of bool * string * Mexpr
    | Mexit

and  Mexpr
    = Mvar of string
    | Mapp of Mexpr * Mexpr
    | Mfun of string * Mexpr
    | Mlet of bool * string * Mexpr * Mexpr
    | Mif of Mexpr * Mexpr * Mexpr
    | Mpair of Mexpr * Mexpr
    | Mconst of Mconst
    | Mstraint of Mexpr * Texpr

and Mconst
    = Mnum of num
    | Mbool of bool
    | Munit;;
```

B.4.2 Grammaire du langage

```
grammar for values M =

precedences
    right "->";
    right ",";

rule entry M =
    parse "let"; Rec r; Pat p; Let d; ";;" -> Mdecl (r,p,d)
        | Expr e; ";;" -> Mexpr e
        | "exit" -> Mexit

and Texpr =
    parse {match parse_Texpr() with MLquote (dynamic (u:Texpr)) -> u} u -> u

and Let =
    parse Pat p; Let m -> Mfun (p, m)
```

```
     | "="; Expr e -> e

and Rec =
    parse -> false
        | "rec" -> true

and Fun =
    parse Pat p; Fun m -> p, Mfun m
     | Pat p; "->"; Expr e -> p,e

and Expr =
    parse "fun"; Fun m -> Mfun m
        | "let"; Rec b; Pat p; Let d; "in"; Expr e -> Mlet (b,p,d,e)
        | "if"; Expr2 b; "then"; Expr2 u; "else"; Expr2 v -> Mif (b,u,v)
        | Expr2 e -> e

and Expr2 =
    parse Expr1 e1; ","; Expr2 e2 -> Mpair (e1,e2)
        | Expr1 e -> e

and Expr1 =
    parse Expr1 e1; Expr0 e2 -> Mapp (e1,e2)
        | Expr0 e -> e

and Expr0 =
    parse Cexpr c -> Mconst c
        | IDENT v -> Mvar v
        | "("; Expr e; ":"; Texpr t; ")" -> Mstraint(e,t)
        | "("; Expr e; ")" -> e

and Cexpr =
    parse NUM n -> Mnum n
        | BOOL b -> Mbool b
        | "("; ")" -> Munit

and Pat =
    parse IDENT v -> v

;;

let parse = (M "M").Parse;;

#set default grammar Type;;
```

B.4.3 Vers une représentation profonde

Nous transformerons la représentation abstraite superficielle en une structure plus riche, sur laquelle il sera possible de coder l'environnement de typage. En fait cette représentation sera à l'issue du processus de typage le programme initial où chaque variable aura été annotée par son type. la récursion sera remplacée par une application du combinateur de point-fixe.

```
type Lexpr
    = Lvar of Lvar
    | Lapp of Lexpr * Lexpr
    | Lfun of Lvar * Lexpr
    | Llet of Lvar * Lexpr * Lexpr
    | Lpair of Lexpr * Lexpr
    | Lconst of Mconst
```

```
    | Lstraint of Lexpr * Texpr

and Lvar = {surname : string; mutable annotation : multi_equation};;

let make_table tab =

    let get s =
     let key = hash_atom s mod (vect_length tab) in
     assoc s tab.(key) in

    let store s arrow =
     let key = hash_atom s mod (vect_length tab) in
     tab.(key) <- (s,arrow)::(filter_neg (function t,_ -> s=t) tab.(key));() in

get, store;;

let global_values = vector 50 of ([]: (string * multi_equation) list);;

let get_value, store_value = make_table global_values;;

let empty = << [0] >>;;

let pat_of_var x = {surname = x; annotation = empty} ;;
```

La transformation est réalisée par la fonction `Lin`. Elle ajoute des pointeurs des variables locales ou globales vers leurs occurrences liantes. Elle échoue lorsqu'elle rencontre une variable libre.

```
exception unbound of string;;

let rec Lin values = Lin_values
 where rec Lin_values e =
 match e
  with Mvar s ->
          Lvar (assoc s values ? {surname = s; annotation = get_value s}) ?
          raise unbound s

     | Mapp (e,e') ->
          Lapp (Lin_values e, Lin_values e')

     | Mfun (x,e) ->
          let p = pat_of_var x in
          Lfun (p, Lin ((x,p)::values) e)

     | Mlet (b,x,e,e') ->
          if b
             then
               let e = Mapp (Mvar "&fix", Mfun (x, e)) in
               Lin_values (Mlet (false, x, e, e'))

             else
               let e = Lin_values e in
               let p = pat_of_var x in
               let e' = Lin ((x,p)::values) e' in
               Llet (p, e, e')

     | Mif (b,e,e') ->
          Lin_values (Mapp (Mapp (Mapp (Mvar "&if", b), e), e'))
```

```
    | Mconst c ->
        Lconst c

    | Mpair (e,e') ->
        Lpair (Lin_values e, Lin_values e')

    | Mstraint (e,t) ->
        Lstraint (Lin_values e, t);;
```

B.5 Simplification des contraintes de typage

Le passage par une syntaxe profonde simplifie le typage. En particulier l'étape précédente assure que l'expression à typer est close dans l'environnement global. La fonction `type_expr` est une adaptation immédiate de la version de `type` donnée dans le chapitre 3 aux structures de données présentes.

```
let rec type_expr n =
  let instance_n = instance n in
  type_expr_n
 where rec type_expr_n e =
    match e
     with Lvar x ->
            instance_n x.annotation

        | Lapp (e,e') ->
            let u = type_expr_n e in
            let v = type_expr_n e' in
            let w = << [n] >> in
            simplify (u, << ^v -> ^w [n] >>);
            w


        | Lfun (x, e) ->
            let u = x.annotation <- << [n] >> in
            let v = type_expr_n e in
            << ^u -> ^v [n] >>

        | Llet (x, e, e') ->
            let N = !active_system in
            let N1 = vector vect_length N + 1 of [] in
            do_vect_i (fun i u -> N1.(i) <- u) N;
            active_system := N1;
              let u = type_expr (n+1) e in
              x.annotation <- u;
            active_system := N;
            do_vect_i (fun i _ -> N.(i) <- N1.(i)) N;
            balance_all n [u] N1.(n+1) N;
            type_expr_n e'

        | Lconst  c ->
        << '(match c
              with Mnum n    -> num
                 | Mbool b   -> bool
                 | Munit     -> unit) [0]>>


        | Lpair(e,e') ->
```

```
            let u = type_expr_n e in
            let v = type_expr_n e' in
            << ^u * ^v [n] >>


        | Lstraint (e,t) ->
            let u = type_expr_n e in
            let v = Texpr_to_Type n t in
            simplify (u,v);
            u;;
```

Le typage d'une expression globale n'est qu'un cas particulier du typage d'une expression locale.

```
#set default grammar Pretty:Exec;;

let error = "Texprchecking error:";;

let  type_global s e=
  begin try
     let e = Lin [] e in
     let p = pat_of_var s in
     let e = Llet (p, e, Lconst Munit) in
     active_system:= [|[]|];
     type_expr 0 e;
     store_value s p.annotation;
     << [<hov 2> [s; " :"]; \-; print_Type {p.annotation}] >>

  with unbound s ->
         << [<hv> error; " unbound variable"; \-; Ident s] >>
     | collision (u,v) ->
         << [<hv> [<h> error; " collision between"]; \-;
                  [<hv> print_Type u; \-; "and"; \-; print_Type v]] >>
     | cycle u ->
         << [<hov> error; " cycle "; \-; print_Type u] >>
  end try;
  active_system:= [|[]|];
();;
```

B.6 Finitions

Nous terminons cette maquette par l'écriture d'un petit environnement agréable, c'est si simple que ce serait dommage de s'en priver.

```
exception exit;;

let step() =
    reset_namer();
    print_newline();
    begin match (match eval_syntax(parse()) with dynamic (x:M) -> x)
      with Mdecl (b,s,e) -> type_global s (if b then Mlet(b,s,e,Mvar s) else e)
         | Mexpr e       -> type_global "it" e
         | Mexit         -> raise exit
    end match;
    print_newline();;

let rec loop() =
 parsing_handler print_syntax_error step();
 loop();;
```

```
let rec top() =
 let prompt = prompt() in
 set_prompt">";
 try loop() with
    exit -> set_prompt prompt
  | _ -> set_prompt prompt reraise;;

#set default grammar Type;;
```

Il sera nécessaire d'initialiser l'environnement global par quelques primitives avant de commencer à jouer.

```
recursive_types:= true;;

step();;

fun  f -> (fun x -> f (x x)) (fun x -> f (x x));;

store_value "&fix" (get_value "it");;

recursive_types:= false;;

step();;

let rec x = (x: bool -> 'a -> 'a -> 'a) in x;;

store_value "&if" (get_value "it");;
```

Lorsque vous aurez exploité toutes les possibilités de cette maquette, vous pourrez tourner la page et essayer l'implantation complète avec des déclarations de types, un mécanisme d'appel par filtrage, et des objets enregistrements ; elle est décrite dans l'annexe suivante.

Annexe C

Une maquette avec des objets enregistrements

Cette annexe reprend le squelette de la maquette précédente et l'étend avec des objets et types enregistrements. Nous verrons que cette extension s'obtient simplement par l'ajout d'opérations de mutation dans l'algorithme de simplification, et bien sûr par l'introduction de primitives sur les enregistrements avec leurs types.

Les objets enregistrements nécessitent de contrôler la formation des expressions de types par des sortes et des puissances. Cette maquette étend donc également les types de ML par des types avec sortes. Afin d'être complètement indépendant de la signature du langage, et de pouvoir essayer différents systèmes de typage des enregistrements, cette extension est faite de façon très générale. Ceci peut paraître compliqué, mais il n'en est rien. Le langage des sortes est indépendant du langage des types, et les sortes n'interviennent qu'au moment de la lecture des expressions de types écrites par l'utilisateur. Les algorithmes de simplification préservant les sortes et les puissances, on peut complètement oublier celles-ci par la suite.

La maquette est également étendue à un langage plus complet, permettant des définitions de types et un appel par filtrage. Ces deux extensions sont liées et permettent de traiter des exemples conséquents.

Chacune de ces extensions est simple, et il ne faut pas se laisser tromper par l'éventuelle difficulté due à leur introduction simultanée. Aussi nous essayerons de rendre compte de cette simplicité en classant chaque programme dans l'une des cinq catégories suivantes :

- Un programme de *récupération* est simplement repris dans la maquette précédente.
- Un programme d'*extension* est obtenu à partir d'un programme de la maquette précédente par une extension immédiate tenant compte des nouvelles constructions, et de l'enrichissement de l'environnement.
- Un programme d'*enregistrement* est spécifique à l'ajout des objets enregistrements.
- Un programme de *sortes* n'effectue que des opérations sur les sortes.
- Un programme de *filtrage* implante les déclarations de types et le typage de l'appel par filtrage.

C.1 Les expressions de types, grammaires et imprimeurs

Cette partie est du code d'extension. La grammaire doit notamment décrire le langage des définitions de types.

```
type Texpr
    = Tvar of string
    | Tarrow of Texpr * Texpr
    | Tpair of Texpr * Texpr
    | Tdot of Texpr * Texpr
```

```
    | Tapp of string * Texpr list
    | Tpi of Texpr
    | Tat of string * Texpr * Texpr
    | Twith of Texpr * (Texpr * Texpr) list

and Tdecl = Tdecl of (string list * string * (string * Texpr) list) list;;


grammar for values Texpr =

rule entry Texpr =
    parse [<hv 0> Type u; \-; Equations E] -> Twith(u,E)
| Type u -> u

and Equations =
    parse [<v 1> [<hov 2> "with"; Equation e];
            ( * (parse \-; [<hov 1> "and"; Equation e] -> e)) E ] -> e::E

and Equation =
    parse -; "'"; Ident v; -; "="; -; Type u -> Tvar v,u

and At =
    parse IDENT s; -; ":"; \-; Type u; ";"; \-; At v -> Tat (s,u,v)
        | Type u -> u

and Type =
    parse [ Type3 u; "."; Type v ] -> Tdot(u,v)
        | Type3 u -> u

and Type3 =
    parse [ Type2 u; \-; "->"; \-; Type3 v ] -> Tarrow(u,v)
        | Type2 u -> u

and Type2 =
    parse [ Type1 u; \-; "*"; \-; Type2 v ] -> Tpair(u,v)
        | Type1 u -> u

and Type1 =
    parse [ Ident C; Args L ] -> Tapp(C, L)
        | "'"; Ident v -> Tvar v
        | "{"; [At r]; "}" -> Tpi r
        | "("; [At u]; ")" -> u

and Args =
    parse -> []
        | \-; "("; Type u; ( * (parse \-; ","; Type u -> u)) L; ")" -> u::L

and args =
    parse -> []
        | \-; "("; "'"; Ident u;
                ( * (parse \-; ","; "'"; Ident u -> u)) L; ")" -> u::L

and entry Decl =
    parse [<v 0> "type"; Body d] -> d

and entry Body =
    parse One d; ( * (parse \-; "and"; -; One d -> d)) L -> Tdecl (d::L)
```

```
and One =
    parse [<hv 0> -; Ident C; args l;
          \-; [<v 0> "="; -; Of h; ( * (parse \-; "|"; -; Of h -> h)) L]]
            -> l,C,h::L

and Of =
    parse Ident s -> s, Tapp("unit",[])
| Ident s; -; "of"; -; Type u -> s,u

;;

let parse_Texpr = (Texpr "Texpr").Parse_raw
and parse_Texpr_decl = (Texpr "Decl").Parse_raw
and parse_Tdecl_body = (Texpr "Body").Parse_raw;;

pp_of_gram_in_file "Texpr" "Texpr_pp";;
load "Texpr_pp";;

let print_Texpr = Texpr
and print_Texpr_decl = Decl;;
```

C.2 Représentation des types

Cette partie comprend des programmes des trois dernières catégories. Le module `label` implante la représentation des étiquettes, dont cette maquette ne dépendra pas ; il correspond à la définition d'un type abstrait `label`. Le module `tools` regroupe diverses fonctions d'usage plus général.

```
module tools;;

let make_table n tab =

    if n > vect_length tab then failwith "make_table";

    let get s =
     let key = n + (hash_atom s) mod (vect_length tab) in
     assoc s tab.(key) in

    let store s arrow =
     let key = n + (hash_atom s) mod (vect_length tab) in
     tab.(key) <- (s,arrow)::(filter_neg (function t,_ -> s=t) tab.(key));() in

get, store;;

end module
 with value make_table;;


module label using
 value make_table :
      num -> ((string * 'a) list vect ->
      (string -> 'a) * (string -> 'a -> unit));;


type label = Lab of num;;

let eq_label (Lab a, Lab b) =  (a = b);;
let lt_label (Lab a, Lab b) =  (a < b);;
```

```
let gt_label (Lab a, Lab b) =  (a > b);;

let labels = vector 50 of ([]: (string * label) list);;
let get_label, store_label = make_table 0 labels;;

let unlabels = ref ([||]: string vect);;
let unlabels_step = 100;;

let store_unlabel (Lab a) s =
  if vect_length !unlabels < a
     then
       (let new_unlabels =
          vector (vect_length !unlabels + unlabels_step)  of "" in
        do_vect_i
          (fun i a -> new_unlabels.(i) <- a)
          !unlabels;
        unlabels := new_unlabels);
  (!unlabels).(a) <- s;;


let last_label = ref 0;;

let label s =
    get_label s ?
    let a = Lab (incr last_label) in
    (store_label s a; store_unlabel a s; a);;

let unlabel (Lab a) = (!unlabels).(a);;

let print_label = print_string o unlabel;;

#printer print_label;;

end module
 with type label;
 value label and unlabel and print_label
   and eq_label and lt_label and gt_label
;;
```

La représentation interne des types n'a pas vraiment changé. La seule différence est la représentation des symboles. Il est en effet essentiel de distinguer les symboles lieurs des autres symboles. Cette distinction se retrouvera dans toutes les fonctions utilisant la représentation des symboles. Il sera aussi utile de préciser la signature de chaque symbole. Nous décrivons d'abord la représentation des puissances et des sortes et les opérations sur ces objets.

C.2.1 Représentation des sortes et leur manipulation

Nous distinguerons les sortes et les puissances qui sont définies par les types concrets suivants :

```
type sort = Sort of string;;

type power = Zero
           | Effective of label list;;
```

Lors de la synthèse des sortes, nous utiliserons des méta-variables de sortes et de puissances. Le mot "méta" signifie simplement que ces variables ne font pas partie du langage des sortes et des puissances, mais permettent quand même de parler de ces objets avant d'en connaître leur valeur.

```
type 'a meta = Free of 'a free | Bound of 'a meta ref
and 'a free = Undef | Defined of 'a;;

type meta_sort == sort meta ref
and meta_power == power meta ref;;

type sig == (meta_sort * meta_power) list * (meta_sort * meta_power);;

grammar for programs meta =
rule entry meta =
    parse "_" -> ref (Free Undef)
        | {parse_caml_expr()} e -> ref (Free (Defined e));;
```

La boîte à outils ressemble fort à celle des multi-équations. Les fonctions `cut` et `free` ne sont que les fonctions `devil` et `baby` sous des visages différents.

```
let rec cut u =
 match !u
  with Bound v -> let v = cut v in u := Bound v;v
     | _ -> u;;

let free u =
  match !(cut u)
   with Free u -> u
      | Bound _ -> failwith "free";;

let satisfy meta (u,v) =
  if not eq (cut u, cut v)  then
     begin match free u, free v
      with Undef, _ -> u:= Bound v
         | _, Undef -> v:= Bound u
         | Defined a, Defined b -> u:= Bound v; meta (a,b)
     end match;
  ();;
```

Nous regrouperons les erreurs de contraintes de sortes dans le type concret et les exceptions suivantes.

```
type sig_error = Esort of sort * sort
               | Epower of power * power
               | Eadd of label * power
               | Eother of string
;;

exception sig;;

exception sig_error of sig_error;;

exception report of sig_error * Texpr;;

let signal error d = raise sig_error (error d)
and report error expr = raise report (error, expr);;
```

Nous pourrons résoudre les contraintes de sortes et de puissances à l'aide de la fonctionnelle `satisfy`. La fonction `add_power` ajoute une étiquette à une puissance, qui doit être une puissance de touffe complètement définie, et ne doit pas contenir l'étiquette ajoutée.

```
let satisfy_power =
  let rec eq_power =
    function
          a::A, b::B  -> eq_label (a,b) & eq_power (A,B) or (raise sig)
```

```
        | [],[] -> true
        | _ -> raise sig in
  let power P =
    match P
     with Zero, Zero -> true
        | Effective A, Effective B -> eq_power (A,B)
        | _ -> raise sig in
  satisfy (fun d -> try power d with sig -> signal Epower d);;

let satisfy_sort =
  let sort (Sort s, Sort t) = (s=t) or (raise sig) in
  satisfy (fun d -> try sort d with sig -> signal Esort d);;

let add_power a A =
  begin match free A
   with Defined (Effective A as P) ->
          begin try <:meta< Effective (add A) >>
           with sig -> signal Eadd (a,P)
          end try
      | Defined Zero -> signal Eadd (a, Zero)
      | Undef -> failwith "unbound power"
  end match
  where rec add =
    function b::B as A ->
               if eq_label (a,b) then (raise sig)
               if gt_label (a,b)
                  then a::A
                  else b::add B
           | [] -> [a];;
```

Nous utiliserons ces fonctions ci-dessous pour synthétiser les contraintes de sortes.

C.2.2 Représentation des multi-équations

La représentation des symboles est maintenant plus riche :

```
type symbol  = At of label
             | Other of other

and other    = {nickname: num; sig: sig; print: infix}

and infix = Infix of Texpr * Texpr -> Texpr  | Prefix of string;;
```

Mais la représentation des types est inchangée. Il en découle que toutes les fonctions n'allant pas lire à l'intérieur des symboles seront également inchangées, notamment la plupart des fonctions du cœur du système.

```
type Type    == multi_equation
and variable == multi_equation

and multi_equation
             = {name: num ; mutable state: state}

and state    = Alive of baby
             | Dead of multi_equation

and baby     = {term: term; mutable down: num ; mutable up: num;
                mutable mark: num}
```

```
and term     = Var
             | Term of symbol * variable list;;
```

Nous récupérons les vieux outils.

```
let rec devil u =
 match u.state
 with Dead v -> let v = devil v in u.state <- Dead v;v
    | _ -> u;;

let baby =
 function
   Alive baby -> baby
 | monster -> raise failure "baby";;

let baby_devil u = baby (devil u).state;;
```

Comme l'utilisateur pourra déclarer de nouveaux symboles, nous aurons besoin d'une table des symboles globaux. Il n'est pas prévu qu'il puisse définir de nouvelles sortes, mais il pourra toujours le faire en ajoutant à la main des symboles primitifs ayant les sortes désirées. Ce serait immédiat de lever cette restriction. Il faudrait simplement augmenter le langage avec des déclarations de sortes.

```
exception unbound of string * string;;

let global_symbols = ref ([]: (string & other) list);;

let get_symbol s =
  try assoc s !global_symbols
  with failure _ -> raise unbound ("type constructor", s)

and store_symbol (_,c as u) = global_symbols:=u::!global_symbols; c;;
```

Nous distinguerons les symboles touffus des symboles passifs, simplement par le signe de leur `nickname`. Cette distinction ne sera nécessaire que pendant la synthèse des sortes. Nous créons quelques sortes et symboles primitifs.

```
let nickname = ref 0
and pi_name = ref 0;;

let create b (name, print, sig) =
  let c = {nickname =
             if b then incr nickname else decr pi_name;
           sig = sig; print = print} in
  store_symbol (name,c);;

let usual  = <:meta< Sort "usual" >>
and field  = <:meta< Sort "field" >>
and flag   = <:meta< Sort "flag" >>;;

let zero   = <:meta< Zero >>
and empty  = <:meta< Effective[] >>;;

let pi     = create false
    ("&pi",    Prefix "&pi",  [field, empty], usual, zero)
and arrow  = create false
    ("&arrow", Infix Tarrow, [usual, zero; usual, zero], usual, zero);;

let pair   = create false
    ("&pair",  Infix Tpair,  arrow.sig)
and dot    = create true
    ("&dot",   Infix Tdot,   [flag, zero; usual, zero], field, zero );;
```

```
let [Pi; Arrow; Pair; Dot] = map Other [pi; arrow; pair; dot];;
```

Nous étendons la grammaire permettant de construire des multi-équations.

```
let active_system = ref ([|[]|]: multi_equation list vect);;

let last_variable = ref 0;;

let new_variable (u, n) =
  let v = {name = incr last_variable;
           state = Alive {term = u; down = n; up = n; mark = 0}} in
  (!active_system).(n) <- v::((!active_system).(n)); v
;;

grammar for programs Type =

rule entry Type =
    parse Term u; "["; caml n; "]" -> new_variable (u, n)
        | Type0 u -> u

and Type0 =
    parse "^"; caml0 u -> u
        | "("; Type u ; ")" -> u

and Term =
    parse -> Var
        | Term1 x -> Term x
        | "#"; caml0 e -> e

and Term1 =
    parse caml0 s; ":"; Type0 u; ";"; Type0 v -> At s,[u;v]
        | "'"; caml0 C  -> C, []
        | "{"; caml L; "}"; "'"; caml0 C -> C, L
        | Type0 u; ( * (parse Type0 u -> u)) L; "'"; caml0 C -> C, u::L
        | Type0 u; (parse "->" -> Arrow | "*" -> Pair | "." -> Dot) C;
          Type0 v -> C, [u;v]

and caml0 = parse {parse_caml_expr0()} e -> e
and caml  = parse {parse_caml_expr()} e -> e;;
```

C.2.3 Synthèse des contraintes de sortes

Le langage des types permet à l'utilisateur de construire des expressions dont les contraintes de sortes peuvent être incohérentes. Il faut donc synthétiser les sortes pour les expressions de types. Il y a deux cas bien différents dans lesquels l'utilisateur écrit des expressions de types.

Le premier est l'écriture de contraintes de types dans un programme. Une contrainte de type est toujours de la sorte `usual` et de puissance `zero`, car elle porte sur une sous-expression du langage, qui a toujours les caractéristiques précédentes. En effet aucune des primitives ne permet de couper un enregistrement en morceaux. Ceci n'est pas fondamental et le système de typage permettrait de le faire ; il faudrait seulement s'assurer que l'on saura compiler les morceaux extraits. Tous les symboles sont donc connus avec leur signatures, et il suffit simplement de propager les contraintes de la racine vers les feuilles. Cette partie ne nécessite pas l'utilisation des méta-variables, mais leur présence permettra à ce vérificateur de devenir un synthétiseur dans la partie suivante.

```
let Texpr_to_Type_with Type_var n  = Type
 where rec Type (S,A as P) u =

 let Type_symbol (m,c) =
```

```
   let C = get_symbol c in
   let sig, T, B = C.sig in
   satisfy_sort (S,T);
   if C.nickname < 0 then satisfy_power (A,B);
   let Type =
     if C.nickname < 0 then Type else (function (T,B) -> Type (T,A))  in
   << {map2 Type sig m ? signal Eother "arity"}  '(Other C) [n] >> in

  try
     match u
      with Tvar u -> Type_var (u, P)
         | Tpi u -> Type_symbol ([u], "&pi")
         | Tarrow (u,v) -> Type_symbol ([u;v], "&arrow")
         | Tpair (u,v) -> Type_symbol ([u;v], "&pair")
         | Tdot (u,v) -> Type_symbol ([u;v], "&dot")

         | Tat (a, u, v) ->
             let a = label a in
             << a : ^(Type (S, zero)  u);
                    ^(Type (S, add_power a A) v) [n]>>

         | Tapp (C,L) ->
            Type_symbol (rev L, C)

         | Twith(t,E) ->
             do_list
               (fun (v,u) ->
                    let V = devil (Type P v) in
                    V.state <- Dead (Type P u))
               E;
             Type P t

  with sig_error e -> report e u;;

let Texpr_to_Type n =

  let dictionary = ref [] in

  let Type_var (s, S,A) =
    (let u, T,B = assoc s !dictionary in
    satisfy_sort (S,T); satisfy_power (A,B); u) ?
    let u = << [n] >> in
    dictionary:= (s, u, S, A)::!dictionary; u in

  Texpr_to_Type_with Type_var n (usual, zero)
;;
```

Le second cas d'utilisation d'expressions de types par l'utilisateur est bien sûr pour la déclaration de nouvelles structures de données. Du point de vue théorique, cela ne pose pas de problème, puisque l'on peut toujours supposer que les symboles que demande l'utilisateur existaient depuis le début du monde, mais n'étaient pas connus de l'utilisateur. Une déclaration de structure de donnée n'est rien d'autre que l'ajout d'une constante dans le dictionnaire de l'utilisateur. Mais certaines définitions n'ont pas de sens, parce ce que leurs contraintes de puissance ou de sorte ne sont pas satisfiables. Une requête de l'utilisateur devra donc être analysée avant d'être ajoutée dans le dictionnaire des symboles.

La déclaration d'un type concret n'a d'intérêt que par la création simultanée de fonctions de construction et de destruction de ce type. Les objets de puissance non nulle étant inaccessibles

à l'utilisateur, nous exigerons que les types créés soient de la puissance `zero` et de la sorte `usual`. Encore une fois, ce choix n'est pas dû au système de typage, mais à son utilisation dans le langage. Il serait possible d'autoriser des types concrets d'une autre sorte ; il faudrait alors donner à l'utilisateur un langage pour les sortes. Les symboles introduits sont des symboles passifs ; ils construisent des types de puissance ϵ à partir de types de puissance éventuellement

Une décalration de type est récursive, c'est-à-dire que les types déclarés, dont on cherche la signature, peuvent apparaître dans le corps de la déclaration. Il n'est plus possible comme pour la vérification de la bonne formation des expressions de types de se contenter de vérifier les sortes, mais il faut les synthétiser. Cela justifie l'introduction de variables de sortes (et de puissances). A la fin du calcul toutes ces variables qui ont été instanciées sont soient des variables d'un type récursif, soit des variables superflues. On les contraint à être de la sorte *usuel* et de la puissance *Zero*.

```
exception rebound of string * Tdecl;;

let type_type n (Tdecl bodies) =
  let old_global_symbols = !global_symbols in
 try
  let rec sig =
    function (s::args) ->
               if mem s args
                  then raise rebound (s, Tdecl bodies)
                  else (<:meta< _ >>, <:meta< _ >>)::(sig args)
           | [] -> [] in

  let bodies =
    map
      (function args, name, _ as body ->
         let C =
            {nickname = decr pi_name; sig = sig args, usual, zero;
             print = Prefix name} in
       store_symbol (name, C), body)

       bodies in

  let type_decl constructors (C, args, name, definition) =

      let dictionary = map2 (fun s p -> s, << [n] >>, p) args (fst C.sig) in

      let v = << {map (fun (_,u,_) -> u) dictionary} '(Other C) [n] >> in

      let Texpr_to_Type =
        let Type_var (s, S, A) =
          (let u, T,B = assoc s dictionary in
          satisfy_sort (S,T); satisfy_power (A,B); u) ?
          raise unbound ("type variable",s) in
        Texpr_to_Type_with Type_var n (usual, zero) in

      it_list
         (fun constructors (C, t) ->
              (C, << ^(Texpr_to_Type t) -> ^v [n] >>)::constructors)
          constructors
          definition in

  let declaration = it_list type_decl [] bodies in

  let constraint (S,A) s =
    begin match free S
```

```
     with Defined _ -> ()
        | Undef -> satisfy_sort (S, usual)
    end match;
    begin match free A
     with Defined _ -> ()
        | Undef -> satisfy_power (A, zero)
    end match in

  do_list
    (function C, args, _ -> do_list2 constraint (fst C.sig) args)
    bodies;

  declaration

 with _ -> global_symbols:= old_global_symbols reraise
;;
```

C.2.4 De la syntaxe profonde vers la syntaxe superficielle

Dans la version actuelle, nous n'avons pas implanté l'algorithme de majoration canonique. Les types imprimés sont donc exactement les types calculés. Si l'on n'utilise jamais de contraintes de types, les types obtenus seront canoniques, ceci est dû à une utilisation des touffes très limitée. Le calcul de la majoration canonique ne pose pas de difficultés, et sa composition avec un imprimeur fournirait un imprimeur canonique.

Les fonctions d'impressions sont donc simplement du code d'*extension*.

```
let make_namer alphabet =
    let history = ref ([]: (num * string) list) in
    let names = ref alphabet in
    let index = ref 0 in
    (fun () -> names := alphabet; index := 0; history := [];()),
      fun v ->
       (assq v !history ?
        let rec name() =
         match !names
          with h::t -> names:=t; h
             | _    ->
                names := map (fun s -> s^string_of_num !index) alphabet;
                incr index; name() in
       let n = name() in history:= (v,n)::!history; n);;

let (reset_namer, namers) =
   it_list
      (fun (Reset,Namers) (sort, alphabet) ->
           let reset, namer = make_namer alphabet in
           ((fun() -> Reset(); reset()), (sort, namer)::Namers))
      (I,[])
      ["usual", ["a"; "b"; "c"; "d"; "e"; "f"; "g"; "h"];
       "field", ["p"; "q"; "r"; "s"; "t"];
       "flag",  ["u"; "v"; "w"; "x"; "y"; "z"]
      ];;

let namer S =
  match free  S with
     Defined (Sort s) -> assoc s namers
   | _ -> failwith "namer";;

let mark = ref 0;;
```

```
exception cycle of multi_equation;;

let Texpr_of_Type U =

  incr mark;
  let E = ref[] in

  let rec Texpr_of_Type S U =
   let U = devil U in
   let u = baby U.state  in
   if eq(u.mark, -!mark) then (u.mark <- -u.mark; Tvar(namer S U.name))
   if eq(u.mark,  !mark) then Tvar(namer S U.name)
   else
     begin match u.term
       with Var -> Tvar(namer S U.name)
          | _ ->
            u.mark <- -!mark;
            let t =
                match u.term
                 with Term (Other {print = Prefix "&pi"; _},[u]) ->
                        Tpi (Texpr_of_Type field u)
                    | Term (Other {print = Prefix s; sig = S,_;_},L) ->
                         Tapp (s, map2 Texpr_of_Type (map fst S) L)
                    | Term (Other {print = Infix C; sig = S,_;_}, L) ->
                        begin match map2 Texpr_of_Type (map fst S) L
                          with u::v::_ ->  C (u, v)
                             | _ -> failwith "Texpr_of_Type"
                        end match
                    | Term (At a, [u;v]) ->
                        Tat (unlabel a, Texpr_of_Type field u,
                             Texpr_of_Type field v)
                    | _ -> failwith "Texpr_of_Type" in
            begin if u.mark < 0
               then u.mark <- 0; t
               else let v = Tvar(namer S U.name) in E:= (v,t)::!E; v
            end if
     end match  in

  let t = Texpr_of_Type usual U in
  match !E
   with [] -> t
      | L -> Twith(t,L);;

let print_Type = print_Texpr o Texpr_of_Type;;

#printer print_Type;;
```

C.3 Simplification des contraintes

La simplification n'est guère plus compliquée. La collision de deux symboles lieurs est remplacée par une opération de mutation. On remarquera que le terme conservé dans la fusion est celui pour lequel les étiquettes sont dans l'ordre canonique. En particulier, cela réduit statistiquement le nombre de mutations. La collision d'un symbole lieur avec un autre symbole est aussi remplacée par une opération de mutation. Les contraintes de sortes assurent que ce symbole est touffu[1] et il n'est donc pas nécessaire de le vérifier dynamiquement.

[1] En effet, tous les symboles passifs construisent des types de la puissance `zero`.

La taille de cette fonction est due seulement à la duplication des cas du fait de la dissymétrie introduite par la notation de couple. Ces duplications pourraient être éliminées en rappelant la fonction `simplify` après avoir retourné les arguments.

```
#pragma infix "minimum";;

let prefix minimum = uncurry min;;

exception collision of multi_equation * multi_equation;;

let rec simplify (U,V) =

 let U = devil U in
 let V = devil V in

 if U.name = V.name then () else

  let u = baby U.state in
  let v = baby V.state in

  begin match u.term, v.term
   with Var, _ ->
          U.state <- Dead V;
          v.down <- u.down minimum v.down

      | _, Var ->
          V.state <- Dead U;
          u.down <- u.down minimum v.down

      | Term(Other C,L), Term(Other D,M) ->

          if not eq (C.nickname, D.nickname) then raise collision (U, V);

          if u.down < v.down
             then V.state <- Dead U
             else U.state <- Dead V;

          do_list simplify (combine (L,M))

    | Term(At a, L), Term(At b, M) ->

          if not eq_label (a,b) then continue;

          if u.down < v.down
             then V.state <- Dead U
             else U.state <- Dead V;

          do_list simplify (combine (L,M))


    | Term(At a, (Ua::Va::_)), Term(At b, (Ub::Vb::_)) ->

          if (baby_devil Va).down == 0
             then (simplify (Va, Ub); simplify (Va, Vb))
          if (baby_devil Vb).down == 0
             then (simplify (Vb, Ua); simplify (Va, Vb))
          else
```

```
      let k = u.down minimum v.down in

      if lt_label (a,b)
           then (V.state <- Dead U; u.down <- k)
           else (U.state <- Dead V; v.down <- k);


      let W = << [k] >> in
      simplify (Va, << b: ^Ub; ^W [k] >>);
      simplify (Vb, << a: ^Ua; ^W [k] >>)


  | Term(At a, (Ua::Va::_)), Term(Other C, []) ->

      u.down <- 0;

      simplify (Ua, V);
      simplify (Va, V)

  | Term(Other C, []), Term(At a, (Ua::Va::_)) ->

      v.down <- 0;

      simplify (Ua, U);
      simplify (Va, U)

  | Term(At a, (Ua::Va::_)), Term(Other C, M) ->

      let k = u.down minimum v.down in

      V.state <- Dead U; u.down <- k;

      let Um, Vm =
        list_it
           (fun U (Um,Vm) ->
               let Uc = << [k] >> in
               let Vc = << [k] >> in
               simplify (U, << a: ^Uc; ^Vc [k]>>);
               Uc::Um, Vc::Vm)
           M
           ([],[]) in

      simplify (Ua, << {Um} '(Other C) [k] >>);
      simplify (Va, << {Vm} '(Other C) [k] >>)

  | Term(Other C, M), Term(At a, (Ua::Va::_)) ->

      let k = u.down minimum v.down in

      U.state <- Dead V; v.down <- k;

      let Um, Vm =
        list_it
           (fun U (Um,Vm) ->
               let Uc = << [k] >> in
               let Vc = << [k] >> in
               simplify (U, << a: ^Uc; ^Vc [k]>>);
```

```
                Uc::Um, Vc::Vm)
              M
              ([],[]) in

          simplify (Ua, << {Um} '(Other C) [k] >>);
          simplify (Va, << {Vm} '(Other C) [k] >>)

    | _ -> failwith "simplify"

 end match;
 ();;
```

C.4 Récupération des vieux outils

La plupart des autres fonctions ne sont pas modifiées par l'ajout des types enregistrements.

```
let check_order n = check_order
where rec check_order U =
  let u = baby_devil U in
  if u.mark < 0 then raise cycle U
  if u.mark < !mark & u.up > n then
  begin match u.term
   with Var -> 0
      | Term(C,L) ->
          u.mark <- - !mark;
          do_list check_order L ;
          u.mark <- - u.mark
  end match;
  ();;

let recursive_types = ref false;;

let balance_at  n =
 let generic_name = ref 0 in balance
 where rec balance k U =

  let u = baby_devil U in

  if u.mark < 0 then (if !recursive_types then k else raise cycle U)
  if u.mark < !mark then

  begin do
    begin match u.term
      with Var ->
             u.up <- u.down minimum k

        | Term(C,L) ->
            begin if u.up > n then
              u.mark <- -!mark;
              let balance = balance (u.down minimum k) in
              u.up <- it_list (fun k U -> max k (balance U)) 0 L
            end if

     end match;

     if u.down > u.up then u.down <- u.up;
     u.mark <- !mark;
```

```
     if u.up > n then u.down <- decr generic_name;
     u.up
   end do

  else u.up;;

let balance_all n entries fresh N =

  let sorted = vector vect_length N + 1 of [] in

  let sort U =
    match U.state
     with Dead _ -> ()
        | Alive u -> sorted.(u.down) <- U::sorted.(u.down);() in

  do_list sort fresh;

  incr mark;
  let balance = balance_at n in
  do_vect_i

    (fun i ->

         begin if i <= n
            then
              do_list
                (let balance_i = balance i in
                 fun ({state = Alive u; _} as U) ->
                        balance_i U;
                        N.(u.up) <- U::N.(u.up)
                   | _ -> failwith "balance_all")

            else
              do_list (balance (n+1)) entries;
              let check = check_order n in
              do_list
                (fun ({state = Alive u; _} as U) ->
                        if u.up > n
                           then !recursive_types or check U
                           else N.(u.up) <- U::N.(u.up);
                        ()
                   | _ -> failwith "balance_all")
     end if)

    sorted;;

type filled = Empty | Filled of multi_equation;;

let instance k U =
  let U = devil U in
  let u = baby U.state in
  if u.down >= 0 then U else
  let table = vector (-u.down) of Empty in
  instance U

 where rec instance V =
  let V = devil V in
  let v = baby V.state in
```

```
  if v.up < u.up
     then V
     else
        match table.(-v.down-1)
         with Filled W -> W
            | Empty ->
               let W = << [k] >> in
               table.(-v.down-1) <- Filled W;
               W.state <- Alive
                 {term =
                    begin match v.term
                     with Var ->  Var
                        | Term(C, L) -> Term(C, map instance L)
                    end match;
                  down = k; up = k; mark = 0};
               W;;
```

C.5 Expressions du langage

La syntaxe abstraite superficielle a fortement grossi, par l'ajout de l'appel par filtrage et des types enregistrements, mais aussi, parce que cette maquette grâce à une syntaxe suffisamment riche doit pouvoir être utilisée avec des exemples conséquents.

```
type M
    = Mexpr of Mexpr
    | Mdecl of bool * Mpat * Mexpr
    | Mtype of Tdecl
    | Mexit

and  Mexpr
    = Mvar of string
    | Mapp of Mexpr * Mexpr
    | Mfun of (Mpat * Mexpr) list
    | Mlet of bool * Mpat * Mexpr * Mexpr
    | Mpair of Mexpr * Mexpr
    | Mconstr of string * Mexpr
    | Mif of Mexpr * Mexpr * Mexpr
    | Mconst of Mconst
    | Mrecord of Mrecord
    | Mdot of Mexpr * string
    | Mstraint of Mexpr * Texpr

and Mrecord
    = Mnull
    | Mwith of Mexpr * (string * Mexpr) list
    | Mwithout of Mexpr * string list

and Mconst
    = Mnum of num
    | Mstring of string
    | Mbool of bool
    | Munit

and  Mpat
    = Mvar_pat of string
    | Mconstr_pat of string & Mpat
```

```
    | Mwild_pat
    | Mconst_pat of Mconst_pat
    | Mpair_pat of Mpat & Mpat
    | Mrecord_pat of (string & Mpat) list
    | Mstraint_pat of Mpat & Texpr
    | Msyn_pat of Mpat & string

and Mconst_pat
    = Mnum_pat of num
    | Mstring_pat of string
    | Mbool_pat of bool
    | Munit_pat
;;


grammar for values M =

precedences
    right "->";
    right "or";
    right "&";
    right ",";
    right INFIX;
    right "@" "^";
    right "::";
    left "+" "-";
    left "*" "/";

rule entry M =
    parse "exit" -> Mexit
        | Decl d; ";;" -> d

and Decl =
    parse "type"; Tdecl d -> Mtype d
        | "let"; Rec r; Pat0 p; Let d -> Mdecl (r,p,d)
        | Expr e -> Mexpr e

and Texpr =
    parse {match parse_Texpr() with MLquote (dynamic (u:Texpr)) -> u} u -> u

and Tdecl =
    parse {match parse_Tdecl_body() with MLquote(dynamic(d:Tdecl)) -> d} d -> d

and Rec =
    parse -> false
        | "rec" -> true

and Let =
    parse Pat0 p; Let m -> Mfun [p, m]
     | "="; Expr e -> e

and Fun =
    parse Fun1 m -> [m]
        | Fun l; "|"; Fun1 m -> m::l

and Fun1 =
    parse Pat0 p; Fun1 m -> p, Mfun [m]
```

```
     | Pat0 p; "->"; Expr e -> p,e

and Function =
    parse Function1 m -> [m]
        | Function l; "|"; Function1 m -> m::l

and Function1 =
    parse Pat p; "->"; Expr e -> p,e


and entry Expr =
    parse "let"; Rec b; Pat0 p; Let d; "in"; Expr e -> Mlet (b,p,d,e)
        | "fun"; Fun m -> Mfun m
| "function"; Function m -> Mfun m
| "match"; Expr e; "with"; Function l -> Mapp(Mfun l,e)
        | "if"; Expr3 b; "then"; Expr3 u; "else"; Expr3 v -> Mif (b,u,v)
        | Expr9 e -> e

and Expr9 =
    parse Expr8 e1; ","; Expr9 e2 -> Mpair (e1,e2)
        | Expr8 e -> e

and Expr8 =
    parse "if"; Expr6 b; "then"; Expr8 u; "else"; Expr8 v -> Mif (b,u,v)
| Expr6 e -> e

and Middle6 =
    parse ("or" as l) -> l
     | ("&" as l) -> l

and Expr6 =
    parse Expr6 u; Middle6 m; Expr6 v -> Mapp(Mapp (Mvar m, u), v)
     | Expr5 e -> e

and Expr5 =
    parse ("not" as n); Expr4 e -> Mapp(Mvar n,e)
        | Expr4 e -> e

and Middle4 =
    parse ("=" as l) -> l
     | ("<=" as l) -> l
     | (">=" as l) -> l
     | (">" as l) -> l
     | ("<" as l) -> l
     | ("<>" as l) -> l

and Expr4 =
    parse Expr3 u; Middle4 m; Expr3 v -> Mapp(Mapp (Mvar m, u), v)
     | Expr3 e -> e

and Middle2 =
    parse ("+" as l) -> l
     | ("*" as l) -> l
     | ("-" as l) -> l
     | ("/" as l) -> l

and Middle1 =
```

```
    parse INFIX i -> i
     | ("@" as l) -> l
     | ("^" as l) -> l

and Expr3 =
    parse Expr3 u; Middle1 m; Expr3 v -> Mapp(Mapp (Mvar m, u), v)
        | Expr3 u; "::"; Expr3 v -> Mconstr("Cons",Mpair(u,v))
| Expr3 u; Middle2 m; Expr3 v -> Mapp(Mapp (Mvar m, u), v)
        | Expr2 e -> e

and Expr2 =
    parse Expr2 e2; Expr1 e1 -> Mapp(e2,e1)
        | Expr1 e -> e

and Expr1 =
    parse "\'"; Ident0 s; Expr0 e -> Mconstr(s,e)
   | Expr0 e -> e

and Expr0 =
    parse Cexpr c -> Mconst c
        | Ident0 v -> Mvar v
        | "{"; Record r; "}" -> Mrecord r
| Expr0 e; "."; IDENT s -> Mdot (e,s)
        | "\'"; Ident1 s -> Mconstr(s,Mconst Munit)
        | "["; "]" -> Mconstr("Nil",Mconst Munit)
        | "("; Expr e; ":"; Texpr t; ")" -> Mstraint(e,t)
| "("; Expr e; ")" -> e

and Cexpr =
    parse NUM n -> Mnum n
        | STRING s -> Mstring s
| BOOL b -> Mbool b
        | "("; ")" -> Munit

and Record =
    parse Expr9 e; "with"; fields l -> Mwith(e,l)
        | Expr9 e; "without"; labs l -> Mwithout(e,l)
        | fields l -> Mwith(Mrecord Mnull,l)

and labs =
    parse -> []
| IDENT s -> [s]
| labs l; ";"; IDENT s -> s::l

and fields =
    parse -> []
     | field p -> [p]
| fields l; ";"; field p -> p::l

and field =
    parse IDENT s; "="; Expr9 e -> (s,e)


and entry Pat =
    parse Pat2 p; Literal"as"; Ident0 s -> Msyn_pat(p,s)
        | Pat2 p -> p
```

```
and Pat2 =
    parse Pat1 p; ","; Pat2 q -> Mpair_pat(p,q)
        | Pat1 p -> p

and Pat1 =
    parse Pat1 p; "::"; Pat1 q -> Mconstr_pat("Cons",Mpair_pat(p,q))
        | "\'"; Ident0 s; Pat0 p -> Mconstr_pat(s,p)
        | Ident0 s; Pat0 p -> Mconstr_pat(s,p)
        | Pat0 p -> p

and Pat0 =
    parse Cpat c -> Mconst_pat c
        | "_" -> Mwild_pat
        | Ident0 s -> Mvar_pat s
        | "["; "]" -> Mconstr_pat("Nil",Mconst_pat Munit_pat)
        | "\'"; Ident1 s -> Mconstr_pat(s,Mconst_pat Munit_pat)
        | "{"; Rpat r; "}" -> Mrecord_pat r
        | "("; Pat p; ":"; Texpr t; ")" -> Mstraint_pat(p,t)
        | "("; Pat p; ")" -> p

and Cpat =
    parse NUM n -> Mnum_pat n
        | STRING s -> Mstring_pat s
| BOOL b -> Mbool_pat b
        | "("; ")" -> Munit_pat

and Rpat =
    parse Rpat1 r -> [r]
        | Rpat R; ";"; Rpat1 r -> r::R

and Rpat1 =
    parse Ident0 s; "="; Pat1 p -> s,p


and Ident0 =
    parse IDENT s -> s
        | "prefix"; Infix0 s -> s

and Ident1 =
    parse Ident0 s -> s
        | BOOL b  -> string_of_bool b

and Infix0 =
    parse INFIX i -> i
        | ("+" as n) -> n
| ("-" as n) -> n
| ("*" as n) -> n
| ("/" as n) -> n
| ("=" as n) -> n
| ("<=" as n) -> n
| (">=" as n) -> n
| ("<" as n) -> n
| (">" as n) -> n
| ("<>" as n) -> n
| ("@" as n) -> n
| ("^" as n) -> n
| ("not" as n) -> n
```

```
| ("or" as n) -> n
| ("&" as n) -> n

;;

let parse = (M "M").Parse;;

#set default grammar Type;;

type Lexpr
    = Lvar of pat
    | Lapp of Lexpr * Lexpr
    | Lfun of (Lpat * Lexpr) list
    | Llet of Lpat * Lexpr * Lexpr
    | Lpair of Lexpr * Lexpr
    | Lrecord of Lrecord
    | Lconst of Mconst
    | Lstraint of Lexpr * Texpr

and Lrecord
    = Lnull
    | Lwith of Lexpr * (label * Lexpr) list
    | Lwithout of Lexpr * label list

and  Lpat
    = Lvar_pat of pat
    | Lconstr_pat of pat & Lpat
    | Lwild_pat
    | Lconst_pat of Mconst_pat
    | Lpair_pat of Lpat & Lpat
    | Lrecord_pat of (label & Lpat) list
    | Lstraint_pat of Lpat & Texpr
    | Lsyn_pat of Lpat & pat

and pat = {surname : string; mutable annotation : Type};;

let global_values = vector 50 of ([]: (string * multi_equation) list);;

let get_value, store_value = make_table 0 global_values
and get_constr, store_constr = make_table 1 global_values;;

let empty = << [0] >>;;

let pat_of_var x = {surname = x; annotation = empty} ;;

let sort_fields_gt = sort (function (a,_), (b,_) -> gt_label (a,b));;

let Linp values p =
 let values = ref values in
 let pat_of_var s =
     let p = pat_of_var s in
     values := (s,p)::!values; p in

 (let p = Linp p in !values, p)

 where rec Linp p =
 match p
  with Mvar_pat s -> Lvar_pat (pat_of_var s)
     | Mconstr_pat(s,p) ->
```

```
        Lconstr_pat
          ({surname = s; annotation = get_constr s}, Linp p ?
           raise unbound ("constructor",s))
     | Mwild_pat -> Lwild_pat
     | Mconst_pat c -> Lconst_pat c
     | Mpair_pat(p,p') -> Lpair_pat (Linp p, Linp p')
     | Mrecord_pat L ->
        Lrecord_pat
          (sort_fields_gt (map (function s,p -> label s, Linp p) (rev L)))
     | Mstraint_pat (p,u) -> Lstraint_pat (Linp p, u)
     | Msyn_pat (p,s) -> Lsyn_pat(Linp p, pat_of_var s)
;;

let rec Lin values = Lin_values
 where rec Lin_values e =
 match e
  with Mvar s ->
        Lvar (assoc s values ? {surname = s; annotation = get_value s}) ?
        raise unbound ("variable",s)

     | Mapp (e,e') -> Lapp (Lin_values e, Lin_values e')

     | Mfun L ->
        let lin (p,e) =
          let values, p = Linp values p in
          p, Lin values e in
        Lfun (map lin L)

     | Mlet (b,p,e,e') ->
        let values, p = Linp values p in
        let e =
          if b
             then Lapp (Lin_values (Mvar "&fix"), Lfun [p, Lin values e])
             else Lin_values e in
let e' = Lin values e' in
        Llet (p, e, e')

     | Mif (b,e,e') ->
          Lin_values (Mapp (Mapp (Mapp (Mvar "&if", b), e), e'))

     | Mconstr(C,e) ->
        Lapp (Lvar {surname = C; annotation = get_constr C}, Lin_values e) ?
        raise unbound ("constructor",C)

     | Mconst c -> Lconst c

     | Mpair (e,e') -> Lpair (Lin_values e, Lin_values e')

     | Mdot (e,s) ->
        Lin_values
          (Mapp (Mfun [Mrecord_pat [s, Mvar_pat"[dot]"], Mvar "[dot]"], e))

     | Mrecord r ->
        Lrecord
          begin match r with
            Mnull -> Lnull
          | Mwith (e,L) ->
```

```
            let L = map (fun (s,e) -> label s, Lin_values e) (rev L) in
            Lwith (Lin_values e, sort_fields_gt L)
          | Mwithout (e,L) ->
            Lwithout (Lin_values e, sort gt_label (map label (rev L)))
          end match

    | Mstraint (e,t) -> Lstraint (Lin_values e, t)
;;
```

C.6 Cœur du système

C.6.1 Une petite optimisation

Il est plus naturel pour le typage de l'application de filtrer le type de l'argument par un type flèche plutôt que de construire un type flèche que l'on s'empresse d'unifier avec le type de l'argument. Nous introduisons donc une fonction `match_arrow`. De même nous utiliserons une fonction `match_pi` pour extraire la touffe d'un type enregistrement.

```
let var0 = << [0] >>;;

let arrow0 = << ^var0 -> ^var0 [0] >>;;

let match_arrow U =
  let U = devil U in
  let u = baby U.state in
  match u.term
   with Term (Other C, V::W::_) ->
          if eq (C.nickname, arrow.nickname) then V,W else continue
      | Var ->
          let V = << [u.down] >> in
          let W = << [u.down] >> in
          U.state <- Dead << ^V -> ^W [u.down] >>;
          V, W
      | _ -> raise collision (U, arrow0)
;;

let pi0 = << ^var0 'Pi [0] >>;;

let match_pi U =
  let U = devil U in
  let u = baby U.state in
  match u.term
   with Term (Other C, V::_) ->
          if eq (C.nickname, pi.nickname) then V else continue
      | Var ->
          let V = << [u.down] >> in
          U.state <- Dead << ^V 'Pi [u.down] >>; V
      | _ -> raise collision (U, pi0)
;;
```

C.6.2 Primitives sur les objets enregistrements

Nous regroupons les types des primitives sur les enregistrements dans un objet mutable unique ; il sera ainsi facile de les modifier, par exemple pour essayer plusieurs variantes du typage des enregistrements. Nous donnons par défaut les primitives du système ΠML.

```
let pre = create true  ("pre", Prefix "pre", [], flag, zero)
and abs = create true  ("abs", Prefix "abs", [], flag, zero);;
```

```
let absent = << '(Other abs) [0]>>
and present = << '(Other pre) [0]>>;;

type type_record =
    {!pat : num -> (label * Type) list -> Type;
     !null : num -> Type;
     !new: num -> Type -> (label * Type) -> Type;
     !forget: num -> Type -> label -> Type};;

let type_record =

  let pat n L =
    let R =
        it_list
          (fun R (a, U) -> << a: (^present . ^U [n]) ; ^R [n] >>)
          << [n] >>
          L  in
    << ^R 'Pi [n] >> in

  let null n = << ^absent . ([n]) [n] >> in

  let new n V (a, U) =
    let W = << [n] >> in
    simplify (V, << a: ([n]); ^W [n] >>);
    << a: (([n]) . ^U [n]) ; ^W [n] >> in

  let forget n V a =
    let W = << [n] >> in
    simplify (V, << a: ([n]); ^W [n] >>);
    << a: (^absent . ([n]) [n]); ^W [n] >> in

  {pat = pat; null = null; new = new; forget = forget};;
```

C.6.3 Typage des motifs de filtrage

Le typage des motifs de filtrage pourrait être obtenu simplement par codage dans un langage sans appel par filtrage. Nous donnons un typage direct, car c'est aussi simple que d'effectuer la traduction. La précaution à prendre est de vérifier que les motifs sont linéaires.

```
let unit   = create true  ("unit",   Prefix "unit",   [], usual, zero)
and num    = create true  ("num",    Prefix "num",    [], usual, zero)
and string = create true  ("string", Prefix "string", [], usual, zero)
and bool   = create true  ("bool",   Prefix "bool",   [], usual, zero);;

let Num    = << '(Other num) [0]>>
and String = << '(Other string)  [0]>>
and Unit   = << '(Other unit) [0]>>
and Bool   = << '(Other bool) [0]>>;;

let map_snd f = map (function u,v -> u, f v);;

let type_pat n =
 let instance_n = instance n in
 type_pat
where rec type_pat p =
    match p
     with Lvar_pat p ->
           p.annotation <- << [n] >>
```

```
        | Lconstr_pat (C, p) ->
           let u,v = match_arrow (instance_n C.annotation) in
           simplify (u, type_pat p);
           v

        | Lwild_pat -> << [n] >>

        | Lconst_pat c ->
            (match c
              with Mnum_pat n    -> Num
                 | Mstring_pat s -> String
                 | Mbool_pat b   -> Bool
                 | Munit_pat     -> Unit)

        | Lpair_pat (p,p') ->
            << ^(type_pat p) * ^(type_pat p')  [n] >>

        | Lrecord_pat L ->
            type_record.pat n  (map_snd type_pat L)

        | Lstraint_pat (p,t) ->
            let u = type_pat p in
            let v = Texpr_to_Type n t in
            simplify (u,v);
            u

        | Lsyn_pat (p,v) ->
            v.annotation <- type_pat p
;;
```

C.6.4 Extension du typage des expressions

```
let rec type_expr n =
  let type_pat_n = type_pat n in
  let instance_n = instance n in
  type_expr_n
 where rec type_expr_n e =
    match e
     with Lvar x ->
            instance_n x.annotation

        | Lapp (e,e') ->
            let u,v = match_arrow (type_expr_n e) in
            simplify (u, type_expr_n e');
            v

        | Lfun ((p,e)::M) ->
            let u = type_pat_n p in
            let v = type_expr_n e in
            do_list
                (function p, e ->
                    simplify (u, type_pat_n p);
                    simplify (v, type_expr_n e))
                M;
            << ^u -> ^v [n] >>
```

```
        | Llet (p, e, e') ->
            let N = !active_system in
            let N1 = vector vect_length N + 1 of [] in
            do_vect_i (fun i u -> N1.(i) <- u) N;
            active_system := N1;
            let u = type_expr (n+1) e in
            let v = type_pat (n+1) p in
            simplify (u,v);
            active_system := N;
            do_vect_i (fun i _ -> N.(i) <- N1.(i)) N;
            balance_all n [u] N1.(n+1) N;
            type_expr_n e'

        | Lconst  c ->
            (match c
              with Mnum n    -> Num
                 | Mbool b   -> Bool
                 | Mstring b -> String
                 | Munit     -> Unit)


        | Lpair(e,e') ->
            let u = type_expr_n e in
            let v = type_expr_n e' in
            << ^u * ^v [n] >>

        | Lrecord Lnull ->
            << ^(type_record.null n) 'Pi [n] >>

        | Lrecord (Lwith (e,L)) ->
            let u = type_expr_n e in
            let v =
              it_list
                 (type_record.new n)
                 (match_pi u)
                 (map_snd type_expr_n L) in
            << ^v 'Pi [n] >>

        | Lrecord (Lwithout (e,L)) ->
            let u = type_expr_n e in
            let v = it_list (type_record.forget n) (match_pi u) L in
            << ^v 'Pi [n] >>

        | Lstraint (e,t) ->
            let u = type_expr_n e in
            let v = Texpr_to_Type n t in
            simplify (u,v);
            u

        | Lfun [] -> failwith "type_expr";;
```

C.7 Extension de l'environnement utilisateur

```
exception error;;

let error = "Typechecking error:";;
```

```
#set default grammar Pretty:Exec;;

let print_power p =
  << [ (print Zero ->  "*"
            | Effective [] ->  "[]"
            | Effective (a::A) ->
                print_label a;
                ( * (print a ->  "."; print_label a)) A) p] >>;;

let handle f =
  try f()
  with rebound (s,d) ->
         << [<v> error; " variable '"; Ident s; " is bound twice in";
            \-; print_Texpr_decl d] >>;
         raise error
     | unbound (m,s) ->
         << [<hv> error; " unbound "; Ident m; \-; Ident s] >>;
         raise error
     | collision (u,v) ->
         << [<hv> [<h> error; " collision between"]; \-;
                  [<hv> print_Type u; \-; "and"; \-; print_Type v]] >>;
         raise error
     | cycle u ->
         << [<hov> error; " cycle "; \-; print_Type u] >>;
         raise error
     | report (err, u) ->
         << [<hov> error; " in signature checking "; \-;
              {function
                     Esort (Sort u, Sort v) ->
                       << " incompatible sorts"; \-; Ident u; " and";
                          \-; Ident v >>
                   | Epower (u,v) ->
                       << " incompatible sorts"; \-; print_power u; " and";
                          \-; print_power v >>
                   | Eadd (a,A) ->
                      << " label "; print_label a; " cannot be added";
                         \-; "to power"; \-; print_power A >>
                   | Eother s ->
                      << Ident s>>} err; " in";
              \-; print_Texpr u] >>;
         raise error;;

let eval_expr s e=
  let eval() =
     let e = Lin [] e in
     let p = pat_of_var s in
     let e = Llet (Lvar_pat p, e, Lconst Munit) in
     active_system:= [|[]|];
     type_expr 0 e;
     store_value s p.annotation;
     << [<hov 2> [s; " :"]; \-; print_Type {p.annotation}] >> in

  handle eval;
  active_system:= [|[]|];;

let eval_type d =
  let eval() =
    active_system:= [|[];[]|];
```

```
    let L = type_type 1 d in
    balance_all 0 (map snd L) [] [|[]|];
    active_system:=[|[]|];
    do_list (uncurry store_constr) L;
      << [<v> "New constructors declared:";
         ( *(print u -> \-; Ident {fst u}; " : "; [print_Type {snd u}])) L] >>
  in
  handle eval;;

exception exit;;

let eval  d =
    reset_namer();
    begin match d
      with Mdecl (b,(Mvar_pat s as p),e) ->
            eval_expr s (if b then Mlet(b,p,e,Mvar s) else e)

         | Mdecl (b,(Mstraint_pat(Mvar_pat s,t) as p),e) ->
            eval_expr s (if b then Mlet(b,p,e,Mvar s) else Mstraint (e,t))

         | Mexpr e -> eval_expr "it" e

         | Mtype d  -> eval_type d

         | Mexit -> raise exit

         | Mdecl _ -> failwith "Construction not supported"
    end match;
    print_newline();;

let step() =
   eval (match eval_syntax(parse()) with dynamic (x:M) -> x);;

let rec typ() =
    let rec loop() =
      print_newline();
      begin try step() with error -> failwith "error" end try;
      typ() in
    try loop() with exit -> ();;

let rec top() =
    let rec loop() =
      print_newline();
      begin try parsing_handler print_syntax_error step()
        with error -> print_newline()
      end try;
      loop() in

    let prompt = prompt() in
    set_prompt "> ";
    begin try loop()
      with exit -> set_prompt prompt
         | _ -> set_prompt prompt reraise
    end try
;;

#set default grammar Type;;
```

Il sera nécessaire d'initialiser l'environnement global par quelques primitives avant de commencer à jouer.

```
recursive_types:= true;;

step();;

fun f -> (fun x -> f (x x)) (fun x -> f (x x));;

store_value "&fix" (get_value "it");;

recursive_types:= false;;

step();;

let rec x = (x: bool -> 'a -> 'a -> 'a) in x;;

store_value "&if" (get_value "it");;
```

Afin de travailler dans plusieurs environnement simultanément, il est utile d'écrire une fonction de sauvegarde de l'environnement.

```
let copy_vect U V = do_vect_i (fun i u -> V.(i) <- u) U;;

let save() =
  let values = vector vect_length global_values of [] in
    copy_vect global_values values;
  let symbols = !global_symbols in
  let pat = type_record.pat in
  let null = type_record.null in
  let new = type_record.new in
  let forget = type_record.forget in

  fun() ->
    copy_vect values global_values;
    global_symbols := symbols;
    type_record.pat <- pat;
    type_record.null <- null;
    type_record.new <- new;
    type_record.forget <- forget;
    ();;

let restore = save();;
```

L'utilisateur pourra se définir un environnement `prelude`. Le nôtre sera le suivant :

```
recursive_types:= true;;
typ();;
let fix_point f = (fun x -> f (x x)) (fun x -> f (x x));;
let rec iff = (iff: bool -> 'a -> 'a -> 'a) in iff;;
exit
recursive_types:= false;;
(*
store_value "fix_point"
  (Type_to_typ 1 <:Type< ('a -> 'a) -> 'a >>)
;;
*)
store_value "if" (get_value "it");;
```

```
typ();;

let rec magic = magic;;
let rec (equal:'a -> 'a -> bool) = equal;;
let (prefix +: num -> num -> num) = magic;;
let (sup: num -> num -> bool) = magic;;
let (prefix &: bool -> bool -> bool) = magic;;
let (Erreur: string -> 'a) = magic;;
let (eq: 'a -> 'a -> bool) = magic;;
let prefix - = prefix +;;
let succ x = 1+x;;
let pred x = 1-x;;
let (I: 'a -> 'a) = magic;;
let (failwith: string -> 'a) = magic;;
let prefix or x y = if x then true else y;;

let fst (x,y) = x;;
let snd (x,y) = y;;

type list('a) = Nil of unit | Cons of 'a * list('a);;
let cons = fun x y -> 'Cons (x,y);;

let prefix o f g x = f (g x);;

let it_list f =
 let rec iter = fun a -> function
   Nil() -> a
 | Cons(h,t) -> iter (f a h) t
 in iter
;;

let list_it f l b =
 let rec iter = function
   Nil() -> b
 | Cons(h,t as l)  -> f h (iter t)
 in iter l
;;

let append = list_it cons;;

let mem u =
 let rec memrec = function
   Nil()  -> false
 | Cons(h,t) -> (equal u h) or (memrec t)
 in memrec
;;


let rec length = function
  Nil()  -> 0
| Cons(_,t) -> succ (length t)
;;

let rec map foo = function
  Nil()   -> 'Nil()
| Cons(h,t) -> 'Cons (foo h, map foo t)
;;
```

```
let hd (Cons(h,t)) = h;;
let tl (Cons(h,t)) = t;;
();;
exit

let prelude = save();;
```

Il est très facile de définir l'environnement de typage pour le système ΠML′.

```
restore();;

global_symbols :=
   filter_neg (fun (s,_) -> mem s ["pre"; "dot"; "field"]) !global_symbols;
   ();;

let Pre = create true  ("Pre", Prefix "Pre", [usual, zero], field, zero)
and Abs = create true  ("Abs", Prefix "Abs", [], field, zero);;

let Absent  = << '(Other Abs) [0]>>
and Present = Other Pre;;

let pat n L =
    let R =
        it_list
          (fun R (a, U) -> << a: (^U 'Present [n]) ; ^R [n] >>)
          << [n] >>
          L  in
    << ^R 'Pi [n] >> in

  let null n = << ^Absent >> in

  let new n V (a, U) =
    let W = << [n] >> in
    simplify (V, << a: ([n]); ^W [n] >>);
    << a: (^U 'Present [n]) ; ^W [n] >> in

  let forget n V a =
    let W = << [n] >> in
    simplify (V, << a: ([n]); ^W [n] >>);
    << a: ^Absent; ^W [n] >> in

    type_record.pat <- pat;
    type_record.null <- null;
    type_record.new <- new;
    type_record.forget <- forget;;

let weak = save();;

restore();;
```

Bibliographie

[1] Barendregt (Hans P.). – *The Lambda-Calulus. Its Syntax and Semantics.* – North-Holland, 1984, *Studies in Logic and The Foundations of Mathematics*, volume 103.

[2] Blott (Stephen) et Wadler (Philip). – How to make *ad-hoc* polymorphism less *ad-hoc. In : Sixteenth Annual Symposium on Principles Of Programming Languages.* – 1989.

[3] Bürckert (Hans-Jürgen), Herold (Alexander) et Schmidt-Schauß (Manfred). – *On equational theories, unification and decidability.* – Rapport technique n° SR-86-20, West Germany, Universität Kairserslautern, 1986.

[4] Bürckert (Hans-Jürgen), Herold (Alexander) et Schmidt-Schauß (Manfred). – On equational theories, unification and decidability. *In : Second Conference on Rewriting Techniques and Applications.* – Springer-Verlag, 1987.

[5] Bürckert (Hans-Jürgen), Herold (Alexander) et Schmidt-Schauß (Manfred). – On equational theories, unification and decidability. *Journal Of Symbolic Computation*, vol. 8, n° 1 & 2, 1989, pp. 3–50.

[6] Canning (Peter), Cook (William), Hill (Walter), Olthoff (Walter) et Mitchell (John C.). – F-bounded polymorphism for object oriented programming. *In : The Fourth International Conference on Functional Programming Languages and Computer Architecture.* – 1989.

[7] Cardelli (Luca). – A semantics of multiple inheritance. *In : Semantics of Data Types. Lecture Notes in Computer Science*, volume 173, pp. 51–68. – Springer Verlag, 1983.

[8] Cardelli (Luca). – Amber. *In : Combinators and Functional Programming Languages. Lecture Notes in Computer Science*, volume 242, pp. 21–47. – Spinger Verlag, 1986. Proceedings of the 13th Summer School of the LITP.

[9] Cardelli (Luca). – A semantics of multiple inheritance. *Information and Control*, 1988.

[10] Cardelli (Luca). – Structural subtyping and the notion of power type. *In : Fifteenth Annual Symposium on Principles Of Programming Languages.* – 1988.

[11] Cardelli (Luca) et Mitchell (John C.). – Operations on records. *In : Fifth International Conference on Mathematical Foundations of Programming Semantics.* – 1989.

[12] Cardelli (Luca) et Wegner (Peter). – On understanding types, data abstraction, and polymorphism. *Computing Surveys*, vol. 17 (4), 1985, pp. 471–522.

[13] Clément (Dominique), Despeyroux (Joëlle), Despeyroux (Thierry) et Kahn (Gilles). – *A Simple Applicative Language : Mini-ML.* – Rapport technique n° 529, France, Institut National de Recherche en Informatique et Automatisme, 1986.

[14] Coppo (Mario). – An extended polymorphic type system for applicative languages. *In : MFCS '80. Lecture Notes in Computer Science*, volume 88, pp. 194–204. – Springer Verlag, 1980.

[15] Courcelle (Bruno). – Fundamental properties of infinite trees. *Theoritical Computer Science*, vol. 25, 1983, pp. 95–169.

[16] Cousineau (Guy) et Huet (Gérard). – *The CAML Primer.* – Institut National de Recherche en Informatique et Automatisme, France, 1989.

[17] Curtis (Pavel). – *Constrained Quantification in Polymorphic Type Analysis* – Thèse de PhD, Cornell, 1987.

[18] Damas (Luis) et Milner (Robin). – Principal type-schemes for functional programs. *In : Nineteenth Annual Symposium on Principles Of Programming Langages.* – 1982.

[19] Dershowitz (Nachum) et Jouannaud (Jean-Pierre). – *Rewrite Systems.* – Rapport technique n° 478, Orsay (France), Université de Paris-Sud, Laboratoire de recherche en Informatique, April 1989.

[20] Doggaz (Narjes) et Kirchner (Claude). – Completion for unification. – 1989. CRIN & INRIA Loraine, Nancy (France).

[21] Eder (E.). – Properties of substitutions and unifications. *Journal Of Symbolic Computation*, vol. 1, n° 1, 1985, pp. 31–46.

[22] Fages (François) et Huet (Gérard). – Complete sets of unifiers and matchers in equational theories. *Theoretical Computer Science*, vol. 43, 1986.

[23] Fuh (You-Chin) et Mishra (Prateek). – Type inference with subtypes. *In : ESOP '88. Lecture Notes in Computer Science*, volume 300, pp. 94–114. – Springer Verlag, 1988.

[24] Fuh (You-Chin) et Mishra (Prateek). – Polymorphic subtype inference : Closing the theory-practice gap. *In : TAPSOFT'89.* – 1989.

[25] Girard (Jean-Yves). – *Interprétation Fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* – Thèse de troisième cycle, Université de Paris 7, 1972.

[26] Hindley (J. Roger) et Seldin (Jonathan P.). – *Introduction to Combinators and λ-calculus.* – Cambridge Universuty Press, 1986, *Volume* 1 *of London Mathematical Society Student texts.*

[27] Huet (Gérard). – *Résolution d'équations dans les langages d'ordre* $1, 2, \ldots, \omega$. – Thèse de doctorat d'état, Université Paris 7, 1976.

[28] Huet (Gérard). – *Lambda-Calcul, Notes de Cours de DEA.* – Institut National de Recherche en Informatique et Automatisme, 1989.

[29] Huet (Gérard) et Oppen (Derek). – Equations and rewrites rules : A survey. *Formal Language Theory*, 1980.

[30] Jategaonkar (Lalita A.) et Mitchell (John C.). – ML with extended pattern matching and subtypes. *In : Proceedings of the 1988 Conference on LISP and Functional Programming.* – 1988.

[31] Kanellakis (Paris) et Mitchell (John C.). – Polymorphic unification and ML typing. *In : Proceedings of the Sixteenth Annual Symposium on Principles Of Programming Languages.* – 1989.

[32] Kfoury (A.J.), Tiuryn (J.) et Urzyczyn (P.). – On the computational power of universally polymorphic recursion. *In : Third Symposium on Logic In Computer Science.* – 1988.

[33] Kfoury (A.J.), Tiuryn (J.) et Urzyczyn (P.). – A proper extension of ML with an effective type-assigment. *In : Fifteenth Annual Symposium on Principles Of Programming Languages.* – 1988.

[34] Kfoury (A.J.), Tiuryn (J.) et Urzyczyn (P.). – An analysis of ML typability. – 1989. Preprint.

[35] Kfoury (A.J.), Tiuryn (J.) et Urzyczyn (P.). – Computational consequences and partial solutions of a generalized unification problem. *In : Fourth Symposium on Logic In Computer Science.* – 1989.

[36] Kirchner (Claude). – *Méthodes et outils de conception systématique d'algorithmes d'unification dans les théories équationnelles.* – Thèse de doctorat d'état en informatique, Université de Nancy 1, 1985.

[37] Kirchner (Claude). – Computing unification algorithms. *In : Proceeding of the first symposium on Logic In Computer Science*, pp. 206–216. – Boston (USA), 1986.

[38] Kirchner (Claude). – *Order-Sorted Equationnal Unification.* – Rapport technique n° 954, France, Institut National de Recherche en Informatique et Automatisme, 1988.

[39] Kirchner (Claude) et Klay (François). – Syntactic theories and unification. – 1989. CRIN & INRIA Loraine, Nancy (France).

[40] Knight (Kevin). – Unification : A multidisciplinary survey. *ACM Computing Surveys*, vol. 21, n° 1, 1989, pp. 93–124.

[41] Mairson (Harry G.). – Deciding ML typability is complete for deterministic exponentional time. *In : Seventeenth Annual Symposium on Principles Of Programming Languages.* – 1990.

[42] Martelli (Alberto) et Montanari (Ugo). – An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, vol. 4, n° 2, 1982, pp. 258–282.

[43] Mitchell (John C.). – Coercion and type inference. *In : Eleventh Annual Symposium on Principles Of Programming Languages.* – 1984.

[44] Mitchell (John C.). – Polymorphic type inference. *Information and Computation*, 1988.

[45] Mitchell (John C.). – Towards a typed foundation for method specialisation and inheritance. *In : seventeenth Annual Symposium on Principles Of Programming Languages.* – 1990. To appear.

[46] Mycroft (Alan). – Polymorphic type schemes and recursive definitions. *In : Sixth International Symposium on Programming. Lecture Notes in Computer Science*, volume 167. – Springer Verlag, 1984.

[47] Nipkow (Tobias). – *Proof Transformations for Equational Theories.* – Rapport technique n° 181, University of Cambridge, Computer laboratory, 1989.

[48] Ohori (Atsushi) et Buneman (Peter). – Type inference in a database langage. *In : ACM Conference on LISP and Functional Programming*, pp. 174–183. – 1988.

[49] Ohori (Atsushi) et Buneman (Peter). – Static type inference for parametric classes. *In : COPSLA'89.* – 1989.

[50] Ohori (Atsushi), Buneman (Peter) et Breazu-Tannen (Val). – *Database Programminf in Machiavelli - A Polymorphic langage with static type inference.* – Rapport technique n° MS-CIS-88-103, University of Pennsylvania, Computer and Information Science department, 1988.

[51] Paterson (Michael S.) et Wegman (Mark N.). – Linear unification. *Journal of Computer and System Sciences*, vol. 16, 1978, pp. 158–167.

[52] Rémy (Didier). – Records and variants as a natural extension of ML. *In : Sixteenth Annual Symposium on Principles Of Programming Languages.* – 1989.

[53] Schmidt-Schauß (Manfred). – Unification in many-sorted equational theories. *In : 8th International Conference on Automated Deduction. Lecture Notes in Computer Science*, volume 230, pp. 538–552. – Springer Verlag, 1986.

[54] Schmidt-Schauß (Manfred). – *Computational Aspects of Orded-Sorted Logic with Term Declarations.* – West Germany, Thèse de PhD, University of Edinburgh, 1987.

[55] Schmidt-Schauß (Manfred) et Siekmann (Jörg H.). – *Unification Algebras : An Axiomatic Approach to Unification, Equation Solving and Constraint Solving.* – Rapport technique n° SR-88-23, West Germany, Universität Kairserslautern, 1988.

[56] Siekmann (Jörg H.). – Unification theory. *Journal Of Symbolic Computation*, vol. 7, n° 7, 1989, pp. 207–276.

[57] Smolka (Gert), Nutt (Werner), Goguen (Joseph A.) et Meseguer (Jose). – Order sorted equational computation. *In : Colloquium on Resolution of Equations in Algebraic Structures.* – Austin (Texas), May 1987.

[58] Tofte (Mads). – *Operational Seamntics and Polymorphic Type Inference.* – Thèse de PhD, University of Edinburgh, 1987.

[59] Wand (Mitchell). – Complete type inference for simple objects. *In : Second Symposium on Logic In Computer Science.* – 1987.

[60] Wand (Mitchell). – Corrigendum : Complete type inference for simple objects. *In : Third Symposium on Logic In Computer Science.* – 1988.

[61] Wand (Mitchell). – Type inference for record concatenation and multiple inheritance. *In : Fourth Annual Symposium on Logic In Computer Science*, pp. 92–97. – 1988.

[62] Weis (Pierre). – *The CAML Reference Manual.* – Institut National de Recherche en Informatique et Automatisme, France, 1989.

Notations

u, v, w,	suites d'éléments.
x, y, z,	éléments d'une suite.
ι, κ,	sortes.
π, ϖ,	tuples de sortes.
f, g, h,	symboles, symboles touffus.
f', g', h',	symboles quelconques des algèbres touffues.
a, b, c,	symboles constants ou étiquettes 2.
P, Q, R,	puissances effectives.
$\bar{P}$, $\bar{Q}$, $\bar{R}$,	puissances quelconques.
α, β, γ, δ,	variables de termes (de types).
V, W,	ensembles de variables.
$\mathcal{X}$, $\mathcal{Y}$, $\mathcal{Z}$,	ensembles des variables liées des types quantifiés.
σ, τ, ρ, π,	termes (de types).
σ_f, τ_f, ρ_f,	termes frontières.
σ_g, τ_g, ρ_g,	types génériques.
μ, ν, ξ,	substitutions.
μ_f, ν_f, ξ_f,	frontières.
μ^g, ν^g, ξ^g,	substitutions génériques.
θ, η,	renommages.
θ^g, η^g,	renommages de $\mathcal{V}^g$ sur $\mathcal{V}$ ou de $\mathcal{V}$ sur $\mathcal{V}^g$.
e, e',	multi-équations.
$\mathcal{M}$, $\mathcal{N}$,	systèmes de multi-équations.
U, U',	unificandes.
Σ, Θ,	ensembles complets d'unificateurs.
φ, ψ, χ,	variables de la sorte field ou expansions.
Γ, Δ,	contextes de typage.

Utilisation des méta-variables.

Index

Résumé

Le langage ML est un langage fonctionnel typé polymorphe avec synthèse de types. Nous proposons une méthode plus qu'une solution, à la fois simple et efficace, permettant la synthèse des types dans une extension de ML avec des objets enregistrements avec héritage. Simplicité, car les règles de typage de ML ne sont pas modifiées, l'héritage étant obtenu simplement par le polymorphisme dans les types enregistrements. Efficacité, car nous montrons comment extraire naturellement à partir des règles de typage un algorithme avec partage et permettant un calcul incrémental de la généralisation.

Les objets enregistrements sont des fonctions partielles de domaines finis d'un ensemble dénombrable d'étiquettes vers l'ensemble des valeurs. Nous leurs associons des types enregistrements qui sont des fonctions partielles de l'ensemble des étiquettes vers l'ensemble des types, que nous rendons totales en définissant des types par défaut pour les étiquettes absentes.

Les types enregistrements sont représentables de manière finie dans une algèbre de termes munie d'une famille d'équations très sévèrement controlées par des sortes, que nous appelons la théorie touffue à brins étiquetés. Dans une touffe, l'extraction des brins est obtenue par substitution pour les variables et par propagation vers la racine par des équations de distributivité, et la permutation des brins est assurée par des équations de commutativité gauche.

Nous montrons que la théorie touffue à brins étiquetés est une théorie syntaxique, et nous en déduisons facilement que l'unification y est décidable et unitaire. Puis nous montrons que l'algèbre des types de ML peut être étendue avec une théorie décidable, unitaire et régulière.

Le langage obtenu est paramétré par le jeu de primitives avec leur types qui réalisent les opérations élémentaires sur les enregistrements. Cette méthode devrait s'étendre à des types récursifs et s'appliquer à d'autres langages que ML, notamment à des langages avec de l'inclusion de types.

Mots Clefs

Algèbres Touffues, Classes, Héritage, Inclusion, Langages Fonctionnels,

ML, Objets Enregistrements, Polymorphisme, Programmation Orientée par les Objets, Synthèse de Types, Théories Equationnelles, Théories Syntaxiques, Typage, Unification.