N° d'ordre:

THÈSE

présentée à

L'UNIVERSITÉ PARIS VII

pour obtenir le titre de

**DOCTEUR DE L'UNIVERSITÉ
PARIS VII**

Spécialité :

INFORMATIQUE

par

François POTTIER

Sujet de la thèse :

SYNTHÈSE DE TYPES EN PRÉSENCE DE SOUS-TYPAGE : DE LA THÉORIE À LA PRATIQUE

Soutenue le 3 juillet 1998 devant le jury composé de :

MM.	**Jean-Pierre**	**JOUANNAUD**	Président
	Claude	**KIRCHNER**	Rapporteurs
	Jens	**PALSBERG**	
	Dominique	**BOLIGNANO**	Examinateurs
	Guy	**COUSINEAU**	
	Michel	**MAUNY**	
	Didier	**RÉMY**	

Table des matières

Remerciements

Je tiens à remercier tout particulièrement M. Didier Rémy pour avoir accepté de diriger ce travail, et pour avoir pris cette charge très à cœur. Toujours disponible, Didier a su, grâce à une intuition et une confiance étonnantes, inciter un « thésard » parfois dérouté à surmonter chaque difficulté.

Moi :	Ça ne marche pas. (*suit une explication technique nébuleuse*)
Lui :	Pourtant il y a là une intuition claire...
	(*griffonne un croquis mystérieux*) Ça doit marcher.
Moi :	... !

A son contact, je pense avoir appris à envisager les choses non plus d'un point de vue purement technique, mais avec recul, en reformulant d'abord chaque problème pour en tirer les questions essentielles, avant d'y répondre. En bref, c'est grâce à son expérience que j'ai pu acquérir – je l'espère – la démarche d'un chercheur.

Je remercie également l'ensemble des membres du jury – en particulier MM. Claude Kirchner et Jens Palsberg, qui ont accepté la charge de rapporteur – pour l'intérêt et le temps qu'ils ont bien voulu accorder à ce travail.

Enfin, que soient également remerciés tous les membres des projets Cristal et Coq de l'INRIA Rocquencourt. Ils m'ont fourni, au cours de ces quatre ans, un environnement de travail remarquable tant par son haut niveau scientifique que par son ambiance détendue.

Introduction

La programmation d'un ordinateur est souvent considérée, au pire, comme une pratique aux rites obscurs, ou au mieux, comme un art. L'une des explications de ce préjugé est la difficulté que l'on rencontre à réaliser un programme dénué d'erreurs. Pour faire de la programmation une science à part entière, on s'est donc attaché, depuis quelques décennies, à développer des méthodes rigoureuses pour garantir l'absence d'erreurs dans un programme.

Dans son acception la plus large, la notion d'erreur désigne les fameux « bugs » : un programme contient une erreur si son comportement ne correspond pas exactement à ce qui était attendu par son auteur. Pour garantir l'absence de telles erreurs, il faut donc d'abord donner une description, ou *spécification*, du comportement souhaité, puis démontrer, de façon mathématique, que le programme est conforme à celle-ci. L'écriture de la spécification ne peut pas, en général, être automatisée, puisqu'elle dépend de l'objectif recherché. Quant à la démonstration, elle contient essentiellement une explication du fonctionnement du programme ; elle peut donc être très complexe, et il est difficile de la faire écrire par une machine. La *preuve de programmes* reste donc actuellement une activité essentiellement humaine, même si une preuve peut être *vérifiée* par une machine.

Il existe une notion moins générale, mais cependant intéressante : *l'erreur d'exécution*. Un programme contient une erreur d'exécution si, lorsque la machine exécute ce programme, elle rencontre une instruction dénuée de sens : par exemple, additionner un entier et un booléen, ou encore lire une donnée à une adresse inexistante. Il est intéressant de pouvoir garantir l'absence de telles erreurs : à défaut d'être certain que le programme calcule le résultat souhaité, au moins a-t-on la garantie qu'il ne « plantera » pas. En considérant une classe d'erreurs plus restreinte, nous avons donc affaibli la garantie obtenue ; mais en contrepartie, nous pouvons maintenant espérer établir celle-ci de façon automatique, grâce au mécanisme de *typage*.

Typage – inférence de types

Ainsi, le typage est une forme simple de preuve de programmes. On y retrouve donc les deux étapes mentionnées plus haut : spécification et démonstration.

La spécification d'un programme est ici son *type*. Par exemple, on peut attribuer à l'opération élémentaire `+` le type $\texttt{int} \times \texttt{int} \rightarrow \texttt{int}$, indiquant ainsi qu'il s'agit d'une fonction à deux arguments entiers et dont le résultat est entier. Ce type décrit en partie le comportement de la fonction `+`. En particulier, il permet de détecter l'erreur d'exécution provoquée par le calcul de `3 + true`, puisqu'on constate que la paire `(3, true)`, de type $\texttt{int} \times \texttt{bool}$, n'est pas un argument acceptable pour la fonction `+`. Cependant, il ne s'agit là que d'une spécification partielle, puisque d'autres fonctions, par exemple la fonction `-`, ont le même type. Si le programmeur confond `+` et `-`, l'erreur ne sera donc pas détectée par le typage ; il ne s'agit pas d'une erreur d'exécution.

La démonstration du fait qu'un programme donné admet un type donné se fait à l'aide d'un jeu de *règles de typage*. Ces règles sont souvent dirigées par la syntaxe, c'est-à-dire qu'on associe une règle à chacune des constructions autorisées par le langage de programmation.

Par exemple, la règle associée à l'application de fonction aura typiquement la forme suivante :

$$\frac{f : \tau \to \tau' \qquad e : \tau}{f(e) : \tau'}$$

Cette règle indique que si l'expression e a le type τ attendu par la fonction f, alors l'application $f(e)$ est légale – elle ne provoquera pas d'erreur d'exécution – et son type est le type résultat de la fonction f, à savoir τ'. En combinant des instances des règles de typage, on forme des *dérivations de typage* ; une dérivation dont la conclusion est $e : \tau$ démontre que le programme e admet le type τ.

La forme des types et des règles de typage varie d'un système de types à l'autre. Cependant, tous partagent une propriété commune : la correction, c'est-à-dire la garantie que si un programme admet un type – en d'autres termes, s'il est *bien typé* – alors il ne provoquera aucune erreur d'exécution. Cette propriété doit bien sûr être énoncée formellement, puis prouvée. Pour l'énoncer, on devra définir la notion d'exécution, en fournissant une *sémantique* du langage de programmation considéré.

Notons, en passant, que le typage tel que nous l'avons défini est parfois qualifié de *fort* et de *statique*. Le premier de ces qualificatifs se réfère à la propriété de correction évoquée ci-dessus. Si un système de typage n'est pas fort, il est donc tout simplement « troué », puisqu'il ne remplit pas parfaitement sa mission de détection des erreurs. Le second s'oppose au typage dit *dynamique*, où les éventuelles erreurs sont détectées non pas à la compilation, mais pendant l'exécution, lorsqu'elles se produisent. Cette technique est certes plus flexible, mais beaucoup moins fiable, puisqu'on n'a plus la garantie d'éviter les erreurs d'exécution ; tout au plus se traduiront-elles par un abrupt message d'erreur, plutôt que par un véritable « plantage ». Nous considérons donc que tout typage se doit d'être fort et statique.

Enfin, nous avons mentionné, lorsque nous avons écarté la notion générale de preuve de programmes pour nous intéresser au typage, que l'avantage de celui-ci est son éventuelle automatisation. La plupart des systèmes de typage sont en effet suffisamment simples pour que la *vérification* du typage soit décidable. Cela signifie que si le programmeur fournit suffisamment d'indications sur le type des objets qu'il manipule, alors la machine est capable de vérifier automatiquement que ces indications sont correctes, et que le programme admet bien le type indiqué. Cette propriété paraît relativement anodine ; pourtant, certains systèmes classiques, comme certaines variantes de $F_{\leq}$ [42], n'en disposent pas. Mieux, on peut souhaiter réaliser *l'inférence de types*, c'est-à-dire, étant donné un programme dénué d'annotations de types, reconstituer automatiquement ces annotations. Ainsi, si le programme est bien typé, la machine calculera son type. Dans les systèmes de typage dotés de *polymorphisme*, un programme donné peut admettre plusieurs types ; l'algorithme d'inférence devra alors en produire le plus général, c'est-à-dire celui qui fournit du programme la spécification la plus précise. Le problème de l'inférence de types peut être difficile ; par exemple, dans le système F [24], la vérification est décidable, mais pas l'inférence. Cependant, quand elle est possible, celle-ci procure un confort indéniable, puisqu'elle permet au programmeur d'omettre toute annotation de types, pour se concentrer sur la logique de son programme. Parmi les langages à inférence utilisés en pratique, on trouve principalement la famille ML [13].

Dans cette thèse, nous nous intéressons à un système de typage pour lequel l'inférence de types est décidable. Nous rappelons l'algorithme d'inférence, et nous nous attachons principalement à *simplifier* les types inférés, pour améliorer l'efficacité de l'algorithme et la lisibilité de ses résultats. Ce système se caractérise en particulier par son emploi du *sous-typage*.

Sous-typage

Nous avons constaté qu'un système de typage, pour avoir un intérêt pratique, doit être décidable et correct. Or, il est bien connu que le problème de déterminer si un programme contient une erreur est indécidable. Donc, les programmes bien typés ne sont pas les programmes corrects ; en d'autres termes, il existe nécessairement des programmes dénués d'erreurs et néanmoins mal typés.

Aucun système n'est donc parfait ; tous rejettent des programmes corrects, mais trop complexes pour être reconnus comme tels. C'est là une des raisons pour lesquelles il existe de nombreux systèmes : on essaie d'imaginer des systèmes de plus en plus fins, de façon à imposer à l'utilisateur aussi peu de limitations que possible.

Parmi les diverses caractéristiques développées pour rendre les systèmes de typage de plus en plus expressifs, citons brièvement le *polymorphisme paramétrique*, c'est-à-dire la possibilité d'abstraire un type par rapport à une *variable de types*. Par exemple, on pourra attribuer à la fonction identité $\lambda x.x$ le *schéma de types* $\forall\alpha.\alpha \rightarrow \alpha$, ce qui signifie qu'elle a à la fois les types `int` $\rightarrow$ `int`, `bool` $\rightarrow$ `bool`, etc. On peut donc lui passer un argument de n'importe quel type. Ce trait est présent, sous une forme très générale, dans le système F ; malheureusement, l'inférence de types n'y est pas décidable. Pour permettre l'inférence, le langage ML se restreint donc au polymorphisme *de premier ordre*, où le quantificateur $\forall$ ne peut apparaître qu'en tête d'un type, et non à l'intérieur. C'est cette forme de polymorphisme qui sera conservée dans le système présenté ici.

L'introduction du *sous-typage*, proposée indépendamment par divers auteurs, par exemple Cardelli [12] et Mitchell [36], est une autre façon d'améliorer la finesse d'un système de typage. On se donne une relation d'ordre sur les types, et on dit que τ est un *sous-type* de τ' lorsque $\tau \leq \tau'$. On ajoute alors au système une règle supplémentaire, qui a en général l'aspect suivant :

$$\frac{e : \tau \qquad \tau \leq \tau'}{e : \tau'}$$

Cette règle, appelée *règle de sous-typage*, indique que si une expression e a le type τ, alors elle a *a fortiori* tous les types τ' dont τ est un sous-type. En d'autres termes, cette règle permet d'affaiblir le type d'une expression, en le remplaçant par un type moins précis. Par exemple, si on dispose des types de base `int` et `float` pour représenter les entiers et les flottants, on peut poser `int` $\leq$ `float`, c'est-à-dire déclarer que `int` est un sous-type de `float`. La règle de sous-typage permet alors de considérer toute expression de type `int` comme ayant également le type `float`. Cela nous autorise à passer un entier en argument à toute fonction attendant un flottant, sans devoir effectuer de conversion explicite : par exemple, si `x` est entier, on pourra écrire `log x` au lieu de `log (float_of_int x)`.

Par ailleurs, la relation `int` $\leq$ `float` induit des relations sur les types composés. Par exemple, une fonction capable de traiter un argument réel peut prétendre qu'elle n'accepte que des entiers ; et, au contraire, une fonction calculant un résultat entier peut prétendre renvoyer un réel. Ainsi, on peut poser `float` $\rightarrow$ `int` $\leq$ `int` $\rightarrow$ `float`. De façon plus générale, on aura $\tau_0 \rightarrow \tau_1 \leq \tau'_0 \rightarrow \tau'_1$ si et seulement si $\tau'_0 \leq \tau_0$ et $\tau_1 \leq \tau'_1$. On remarque que le sens de la relation est renversé en ce qui concerne τ_0 et τ'_0 ; on dira que le constructeur $\rightarrow$ est *contravariant* par rapport à son premier argument et *covariant* par rapport au second.

Jusqu'ici, la relation de sous-typage n'est non triviale que sur les types de base, et s'étend ensuite de façon régulière aux types composés. Dans ce cas, on parle parfois de sous-typage *atomique*. L'inférence de types dans un tel système a été beaucoup étudiée [36, 29, 48, 21] ; en particulier, on connaît diverses bornes de complexité en fonction de la forme de la relation de sous-typage considérée. Fuh et Mishra [22, 23] se sont intéressés au sous-typage atomique en conjonction avec le polymorphisme à la ML, et à la simplification des typages obtenus. Cependant, d'un point de vue pratique, se limiter à des relations entre types de base, comme `int` $\leq$ `float`, peut paraître d'intérêt relativement mineur. Si la notion de sous-typage a connu un tel succès au cours des dernières années, c'est principalement parce qu'elle semble jouer un rôle central dans la définition de systèmes de typage pour les *langages orientés objet*.

Ces langages proposent une philosophie légèrement différente des langages classiques. Au lieu de manipuler des données, on manipule des *objets*. Un objet est simplement une entité capable de répondre à certains *messages* par une valeur. Tous les objets ne répondent pas aux mêmes messages ; en particulier, l'envoi d'un message provoque une erreur d'exécution si l'objet ne sait pas y répondre. L'une des façons classiques de coder les objets consiste à les représenter par de simples *enregistrements* (*records* en anglais) où chaque champ correspond à un message. Pour reprendre un exemple célèbre, un objet « point » aura ainsi le type `{ x: int }`, tandis qu'un objet « point coloré » aura le type `{ x: int; c: color }`. Or, l'une des bases de la programmation avec objets est l'idée que l'on peut manipuler uniformément des objets de nature différente, pourvu qu'on leur envoie uniquement des messages reconnus par tous. Par exemple, on voudrait pouvoir appliquer la fonction $\lambda p.(p.\mathtt{x})$, qui envoie le message `x` à un point p, aux points comme aux points colorés. Dans un système à la ML, il serait alors nécessaire que le domaine de cette fonction soit unifiable avec `{ x: int }` et avec `{ x: int; c: color }`, ce qui n'est pas le cas. Le sous-typage fournit une solution à ce problème : on décrète que `{ x: int; c: color }` est un sous-type de `{ x: int }`. La fonction $\lambda p.(p.\mathtt{x})$ a le type `{ x: int }` $\rightarrow$ `int`. Donc, son domaine est un supertype commun à `{ x: int }` et à `{ x: int; c: color }`, ce qui indique qu'elle peut être appliquée aux points comme aux points colorés. La relation de sous-typage permet ainsi « d'oublier » qu'un objet accepte certains messages, et donc de le mélanger avec d'autres objets qui auraient eu sans cela un type différent. On peut ainsi appliquer une même fonction à ces objets, ou encore les stocker au sein d'une structure hétérogène. Cette flexibilité des systèmes de typage orientés objet est une forme de *polymorphisme*.

On notera que dans ce cas, la relation de sous-typage n'est plus atomique. Lorsque le sous-typage était engendré uniquement par les relations entre types de base, si deux types sans variables étaient dans la relation de sous-typage, alors ils avaient nécessairement la même structure ; c'est-à-dire qu'il s'agissait de deux arbres ne différant que par leurs feuilles. Or, ce n'est plus le cas ici, puisque nous avons maintenant `{ x: int; c: color }` $\leq$ `{ x: int }`, où l'arbre de gauche a deux branches et celui de droite une seule. D'un point de vue théorique, il n'est d'ailleurs pas nécessaire d'introduire ces types enregistrement pour définir une relation de sous-typage non atomique. Il suffit pour cela de doter l'ensemble des types d'un plus petit élément $\bot$ et d'un plus grand élément $\top$. On a alors, par exemple, $\bot \leq \top \rightarrow \bot$, et le sous-typage n'est pas atomique. Dans ces conditions, une contrainte de la forme $\alpha \leq \beta_0 \rightarrow \beta_1$ ne nous autorise pas à affirmer que α peut s'écrire $\alpha_0 \rightarrow \alpha_1$, parce que α peut également valoir $\bot$. Cela signifie qu'une contrainte de sous-typage ne peut plus se décomposer en un ensemble de contraintes entre variables et types de base, comme dans le cas atomique.

Ensuite, notons que le typage des objets exige typiquement le maniement de types récursifs, à cause du mécanisme d'héritage, qui permet à une méthode de faire référence à l'objet receveur du message à travers le mot-clef `self`. Par comparaison, dans le langage ML, toute contrainte d'unification récursive est illégale. Les types récursifs doivent être explicitement déclarés et nommés. Par conséquent, pour permettre l'inférence, une étiquette d'enregistrement donnée doit apparaître dans au plus une définition de type. Or, cette restriction n'est pas acceptable lorsqu'on programme en style orienté objet, puisqu'on souhaite précisément que des objets de types différents acceptent des messages communs. Nous n'utiliserons donc pas de types nommés, et nous accepterons les contraintes de sous-typage récursives. Notons que par conséquent, aucune déclaration de types ne sera nécessaire de la part de l'utilisateur.

Ainsi, nous sommes conduits à étudier un système où le sous-typage est non atomique, et où les contraintes de sous-typage sont éventuellement récursives. Différentes possibilités se présentent quant au langage et au système de types utilisés ; cependant, les algorithmes d'inférence associés à ces différents choix reposent sur la même idée directrice.

Inférence de types en présence de sous-typage

Comment effectuer l'inférence de types en présence de sous-typage ? L'ensemble des algorithmes d'inférence existants utilisent une approche commune, à base de *contraintes*. Le principe en est simple. D'ailleurs, l'algorithme d'inférence du langage ML peut lui-même être considéré comme un cas particulier de cette méthode générale. Ainsi, plutôt que de considérer ces algorithmes comme entièrement nouveaux, on peut les voir comme des généralisations d'un algorithme bien connu.

Voici ce principe, énoncé de façon abstraite. (On ignore ici le traitement du polymorphisme introduit par les constructions `let`.) On associe d'abord à chaque nœud du programme, c'est-à-dire à chacune de ses sous-expressions, une variable de types. Ensuite, de la structure syntaxique du programme, on déduit une série de relations, ou *contraintes*, entre ces variables. Si l'ensemble de contraintes obtenu admet une solution, alors le programme est bien typé ; dans le cas contraire, il est rejeté.

Dans le cas de ML, les contraintes manipulées sont des égalités entre types, et une contrainte est engendrée à chaque nœud d'application de fonction. En effet, le type de l'argument fourni à une fonction doit coïncider avec le type du paramètre attendu. En d'autres termes, si on considère l'application $(e_1^{\alpha_1}\, e_2^{\alpha_2})^{\alpha}$, où chaque sous-expression est annotée par la variable de types associée, alors on a nécessairement $\alpha_1 = \alpha_2 \rightarrow \alpha$. L'analyse du programme engendre donc autant de contraintes que celui-ci contient d'applications. Reste à déterminer si ces contraintes admettent une solution, ce que l'algorithme d'unification effectue de façon particulièrement efficace. De plus, cet algorithme calcule l'unificateur le plus général, qui décrit l'ensemble des solutions ; les contraintes ne sont donc jamais manipulées explicitement.

Lorsqu'on ajoute la règle de sous-typage, les choses se compliquent légèrement. En effet, lors d'une application de fonction, il n'est plus nécessaire que le type fourni coïncide avec le type attendu ; il suffit que le premier soit un sous-type du second. Par conséquent, l'application $(e_1^{\alpha_1}\, e_2^{\alpha_2})^{\alpha}$ donne lieu à la contrainte $\alpha_1 \leq \alpha_2 \rightarrow \alpha$. Il s'agit cette fois d'une contrainte de sous-typage, au lieu d'une contrainte d'égalité. L'analyse du programme engendre donc une conjonction de telles contraintes, et il suffit de déterminer si celle-ci admet une solution pour savoir si le programme est bien typé. Malheureusement – c'est là qu'est la principale nouveauté – l'algorithme d'unification ne s'applique plus. On utilise un algorithme de *clôture* pour décider si les contraintes sont ou non solubles. Par ailleurs, celles-ci n'admettent pas nécessairement de solution principale ; la seule façon connue de décrire l'ensemble des solutions est donc de fournir l'ensemble de contraintes lui-même. Ainsi, le schéma de types principal associé au programme devra contenir, sous forme explicite, l'ensemble de contraintes. On manipulera donc des schémas de la forme $\forall\bar{\alpha} \mid C.\tau$; un tel schéma indique que le programme a le type τ, pourvu que les variables $\bar{\alpha}$ vérifient la conjonction de contraintes C.

Même si nous n'avons parlé jusqu'ici que de typage et d'inférence de types, il est important de prendre conscience que les mêmes algorithmes pourraient être décrits en termes d'interprétation abstraite. En effet, chaque application de fonction donnant naissance à une contrainte, le graphe des contraintes n'est autre qu'une représentation approchée du flot de données correspondant au programme. Par exemple, en l'absence de polymorphisme, notre système serait équivalent à une 0-CFA, comme le font remarquer Palsberg, O'Keefe et Smith [40, 41]. On peut également comparer notre inférence de types à une analyse abstraite modulaire, comme la SBA [26, 20]. Dans les deux cas, le résultat de l'analyse est une description du comportement du programme sous forme d'un ensemble de contraintes. Le typage se distingue traditionnellement de l'interprétation abstraite par une plus grande simplicité et une moins grande finesse ; ici, du fait de la précision des systèmes de typage considérés, la distinction devient floue, et les deux points de vue sont possibles. Par ailleurs, notons que le parallèle avec l'interprétation abstraite permet de comprendre de façon imagée certaines de nos simplifications : par exemple, le *dépoussiérage* consiste simplement à simuler l'écoulement du flot de données à travers le graphe de contraintes, et à détruire toutes les arêtes – ou contraintes – inutilisées.

L'utilisation de contraintes est donc le point commun aux différents systèmes d'inférence en présence de sous-typage. Cependant, ces systèmes diffèrent en de nombreux points. D'abord, le langage de programmation considéré peut varier. Ensuite, la syntaxe des contraintes, ainsi que la sémantique qui leur est attribuée, diffèrent d'un système à l'autre. Les algorithmes de résolution et de simplification de contraintes, qui en dépendent directement, seront donc largement différents.

Quel système choisir ?

Nous avons expliqué notre intérêt pour l'inférence de types dans un système où les contraintes de sous-typage sont non atomiques et récursives. Une palette de choix assez étendue s'offre à nous, dont voici quelques représentants notables.

Dans cette thèse, nous choisissons de nous intéresser au λ-calcul avec `let`, c'est-à-dire à un noyau fonctionnel extrêmement simple – le même que celui utilisé par ML. En effet, la notion de sous-typage n'est pas propre aux langages orientés objet ; il paraît donc naturel de l'étudier dans le cadre bien compris des langages fonctionnels, plutôt que de devoir choisir l'une des diverses propositions de calculs orientés objet. De plus, il est important de ne pas écarter la notion de fonction, puisqu'elle exige de maîtriser le phénomène de contravariance. Le système de types est réduit à sa plus simple expression : les types bruts sont formés à l'aide des constructeurs $\bot$, $\top$, et $\rightarrow$. Ce système, bien que restreint, est d'une complexité intrinsèque non négligeable ; aussi, les résultats obtenus s'étendront aisément à un système plus riche, muni par exemple de types de base, de types enregistrements et variantes extensibles, de types références, etc. On pourra alors s'intéresser à nouveau à la notion d'objet, soit à travers un codage, soit en appliquant nos techniques à un langage muni d'objets primitifs. Le problème de l'inférence de types pour ce système a été résolu initialement par Eifrig, Smith et Trifonov [16], dans le but de l'appliquer ensuite à des langages orientés objet.

A l'opposé, Abadi et Cardelli [1, 2] proposent ce que l'on pourrait considérer comme un langage orienté objet canonique. Il s'agit d'un calcul extrêmement simple, dans lequel la notion d'objet est primitive. Abadi et Cardelli étudient successivement différents systèmes, de complexité croissante, pour ce langage. Tous disposent d'une règle de sous-typage qui permet, comme attendu, « d'oublier » la présence de certaines méthodes, et qui est donc non atomique. Le problème de l'inférence de types, pour le plus puissant des systèmes dits *de premier ordre*, a été résolu par Palsberg [39], qui utilise un algorithme basé sur le même principe que celui d'Eifrig, Smith et Trifonov. Cependant, les deux systèmes présentent une différence fondamentale : tandis que le constructeur $\rightarrow$ présent chez Eifrig *et al.* est contravariant par rapport à son premier argument et covariant par rapport au second, les types objet d'Abadi et Cardelli sont *invariants*, c'est-à-dire qu'une relation de sous-typage entre deux types objet entraîne *l'égalité* de leurs composants communs. Comme le montre Henglein [27], cette particularité permet d'améliorer l'efficacité de l'algorithme d'inférence. Pour simuler de façon satisfaisante la notion de type de fonction, Abadi et Cardelli introduisent des types universellement et existentiellement quantifiés ; mais cela se fait au détriment de l'inférence de types.

Aiken, Wimmers et Lakshman [6, 7] s'intéressent également au λ-calcul avec `let`, mais adoptent un système de types fort différent. Dans notre système, comme en ML, les types bruts sont des termes, ordonnés par la relation de sous-typage. Aiken *et al.* se placent dans le *modèle idéal* [35], c'est-à-dire que les types bruts sont des parties du modèle, ordonnées par inclusion ensembliste. Dans les deux cas, l'inférence de types passe par un problème de résolution de contraintes ; cependant, dans le premier cas, les contraintes s'expriment dans un formalisme dédié, tandis que dans le second, on est dans le cadre de la théorie générale des contraintes ensemblistes. Il en résulte un système plus expressif, et plus complexe. La simplicité de notre système nous permet de formaliser relativement aisément nos méthodes de simplification, tandis que les algorithmes implémentés dans Illyria [3] restent non documentés. Quant à la perte d'expressivité, elle est en partie compensée par l'ajout de types complexes à notre système. Le système d'Aiken *et al.* reste cependant plus fin ; par exemple, il nous est actuellement impossible de typer les filtrages de façon aussi précise que [7].

Felleisen et Flanagan [19, 20, 18] manipulent également des contraintes ensemblistes. Ici, il ne s'agit plus de typage, mais d'une analyse ensembliste (*set-based analysis*) ; cependant, comme on l'a remarqué plus haut, cette distinction n'est pas toujours utile. Le système de Felleisen et Flanagan présente plusieurs points communs avec le nôtre ; notre algorithme de *minimisation* en est d'ailleurs inspiré. La principale différence réside probablement dans le traitement des fonctions. En effet, dans notre système, le domaine d'une fonction est le type de son argument *formel*, c'est-à-dire le type qu'elle est capable de traiter ; c'est pourquoi le constructeur $\rightarrow$ doit être contravariant vis-à-vis de son premier argument. Chez Felleisen et Flanagan, au contraire, le domaine d'une fonction représente son argument *effectif*, c'est-à-dire les valeurs passées à cette fonction au cours du programme ; c'est pourquoi le destructeur dom est covariant. De plus, la sémantique des contraintes autorise l'application de ce destructeur à des objets autres que des fonctions. La solubilité des contraintes n'implique donc plus la correction des programmes ; un test supplémentaire sera nécessaire pour garantir celle-ci. Ces deux particularités du destructeur dom modifient significativement la théorie ; en particulier, chez Felleisen et Flanagan, tout ensemble de contraintes admet une plus petite solution, et l'implication de contraintes est décidable.

Müller, Niehren et Podelski [37] s'intéressent à l'analyse statique du langage Oz. L'ensemble des valeurs possibles pour chaque variable du programme est approximé par un ensemble de termes infinis. Encore une fois, ces ensembles sont reliés par des contraintes d'inclusion, qu'il faut donc résoudre pour déterminer si le programme est bien typé. Pour que le programme soit accepté, il faudra non seulement que les contraintes admettent une solution, mais que celle-ci associe à chaque variable un ensemble non vide. Pour cette raison, Müller *et al.* interprètent les contraintes dans le modèle des ensembles de termes *non vides*. Cette décision modifie les propriétés logiques du système – en particulier l'implication de contraintes – qui gagne alors en simplicité. Ce système présente, en principe, des points communs avec ceux mentionnés précédemment ; notons toutefois que tous les constructeurs y sont covariants.

Enfin, si nous avons présenté le sous-typage comme un mécanisme permettant de typer les langages orientés objet, signalons qu'il n'est pas la seule possibilité. En effet, il est possible de traiter les objets au sein d'un langage classique comme ML, à condition d'enrichir le système de types par des *variables de rangée* [45]. C'est la solution adoptée par le langage Objective Caml [47]. Ces deux solutions ne sont d'ailleurs pas mutuellement exclusives ; nous discutons, dans la section 14.5, de l'interaction entre variables de rangée et sous-typage.

Simplification

Les détails de l'algorithme d'inférence, pour le système qui nous intéresse, ont été donnés initialement par Eifrig, Smith et Trifonov [16], approximativement à l'époque du début de cette thèse. Cependant, l'algorithme était encore loin d'être utilisable en pratique. En effet, le nombre de contraintes engendrées est linéaire en la taille du programme, puisque chaque application de fonction en crée une. (Il est même exponentiel en théorie, puisque la construction `let` permet de dupliquer l'ensemble des contraintes.) Or, l'algorithme de clôture travaille en temps cubique ; cela rend donc l'inférence de types très lente. De plus, les schémas de types inférés, qui contiennent, rappelons-le, l'ensemble de contraintes, sont très peu lisibles. Il devient donc difficile pour l'utilisateur d'utiliser le moteur d'inférence comme une aide à la compréhension de son programme.

Il était donc nécessaire, pour améliorer efficacité et lisibilité, de développer des méthodes de *simplification* de contraintes. Quelques méthodes existaient déjà, mais seulement pour des systèmes sensiblement plus simples, comme celui de Fuh et Mishra [23], où l'on peut se contenter de manipuler des contraintes entre variables et types de base. Il fallait donc mettre au point des algorithmes de simplification adaptés à notre système, plus complexe ; mais aussi, pour faciliter la définition et la preuve de ces algorithmes, donner de celui-ci une formulation aussi simple et élégante que possible.

Dans sa forme actuelle, le système est fondé sur une relation de *comparaison polymorphe* *entre schémas de types*, notée $\leq^\forall$. La définition de cette relation fait appel à la sémantique des contraintes, c'est-à-dire qu'elle s'exprime en termes des solutions de l'ensemble de contraintes apparaissant dans chacun des schémas. Cette relation apparaît dans la règle de sous-typage, que l'on pourrait écrire, de façon simplifiée :

$$\frac{e : \sigma \qquad \sigma \leq^\forall \sigma'}{e : \sigma'}$$

Elle joue donc, intuitivement, le même rôle que la relation de sous-typage évoquée plus haut, mais se situe au niveau des schémas de types, c'est-à-dire qu'elle tient compte des quantificateurs universels apparaissant en tête de σ et σ'. Cette relation constitue la justification théorique de toutes nos méthodes de simplification : en effet, pour « simplifier » un schéma de types σ, il suffit de le remplacer par un schéma σ' équivalent au sens de cette relation, c'est-à-dire tel que $\sigma =^\forall \sigma'$. Quels sont les critères permettant d'affirmer que σ' est plus « simple » que σ ? Si le but recherché est la lisibilité, le critère sera probablement la taille de la représentation textuelle de σ'. Cependant, si le but recherché est l'efficacité, la représentation la plus succincte n'est pas forcément la plus aisée à manipuler. On cherchera donc toujours à minimiser la taille de σ', mais en respectant certains invariants utiles à nos algorithmes. Il est donc important de séparer ces deux objectifs.

Une méthode de simplification est donc simplement un algorithme capable de transformer un schéma de types donné en un schéma équivalent. Nous présenterons trois méthodes, lesquelles se composent naturellement pour donner une combinaison efficace et puissante. La *canonisation* fait en sorte que chaque variable admette au plus une borne construite, en introduisant des variables auxiliaires. Cette propriété est indispensable pour formuler l'algorithme de minimisation ; de plus, on peut voir la canonisation comme une façon d'augmenter le partage de la structure entre termes. Le *dépoussiérage* (*garbage collection* en anglais) détermine, par un calcul de point fixe, quelles contraintes ont réellement une incidence sur la dénotation du schéma de types, et élimine toutes les autres. Enfin, la *minimisation* force un partage maximal (ou presque) entre les nœuds du graphe de contraintes, par un procédé semblable à la minimisation d'un automate fini.

Pour fonctionner de façon optimale, ces algorithmes utilisent deux invariants. *L'invariant des petits termes* impose des restrictions assez importantes sur la forme des termes manipulés. Intuitivement, il exige qu'à chaque nœud, dans un terme, soit associé une variable de types. Ainsi, le partage entre nœuds se réduit simplement à l'identification de variables ; on pourra établir un rapprochement avec la présentation sous forme de multi-équations des problèmes d'unification. *L'invariant de mono-polarité* exige qu'une même variable de types ne soit pas utilisée à la fois pour représenter un paramètre et un résultat ; il a divers effets bénéfiques, en théorie comme en pratique. Pour respecter chacun de ces deux invariants, il suffit de modifier légèrement la formulation des règles d'inférence, ce que nous ferons le moment venu.

Par ailleurs, nous étudions la sémantique des contraintes, c'est-à-dire la façon de décider diverses sortes de propositions logiques ayant trait aux solutions des ensembles de contraintes. De cette étude ressortent trois algorithmes. Le premier permet de décider si une conjonction de contraintes donnée admet une solution ; il sert à déterminer, une fois les contraintes obtenues par analyse du programme, si celui-ci est bien typé. Le second tente de décider l'implication de contraintes ; le troisième, qui le généralise, de décider la relation $\leq^\forall$. Toutefois, ces deux derniers sont incomplets. Ils ne sont pas utilisés en pratique ; ils servent, sur le papier, à prouver la correction des méthodes de simplification décrites ci-dessus. Ainsi, nos méthodes de simplification construisent directement un schéma simplifié équivalent au schéma original ; c'est un progrès très important par rapport à l'approche qui consiste à produire un schéma de façon heuristique, et à utiliser l'un de ces deux algorithmes pour déterminer s'il est équivalent au schéma initial. Notons néanmoins que le dernier algorithme est utilisé pour comparer les schémas de types inférés aux signatures fournies par l'utilisateur, dans le cadre du système de modules.

Il est intéressant de remarquer que toute la partie de la théorie décrite ci-dessus, c'est-à-dire l'ensemble du système de simplification de contraintes, découle uniquement de la sémantique des contraintes, c'est-à-dire de leur interprétation dans l'ensemble des types bruts. En particulier, elle est indépendante du langage et des règles de typage choisis. Elle est donc immédiatement applicable à d'autres langages, pourvu que la sémantique des contraintes soit inchangée.

Les aspects de notre théorie qui dépendent du langage sont donc simplement la définition même du langage, sa sémantique et ses règles de typage, accompagnées bien sûr d'une preuve de correction. Dans notre cas, cette dernière est relativement simple, à condition de la formuler de façon appropriée – la puissance de la relation $\leq^\forall$ pose quelques problèmes, et nous serons amenés à introduire un jeu de règles de typage auxiliaires pour les résoudre.

La contribution de cette thèse se situe donc principalement dans le domaine de la simplification. L'algorithme initial [16] prouvait la décidabilité de l'inférence, mais n'était guère utilisable en pratique. Nous proposons ici un système efficace et homogène, formé de plusieurs algorithmes complémentaires, manipulant des données sous une forme adaptée. Historiquement, notre travail a été réalisé parallèlement à celui de Trifonov et Smith ; aussi se complètent-ils mutuellement. Certaines idées ont été proposées de façon entièrement indépendante, tandis que dans d'autres cas, l'un des groupes s'appuyait sur les travaux de l'autre pour les améliorer ensuite. Cette thèse présente le système sous sa forme la plus aboutie, en s'efforçant d'indiquer à qui sont dûs les principaux concepts. En résumé, dans [43], nous présentons *l'élimination des contraintes inaccessibles*, et introduisons la relation d'implication de contraintes, ainsi que son axiomatisation. Dans [49], Trifonov et Smith reformulent le système de typage à l'aide du λ-*lifting*, ce qui leur permet de généraliser ces deux notions, obtenant ainsi le dépoussiérage et la relation de comparaison polymorphe entre schémas de types. De plus, ils y présentent l'algorithme de canonisation. Enfin, dans cette thèse, nous introduisons l'algorithme de minimisation, et traitons un certain nombre de points qui contribuent de façon significative à l'obtention d'un système de simplification complet et homogène. En particulier, nous améliorons l'efficacité de l'algorithme de canonisation en le combinant avec une phase de dépoussiérage ; nous reformulons les règles d'inférence de façon à respecter l'invariant des petits termes et l'invariant de mono-polarité, et nous montrons qu'efficacité et lisibilité sont des buts contradictoires.

Plan

La première partie de cette thèse présente le système de typage étudié. Comme nous l'avons indiqué plus haut, celui-ci est lui-même formé de deux composants essentiellement indépendants : un langage de contraintes et un langage de programmation, chacun étant défini par sa syntaxe et sa sémantique. Le second est accompagné d'un jeu de règles de typage, dont la formulation fait appel au premier.

Le langage de contraintes peut être vu comme un système logique : il permet d'exprimer des formules, lesquelles sont interprétées dans un modèle. Le modèle est l'ensemble des *types bruts*, c'est-à-dire des types sans variables, auquel la relation de sous-typage confère une structure de treillis. L'introduction de *types* avec variables, puis de *contraintes* entre types et de *schémas de types*, permettra d'exprimer des formules concernant l'existence d'une solution, l'implication de contraintes, ou la comparaison de deux schémas de types.

Une fois le langage de contraintes défini, vient l'exposé du *système de typage* à proprement parler. Il comprend la définition du langage et de sa sémantique, la présentation des règles de typage et de l'algorithme d'inférence de types, et la preuve de correction du typage vis-à-vis de la sémantique.

La deuxième partie de cette thèse contient l'essentiel de notre théorie. On y étudie d'abord la décidabilité des trois problèmes logiques mentionnés ci-dessus : solubilité de contraintes, implication de contraintes et comparaison de schémas de types. Vient ensuite la présentation de nos trois principales méthodes de simplification, à savoir dépoussiérage, canonisation et minimisation. Enfin, l'invariant des petits termes et l'invariant de monopolarité se voient chacun dédier un chapitre. Chacun de ces invariants est introduit au moment le plus opportun, et est supposé en vigueur dans la suite de l'exposé, ce qui contribue à simplifier celle-ci. On aurait parfois pu se passer de ces hypothèses simplificatrices, et obtenir ainsi des énoncés légèrement plus généraux ; mais cela aurait été au détriment de la simplicité des preuves.

Enfin, la troisième partie examine le problème sous un jour plus pratique. Elle passe d'abord en revue un certain nombre d'extensions du système, écartées de la présentation théorique par souci de simplicité, mais qui ne posent aucune difficulté particulière. Elle décrit ensuite la façon dont les acquis théoriques de la seconde partie conduisent à une implémentation, et en mesure les performances. Enfin, un chapitre est consacré à l'examen de plusieurs problèmes résiduels, parmi lesquels le traitement des programmes impératifs et un certain manque d'efficacité, et à l'esquisse de plusieurs solutions.

Première partie

Présentation du système

Chapitre 1

Types bruts

PARMI LES DIVERSES FORMES DE TYPES INTRODUITES DANS CETTE THÈSE, la notion de *type brut* est la plus simple et la plus essentielle. Il s'agit des arbres réguliers formés à partir des constructeurs élémentaires $\bot$, $\top$ et $\rightarrow$. L'ensemble des types bruts est muni d'un ordre partiel, appelé *sous-typage*, qui en fait un treillis. (Nous avons choisi le terme de *type brut* plutôt que celui de *type de base*, car nous avons préféré réserver ce dernier aux types de données élémentaires comme `int`, `bool`, etc.)

Pour les besoins de l'inférence de types, nous définirons plus tard la notion de *type*. Nous y introduirons des *variables de types*, ainsi que deux constructeurs supplémentaires $\sqcup$ et $\sqcap$. Les types seront manipulés de façon formelle par l'intermédiaire de *graphes de contraintes*. En termes logiques, les graphes de contraintes sont des formules, interprétées dans le modèle des types bruts.

Le traitement du polymorphisme demandera ensuite l'introduction de *schémas de types*. De façon grossière, un schéma de types peut être interprété comme l'ensemble de ses instances, c'est-à-dire une partie de l'ensemble des types bruts. Un schéma de types est *plus général* qu'un autre si l'interprétation du premier, dans l'ensemble des types bruts, est un sur-ensemble de celle du second. Le treillis des types bruts constitue donc le fondement logique de notre théorie.

Pour simplifier autant que possible notre théorie, nous réduisons le treillis des types bruts à sa plus simple expression. Cela diminue la complexité apparente des énoncés et des preuves, sans rien changer à leur principe. Un treillis de types bruts beaucoup plus riche est décrit, de façon moins détaillée, au chapitre 14.

Nous définissons d'abord l'ensemble des types bruts (section 1.1), puis la relation de sous-typage sur cet ensemble (section 1.2).

1.1 Types bruts

Cette section définit les types bruts comme étant des arbres réguliers. Pourquoi choisir les arbres réguliers, plutôt que les arbres finis ? En ML, par exemple, les types bruts sont finis. Les contraintes d'unification récursives sont rejetées par le *test d'occurrence* (*occur check* en anglais), et les types de données récursifs doivent être explicitement déclarés par l'utilisateur. Cependant, il serait facile (quoique pas nécessairement désirable d'un point de vue pratique) d'étendre la théorie de ML au cas de types bruts réguliers. Pour ML, les deux choix sont donc possibles. Dans notre cas, cependant, les contraintes d'unification sont remplacées par des contraintes de sous-typage. Il devient donc possible pour une contrainte récursive d'avoir une solution finie. Rejeter les contraintes n'ayant aucune solution finie devient alors plus difficile et ne semble plus naturel. Par ailleurs, d'un point de vue pratique, il semble intéressant de pouvoir inférer le type de programmes maniant des structures de données récursives, sans exiger de déclarations de types explicites. Par exemple, cela est utile dans le cadre de l'inférence de types pour des langages orientés objets.

On peut également considérer l'autre extrême : pourquoi se limiter aux arbres réguliers, et ne pas choisir des arbres infinis arbitraires ? On vérifie en fait que l'utilisation de ces derniers ne modifierait pas les propriétés logiques fondamentales de notre système. Plus précisément, un graphe de contraintes admet une solution régulière si et seulement si il admet une solution (éventuellement irrégulière). Mieux, toute assertion d'implication de contraintes a la même valeur booléenne dans les deux modèles. La démonstration de ces faits dépasse le cadre de cette thèse ; elle est cependant relativement simple, et utilise des propriétés topologiques similaires à celles que l'on trouve dans la section 2.6.1. Pour conclure, il semble que la restriction aux arbres réguliers n'entraîne aucune perte de puissance, tout en procurant un gain de simplicité.

Définition 1.1 *Soit Σ_b la* signature brute *constituée de $\bot$ et $\top$ avec l'arité* 0 *et de $\rightarrow$ avec l'arité* 2. *Un* chemin *p est une suite finie de* 0 *et de* 1, *c'est-à-dire un élément de $\{0, 1\}^*$. Le chemin vide est noté ϵ. La longueur d'un chemin p est notée $|p|$. Sa* parité *$\pi(p)$ est le nombre de* 0 *qu'il contient, modulo* 2. *Un* arbre brut *τ est une fonction partielle des chemins vers Σ_b, de domaine non vide et clos par préfixe, telle que $\tau(p0)$ et $\tau(p1)$ sont définis ssi $\tau(p) = \rightarrow$. L'ensemble des arbres bruts est noté $\mathbb{T}_\infty$. Etant donné $p \in \mathrm{dom}(\tau)$, le* sous-arbre de τ de racine p *est l'arbre $q \mapsto \tau(pq)$. Un arbre est* fini *ssi son domaine est fini. Un arbre est* régulier *ssi il a un nombre fini de sous-arbres. Un* type brut *est un arbre brut régulier. L'ensemble des types bruts est noté $\mathbb{T}$.*

Les arbres réguliers peuvent être représentés de façon finie par des automates d'arbres. Cette équivalence sera utilisée dans la définition formelle des opérations $\sqcup$ et $\sqcap$ sur les types bruts.

Définition 1.2 *Un* automate de termes *sur Σ_b est un quadruplet $\mathcal{A} = (Q, q_0, \delta, l)$, où*

- *Q est un ensemble fini d'*états,
- *$q_0 \in Q$ est l'*état initial,
- *$\delta : Q \times \{0, 1\} \rightarrow Q$ est une* fonction de transition *(partielle),*
- *$l : Q \rightarrow \Sigma_b$ est une* fonction d'étiquetage,

tel que pour tout état $q \in Q$ et pour tout $i \in \{0, 1\}$, $\delta(q, i)$ est défini ssi $l(q) = \rightarrow$.

δ s'étend naturellement à une fonction partielle $\hat{\delta} : Q \times \{0, 1\}^* \rightarrow Q$. Le terme $\tau_{\mathcal{A}}$ associé à l'automate $\mathcal{A}$ est $p \mapsto l(\hat{\delta}(q_0, p))$.

Proposition 1.1 *Un arbre τ est régulier ssi il existe un automate $\mathcal{A}$ tel que $\tau = \tau_{\mathcal{A}}$.*

Démonstration. En établissant un isomorphisme entre les états de $\mathcal{A}$ et les sous-arbres de $\tau_{\mathcal{A}}$. □

Du point de vue de l'écriture, il est peu agréable de devoir considérer les types comme des arbres, ou comme des automates ; nous introduisons donc quelques notations usuelles à propos des types.

Définition 1.3 *Soit $\bot$ (resp. $\top$) l'arbre τ tel que $\mathrm{dom}(\tau) = \{\epsilon\}$ et $\tau(\epsilon) = \bot$ (resp. $\top$). Si τ_0 et τ_1 sont des arbres, $\tau_0 \rightarrow \tau_1$ est l'arbre τ défini par $\tau(\epsilon) = \rightarrow$, $\tau(0p) = \tau_0(p)$ et $\tau(1p) = \tau_1(p)$.*

1.2 Sous-typage brut

Nous avons défini l'ensemble des types bruts. Nous allons à présent le munir d'un ordre partiel, appelé *sous-typage*, et montrer que cela en fait un treillis. Techniquement, il existe plusieurs façons équivalentes de définir le sous-typage sur les arbres réguliers : par approximations finies, par co-induction ou, comme ici, par quantification sur les chemins. Chacun de ces choix donne la même relation d'ordre, caractérisée par la proposition 1.2. De même, la définition technique des opérateurs $\sqcup$ et $\sqcap$ est peu intéressante ; leur comportement est caractérisé par la proposition 1.5.

Définissons d'abord le sous-typage.

Définition 1.4 *On munit Σ_b d'une relation d'ordre en posant $\bot \leq_b \rightarrow \leq_b \top$. Cela en fait un treillis ; ses opérateurs de borne supérieure et de borne inférieure sont notés $\sqcup_b$ et $\sqcap_b$, respectivement. De plus, on notera $\leq_b^0$ pour $\leq_b$ et $\leq_b^1$ pour la relation inverse.*

Etant donnés deux types bruts τ et τ', on dit que τ est un sous-type *de τ', et l'on écrit $\tau \leq \tau'$, ssi*

$$\forall p \in \mathrm{dom}(\tau) \cap \mathrm{dom}(\tau') \quad \tau(p) \leq_b^{\pi(p)} \tau'(p)$$

Comme mentionné plus haut, on se fera une meilleure idée du sous-typage grâce à la caractérisation suivante. Elle indique que $\bot$ est le plus petit élément, $\top$ le plus grand, et que le sous-typage se propage structurellement le long du constructeur $\rightarrow$. Bien sûr, celui-ci est contravariant sur son domaine et covariant sur son codomaine.

Proposition 1.2 *L'assertion $\tau \leq \tau'$ est vérifiée ssi au moins l'une des assertions suivantes est vraie :*

- *$\tau = \bot$;*
- *$\tau' = \top$;*
- *$\exists \tau_0 \tau_1 \tau_0' \tau_1' \quad \tau = \tau_0 \rightarrow \tau_1$, $\tau' = \tau_0' \rightarrow \tau_1'$, $\tau_0' \leq \tau_0$ et $\tau_1 \leq \tau_1'$.*

Démonstration. Supposons d'abord que $\tau = \tau_0 \rightarrow \tau_1$ et $\tau' = \tau_0' \rightarrow \tau_1'$. Alors $\tau \leq \tau'$ est, par définition, équivalent à

$$\forall p \in \mathrm{dom}(\tau) \cap \mathrm{dom}(\tau') \quad \tau(p) \leq_b^{\pi(p)} \tau'(p)$$

Cela peut être écrit

$$\forall i \in \{0,1\} \quad \forall q \in \mathrm{dom}(\tau_i) \cap \mathrm{dom}(\tau_i') \quad \tau(iq) \leq_b^{\pi(iq)} \tau'(iq)$$

Puisque $\tau(iq) = \tau_i(q)$ et $\tau'(iq) = \tau_i'(q)$, cela est équivalent à $\tau_0' \leq \tau_0 \wedge \tau_1 \leq \tau_1'$ (choisir successivement $i = 0$ et $i = 1$).

En ce qui concerne le cas général, supposons donnés τ et τ'. Le couple $(\tau(\epsilon), \tau'(\epsilon))$ peut prendre 9 valeurs différentes. Sur ces 9 cas, 4 sont des cas où $\tau \leq \tau'$ est faux, 4 sont des cas où $\tau = \bot$ ou $\tau' = \top$, et le dernier est celui que nous avons traité ci-dessus. □

Voici maintenant une autre caractérisation du sous-typage, basée sur une suite d'approximations finies, qui nous sera utile dans plusieurs preuves.

Définition 1.5 *Soit $\leq_0$ la relation binaire uniformément vraie sur les types bruts. Pour $k \geq 1$, on définit $\leq_k$ par*

- *Si $\tau = \tau_0 \rightarrow \tau_1$ et $\tau' = \tau_0' \rightarrow \tau_1'$, alors $\tau \leq_k \tau'$ ssi $\tau_0' \leq_{k-1} \tau_0$ et $\tau_1 \leq_{k-1} \tau_1'$.*
- *Sinon, $\tau \leq_k \tau'$ ssi $\tau(\epsilon) \leq_b \tau'(\epsilon)$.*

On vérifie aisément que $(\leq_k)_{k \geq 0}$ est une suite décroissante de pré-ordres. En fait, son intersection est précisément $\leq$, comme le montre la proposition suivante.

Proposition 1.3 *$\tau \leq \tau'$ est équivalent à*

$$\forall k \geq 0 \quad \tau \leq_k \tau'$$

Pour cette raison, la relation $\leq$ sera parfois notée $\leq_\infty$.

Démonstration. Nous allons prouver, par induction, que pour tout $k \geq 0$, $\tau \leq_k \tau'$ est équivalent à

$$\forall p \in \mathrm{dom}(\tau) \cap \mathrm{dom}(\tau') \quad |\, p \,| < k \Rightarrow \tau(p) \leq_b^{\pi(p)} \tau'(p)$$

où $|\, p \,|$ est la longueur du chemin p. Cela est immédiat pour $k = 0$. Supposons que le résultat soit vérifié au rang $k - 1$, où $k \geq 1$. Distinguons deux cas.

Supposons d'abord $\tau = \tau_0 \rightarrow \tau_1$ et $\tau' = \tau_0' \rightarrow \tau_1'$. Alors $\tau \leq_k \tau'$ est équivalent à $\tau_0' \leq_{k-1} \tau_0$ et $\tau_1 \leq_{k-1} \tau_1'$. Par hypothèse d'induction, la première de ces assertions est équivalente à

$$\forall q \in \mathrm{dom}(\tau_0') \cap \mathrm{dom}(\tau_0) \quad |\, q \,| < k - 1 \Rightarrow \tau_0'(q) \leq_b^{\pi(q)} \tau_0(q)$$

La seconde peut être réécrite de façon similaire, et nous pouvons les recombiner pour former

$$\forall i \in \{0, 1\} \quad \forall q \in \mathrm{dom}(\tau_i) \cap \mathrm{dom}(\tau_i') \quad |\, iq \,| < k \Rightarrow \tau(iq) \leq_b^{\pi(iq)} \tau'(iq)$$

qui est équivalente à notre but.

Dans le cas contraire, $\tau \leq_k \tau'$ est, par définition, équivalent à $\tau(\epsilon) \leq_b \tau'(\epsilon)$. Par ailleurs, $\mathrm{dom}(\tau)$ ou $\mathrm{dom}(\tau')$ est égal à $\{\epsilon\}$, donc le but est également équivalent à $\tau(\epsilon) \leq_b \tau'(\epsilon)$.

L'induction est terminée. De ce résultat, il découle clairement que l'intersection de $(\leq_k)_{k \geq 0}$ coïncide avec la relation de sous-typage. □

Nous pouvons à présent établir une propriété fondamentale de la relation de sous-typage :

Proposition 1.4 *L'ensemble $\mathbb{T}$ des types bruts, muni de la relation de sous-typage, est un treillis. Ses opérateurs de borne supérieure et de borne inférieure sont notés $\sqcup$ et $\sqcap$, respectivement.*

Démonstration. D'abord, montrons que $\leq$ est un ordre, ce que nous n'avons pas encore fait. $\leq$ est l'intersection de $(\leq_k)_{k \geq 0}$, qui est une suite de pré-ordres, donc est un pré-ordre. Pour montrer que cette relation est antisymétrique, supposons $\tau \leq \tau' \leq \tau$. Cela se traduit par $\forall p \in \mathrm{dom}(\tau) \cap \mathrm{dom}(\tau') \quad \tau(p) = \tau'(p)$, i.e. τ et τ' coïncident sur l'intersection de leurs domaines. Montrons que leurs domaines coïncident. Supposons que $\mathrm{dom}(\tau) \setminus \mathrm{dom}(\tau')$ soit non vide (l'autre cas étant symétrique). Nous pouvons en isoler un élément p de longueur minimale. Puisque $\epsilon \in \mathrm{dom}(\tau')$, nous avons $|\, p \,| > 0$ et nous pouvons écrire $p = qi$ où $i \in \{0, 1\}$. Parce que $\mathrm{dom}(\tau)$ est clos par préfixe, q lui appartient nécessairement ; et puisque p est minimal, nécessairement $q \in \mathrm{dom}(\tau')$. Il en découle que $\tau(q) = \tau'(q)$. Rappelons à présent que tout arbre σ doit satisfaire la propriété suivante : pour tout chemin r, $\sigma(r0)$ et $\sigma(r1)$ sont définis ssi $\sigma(r) = \rightarrow$. Puisque $\tau(qi)$ est défini, il s'ensuit que $\rightarrow = \tau(q) = \tau'(q)$ donc $\tau'(qi) = \tau'(p)$ est également défini. Nous avons obtenu une contradiction avec $p \notin \mathrm{dom}(\tau')$. Ainsi, $\mathrm{dom}(\tau) = \mathrm{dom}(\tau')$, et $\tau = \tau'$. $\leq$ est antisymétrique, et est donc un ordre partiel.

Nous allons à présent donner une définition explicite de $\sqcup$ et $\sqcap$, puis prouver qu'il s'agit bien des opérateurs de borne supérieure et de borne inférieure. Il serait agréable de définir ces opérateurs à l'aide de quelques équations intuitives, mais celles-ci seraient alors récursives et ne constitueraient donc pas une définition acceptable. Une autre possibilité serait de les définir à l'aide d'une suite d'approximations finies, mais il faudrait alors prouver que la limite de la suite est un arbre régulier. La solution la plus simple semble être, étant donnés deux arbres réguliers τ_1 et τ_2, de construire $\tau_1 \sqcup \tau_2$ et $\tau_1 \sqcap \tau_2$ en tant qu'automates de termes.

Pour $j \in \{1, 2\}$, soit τ_j l'arbre régulier défini par l'automate $\mathcal{A}_j = (Q_j, q_j^0, \delta_j, l_j)$. On peut supposer – quitte à les ajouter si nécessaire – que $\mathcal{A}_j$ contient un état d'étiquette $\top$ et un autre d'étiquette $\bot$; par abus de langage, nous les noterons respectivement $\top$ et $\bot$. Introduisons quelques notations. $\sqcup^n$ signifiera $\sqcup$ quand n est pair et $\sqcap$ quand n est impair. On adopte une convention similaire en ce qui concerne $\sqcup_b^n$. Soit $Q = \{\sqcup, \sqcap\} \times Q_1 \times Q_2$. Considérons un état $q \in Q$, de la forme $(\sqcup^n, q_1, q_2)$. Son étiquette $l(q)$ est donnée par $l_1(q_1) \sqcup_b^n l_2(q_2)$. Les transitions provenant de l'état q sont définies par :

- Si $l(q) = \rightarrow$, alors $\delta(q, i) = (\sqcup^{n+1+i}, \delta_1(q_1, i), \delta_2(q_2, i))$ pour $i \in \{0, 1\}$. Ici, $\delta_j(q_j, i)$ est indéfini quand $l_j(q_j) \neq \rightarrow$; il faudra alors lire $\perp$ quand $n+1+i$ est pair et $\top$ quand il est impair.
- Sinon, $\delta(q, i)$ est indéfini pour $i \in \{0, 1\}$.

Pour $s \in \{\sqcup, \sqcap\}$, on pose $q_0^s = (s, q_1^0, q_2^0)$. Nous pouvons finalement définir $\tau_0 \, s \, \tau_1$ comme étant l'arbre régulier associé à l'automate $\mathcal{A}^s = (Q, q_0^s, \delta, l)$.

Nous devons maintenant vérifier que $\sqcup$ et $\sqcap$ sont bien les opérateurs de borne supérieure et de borne inférieure pour la relation $\leq$.

D'abord, montrons que $\tau_1 \sqcup \tau_2$ est un majorant de τ_j ($j \in \{1, 2\}$). Soit $p \in \mathrm{dom}(\tau_1 \sqcup \tau_2) \cap \mathrm{dom}(\tau_j)$. Puisque $p \in \mathrm{dom}(\tau_1 \sqcup \tau_2)$, on a $l(q) \notin \{\perp, \top\}$ pour tout état $q = \delta(q_0^\sqcup, r)$ associé à un préfixe strict r de p. D'après la définition de $l(q)$ pour un tel q, et sachant que $p \in \mathrm{dom}(\tau_j)$, on obtient que le $(1+j)^{\text{ème}}$ composant de l'état $\delta(q_0^\sqcup, p)$ est $\hat{\delta}_j(q_j^0, p)$. D'après la définition de la fonction d'étiquetage l, l'étiquette de cet état, qui est $(\tau_1 \sqcup \tau_2)(p)$, est supérieure à $l_j(\hat{\delta}_j(q_j^0, p))$, qui est $\tau_j(p)$. Nous avons donc montré que

$$\forall p \in \mathrm{dom}(\tau_1 \sqcup \tau_2) \cap \mathrm{dom}(\tau_j) \quad \tau_j(p) \leq_b^{\pi(p)} (\tau_1 \sqcup \tau_2)(p)$$

ce qui signifie, par définition, $\tau_j \leq \tau_1 \sqcup \tau_2$.

Réciproquement, soit τ un majorant de τ_1 et de τ_2. Nous avons

$$\forall j \in \{1, 2\} \quad \forall p \in \mathrm{dom}(\tau) \cap \mathrm{dom}(\tau_j) \quad \tau_j(p) \leq_b^{\pi(p)} \tau(p)$$

Considérons un chemin $p \in \mathrm{dom}(\tau) \cap \mathrm{dom}(\tau_1 \sqcup \tau_2)$. Soit $j \in \{1, 2\}$. Si $p \in \mathrm{dom}(\tau_j)$, alors $\tau_j(p) \leq_b^{\pi(p)} \tau(p)$ d'après l'assertion ci-dessus. Sinon, il est facile de vérifier que l'étiquette du $(1+j)^{\text{ème}}$ composant de l'état $\hat{\delta}(q_0^\sqcup, p)$ est le plus petit élément de l'ordre $\leq_b^{\pi(p)}$. Dans les deux cas, l'étiquette de ce $(1+j)^{\text{ème}}$ composant est inférieure (pour $\leq_b^{\pi(p)}$) à $\tau(p)$. Puisque cela est vrai pour tout $j \in \{1, 2\}$, l'étiquette $l(\hat{\delta}(q_0^\sqcup, p) = (\tau_1 \sqcup \tau_2)(p)$, qui est la borne supérieure de ces deux étiquettes, est également inférieure à $\tau(p)$. Puisque cela est vrai pour tout $p \in \mathrm{dom}(\tau) \cap \mathrm{dom}(\tau_1 \sqcup \tau_2)$, nous avons démontré que $\tau_1 \sqcup \tau_2 \leq \tau$.

La conjonction des deux paragraphes précédents montre que $\sqcup$ est l'opérateur de borne supérieure pour l'ordre $\leq$. De même, on vérifie que $\sqcap$ est l'opérateur de borne inférieure. □

Comme dans tout treillis, $\sqcup$ et $\sqcap$ sont associatives et commutatives. De plus, nous pouvons les caractériser à l'aide des quelques équations suivantes. Les quatre premières définissent le comportement de $\top$ et de $\perp$, tandis que les deux dernières indiquent que $\sqcup$ et $\sqcap$ sont distributives sur $\rightarrow$.

Proposition 1.5 *Les équations suivantes sont des identités :*

$$\begin{array}{cc}
\perp \sqcup \tau = \tau & \perp \sqcap \tau = \perp \\
\top \sqcup \tau = \top & \top \sqcap \tau = \tau
\end{array}$$
$$(\tau_1 \rightarrow \tau_2) \sqcup (\tau_1' \rightarrow \tau_2') = (\tau_1 \sqcap \tau_1') \rightarrow (\tau_2 \sqcup \tau_2')$$
$$(\tau_1 \rightarrow \tau_2) \sqcap (\tau_1' \rightarrow \tau_2') = (\tau_1 \sqcup \tau_1') \rightarrow (\tau_2 \sqcap \tau_2')$$

Démonstration. La preuve ne pose pas de difficultés, en revenant à la définition de $\sqcup$ et $\sqcap$ en tant que produits d'automates de termes (voir la preuve de la proposition 1.4). □

Enfin, la proposition suivante, d'intérêt principalement technique, indique que $\sqcup$ et $\sqcap$ ont également un bon comportement vis-à-vis de $\leq_k$.

Proposition 1.6 *Soit $k \in \mathbb{N}^+$. Pour tous types bruts τ_0, τ_1 et τ, l'on a*

$$\tau_0 \leq_k \tau \wedge \tau_1 \leq_k \tau \iff \tau_0 \sqcup \tau_1 \leq_k \tau$$
$$\tau \leq_k \tau_0 \wedge \tau \leq_k \tau_1 \iff \tau \leq_k \tau_0 \sqcap \tau_1$$

Démonstration. Considérons la première ligne (la seconde étant symétrique). En ce qui concerne le sens direct, nous avons déjà prouvé le cas $k = \infty$ dans le cadre de la proposition 1.4 ; pour un k arbitraire, la démonstration est essentiellement identique (il suffit de ne considérer que les chemins de longueur strictement inférieure à k). En ce qui concerne la réciproque, le résultat est immédiat, puisque $\leq$ est plus fine que $\leq_k$. $\square$

Chapitre 2

Types

Ayant introduit les types bruts, qui constituent le modèle logique de notre système, il nous faut à présent définir les types, qui seront les composants élémentaires de nos formules logiques. La structure des types est voisine de celles des types bruts, avec quelques différences. D'abord, une formule doit être finie ; les types sont donc des *termes finis*, à la différence des types bruts, qui sont des arbres potentiellement infinis. Ensuite, l'intérêt d'une formule est de contenir des variables ; nous introduisons donc des *variables de types*. Enfin, troisième et dernière différence, nous autorisons les types à contenir des expressions symboliques formées à l'aide des constructeurs $\sqcup$ et $\sqcap$.

Cette dernière décision est discutable ; c'est pourquoi nous commençons par la justifier (section 2.1). Nous passons ensuite à la partie centrale de ce chapitre, c'est-à-dire la définition précise de la notion de type esquissée ci-dessus (section 2.2). Suivent quelques définitions et développements techniques (sections 2.3 à 2.6).

2.1 A propos des constructeurs $\sqcup$ et $\sqcap$

La décision d'introduire ces constructeurs est motivée par le désir de coder une conjonction de contraintes, de la forme $\bigwedge_{i\in I}(\tau_i \leq \alpha)$, en une seule contrainte $(\sqcup_{i\in I}\tau_i) \leq \alpha$. (Le cas du constructeur $\sqcap$ étant symétrique.) Ainsi, dans un graphe de contraintes (cf. définition 3.5), chaque variable aura exactement une borne inférieure et une borne supérieure. En l'absence de ces constructeurs, les graphes de contraintes devraient garder trace d'un ensemble de bornes pour chaque variable.

Les constructeurs $\sqcup$ et $\sqcap$ introduits dans le langage des types sont accompagnés de règles de réécriture, qui permettent de calculer la *forme normale* d'un type (cf. définition 2.1). Par exemple, le type

$$(\alpha \to \beta) \sqcup (\gamma \to \bot)$$

sera immédiatement réduit en $(\alpha \sqcap \gamma) \to \beta$.

On notera que $\sqcup$ et $\sqcap$ ne seront employés que de façon restreinte. En effet, ils ne servent qu'à coder des conjonctions de contraintes. Ainsi, dans un graphe de contraintes, toute variable α aura une borne inférieure de la forme $\sqcup_{i\in I}\tau_i$, où les τ_i sont dénués d'occurrences des constructeurs $\sqcup$ et $\sqcap$. Ce type peut s'écrire de plusieurs manières, grâce aux règles de réécriture mentionnées ci-dessus. Cependant, toutes ont en commun la propriété suivante : le constructeur $\sqcup$ apparaît uniquement en position *positive*, et le constructeur $\sqcap$ en position *négative*. Le cas de la borne supérieure de α est symétrique.

C'est pendant la phase de *clôture* (cf. définition 7.1) qu'a lieu la combinaison des diverses contraintes provenant de l'analyse du programme. Cette phase calcule donc les bornes de chaque variable et les met en forme normale, comme indiqué ci-dessus. Vient ensuite la phase de *canonisation* (cf. chapitre 11), qui élimine toute occurrence des constructeurs $\sqcup$ et $\sqcap$.

La combinaison des bornes et leur mise en forme normale permet d'améliorer le partage et de représenter les contraintes sous une forme plus compacte. Par exemple, d'après les règles de calcul associées aux constructeurs $\sqcup$ et $\sqcap$, le terme $(\alpha_0 \to \alpha_1) \sqcup (\beta_0 \to \beta_1)$ se réduit en $(\alpha_0 \sqcap \beta_0) \to (\alpha_1 \sqcup \beta_1)$, où le constructeur $\to$ est partagé. Mieux, ce calcul peut provoquer la disparition de certains termes ; par exemple, $(\alpha_0 \to \alpha_1) \sqcup \top$ se réécrit en $\top$.

Si nous avions choisi de garder trace de chacune des bornes, individuellement, au sein d'un ensemble, cette mise en forme normale n'aurait pas lieu dès la clôture ; elle ne serait effectuée que lors de la canonisation. Par conséquent, l'introduction des constructeurs $\sqcup$ et $\sqcap$, ainsi que des règles de normalisation associées, revient en fait à réaliser une partie du travail de canonisation, de façon incrémentale, pendant la phase de clôture. Elle permet ainsi, d'un point de vue pratique, de manipuler des graphes plus compacts. Quel que soit le choix effectué, une fois la canonisation terminée, chaque variable admet une unique borne inférieure et une unique borne supérieure, toutes deux dénuées d'occurrences de $\sqcup$ ou $\sqcap$; seule la « partie avant » du système est donc affectée par cette décision.

Exemple. Imaginons qu'une variable α ait pour bornes inférieures $\beta_1 \to \beta_2$ et $\gamma_1 \to \gamma_2$. Nous combinerons donc ces deux types en une borne unique, $(\beta_1 \sqcap \gamma_1) \to (\beta_2 \sqcup \gamma_2)$. Supposons à présent que, au cours de l'analyse du programme, une troisième borne inférieure apparaisse pour α, à savoir $\beta_1 \to \gamma_2$. Nous combinons donc cette nouvelle borne avec l'ancienne, en réalisant le calcul

$$((\beta_1 \sqcap \gamma_1) \to (\beta_2 \sqcup \gamma_2)) \sqcup (\beta_1 \to \gamma_2)$$

Or le résultat de ce calcul n'est autre que

$$(\beta_1 \sqcap \gamma_1) \to (\beta_2 \sqcup \gamma_2)$$

ce qui indique que la nouvelle borne n'apporte aucune information supplémentaire.

Si nous avions fait le choix de représenter l'ancienne borne inférieure de la variable α par l'ensemble $\{\beta_1 \to \beta_2, \gamma_1 \to \gamma_2\}$, alors le test d'appartenance de la nouvelle borne $\beta_1 \to \gamma_2$ à cet ensemble aurait échoué. Celle-ci aurait donc été ajoutée à l'ensemble des bornes inférieures de α. On constate ainsi que les ensembles de bornes pourraient croître de façon redondante, entraînant une perte d'efficacité en espace et en temps.

D'un point de vue théorique, la présentation de la notion de la clôture est légèrement compliquée par cette décision, comme on pouvait s'y attendre, tandis que la description de l'algorithme de canonisation est simplifiée. L'influence de cette alternative sur la théorie est donc relativement limitée ; notre choix est principalement motivé par ses avantages pratiques.

Trifonov et Smith [49] n'autorisent pas les constructeurs $\sqcup$ et $\sqcap$ à apparaître dans les types. Comme nous l'avons expliqué plus haut, leur système est néanmoins de puissance équivalente. Aiken, Wimmers et Lakshman [6, 7] proposent un système dans lequel les constructeurs $\cup$ et $\cap$ sont utilisés de façon moins restreinte. (Ces constructeurs représentent l'union et l'intersection ensembliste, non les opérations du treillis des types, mais se comportent, en première approximation, de façon comparable.) Une expression $\tau_1 \cup \tau_2$ est autorisée à apparaître en position négative, à condition qu'il s'agisse d'une union disjointe (c'est-à-dire $\tau_1 \cap \tau_2 = \varnothing$ pour toute assignation des variables libres). Une expression $\tau_1 \cap \tau_2$ est autorisée à apparaître en position positive, à condition que τ_2 n'ait aucune variable libre et soit *clos supérieurement*. Ces possibilités ne sont pas disponibles dans notre système ; cependant, notons qu'une partie de leur puissance peut être recouvrée grâce aux types variantes, que nous introduirons au chapitre 14. Par exemple, dans [6], lorsqu'on rencontre une union en position négative $\tau_1 \cup \tau_2$, on a typiquement $\tau_1 = a(\tau_1')$ et $\tau_2 = b(\tau_2')$, où a et b sont des constructeurs de données distincts, puisque l'union doit être disjointe. Alors, le type $\tau_1 \cup \tau_2$ correspond, dans notre système, au type $\texttt{[a: } \tau_1' \texttt{ | b: } \tau_2' \texttt{]}$, en utilisant la notation de la section 14.4. Lorsqu'on rencontre une intersection en position positive $\tau_1 \cap \tau_2$, on a typiquement $\tau_2 = \neg a(1)$, où a est un constructeur de données et 1 est l'équivalent de notre type $\top$. Alors, le type $\tau_1 \cap \tau_2$ peut s'écrire, dans notre système, $\texttt{[a: Abs; } \rho \texttt{]}$, à condition d'ajouter la contrainte $\tau_1 \leq \texttt{[a: Pre } \top \texttt{; } \rho \texttt{]}$, qui permet de « soustraire » à τ_1 le champ a et de représenter tous les autres champs par la variable de rangée ρ. (Les types variantes avec variables de rangée sont présentés à la section 14.5.2.) Notre système est donc relativement expressif. Cependant, certaines possibilités offertes dans [7], en particulier l'utilisation de l'union et des *types conditionnels* pour obtenir un typage très fin des filtrages, ne peuvent pour l'instant pas être exprimées dans notre système.

2.2 Types

Passons à présent à la définition des types. Celle-ci se fait en deux étapes. Nous introduisons d'abord des termes appelés *prétypes*. Ensuite, pour tenir compte des propriétés des constructeurs $\sqcup$ et $\sqcap$, nous quotientons l'ensemble des prétypes par une certaine relation de congruence, ce qui donne naissance aux *types*.

Nous sommes amenés à introduire diverses sortes de (pré)types, qui diffèrent seulement par les positions dans lesquelles les constructeurs $\sqcup$ et $\sqcap$ sont autorisés à apparaître. Les types *simples* ne sont pas autorisés à contenir ces constructeurs ; une fois passée la phase de canonisation, nous n'utiliserons plus que de tels types. Les bornes inférieure et supérieure d'une variable, dans un graphe de contraintes, seront respectivement un *pos-type* et un *neg-type*. Enfin, les bi-prétypes, qui n'imposent aucune restriction sur l'utilisation des constructeurs $\sqcup$ et $\sqcap$, seront utilisés uniquement dans la section 2.6. Nous ne leur appliquerons pas l'opération de passage au quotient mentionnée ci-dessus.

Définition 2.1 *Soit $\mathcal{V}$ un ensemble dénombrable de* variables de types*, notées α, β, etc. L'ensemble des* prétypes simples*, noté $p\mathcal{T}$, est constitué des termes définis par la grammaire suivante :*

$$\tau ::= \alpha \mid \bot \mid \top \mid \tau \rightarrow \tau$$

L'ensemble des pos-prétypes*, noté $p\mathcal{T}^+$, est défini par*

$$\tau^+ ::= \alpha \mid \bot \mid \top \mid \tau^- \rightarrow \tau^+ \mid \sqcup\{\tau^+, \dots, \tau^+\}$$

L'ensemble des neg-prétypes*, noté $p\mathcal{T}^-$, est défini par*

$$\tau^- ::= \alpha \mid \bot \mid \top \mid \tau^+ \rightarrow \tau^- \mid \sqcap\{\tau^-, \dots, \tau^-\}$$

Enfin, l'ensemble des bi-prétypes*, noté $p\mathcal{T}^\pm$, est défini par*

$$\tau^\pm ::= \alpha \mid \bot \mid \top \mid \tau^\pm \rightarrow \tau^\pm \mid \sqcup\{\tau^\pm, \dots, \tau^\pm\} \mid \sqcap\{\tau^\pm, \dots, \tau^\pm\}$$

Nous parlerons de prétypes *tout court lorsque la distinction sera sans importance ou claire d'après le contexte.*

Plutôt que de définir $\sqcup$ et $\sqcap$ comme des constructeurs binaires, nous les rendons unaires, leur argument étant un ensemble de termes. Cela facilite le traitement des problèmes d'associativité et de commutativité. La notation $\tau \sqcup \tau'$ représentera $\sqcup\{\tau, \tau'\}$.

Dans le langage des prétypes, il existe de nombreux termes qui, intuitivement, représentent le même type. Par exemple, le terme

$$\alpha \sqcup (\sqcup\{\alpha, \bot \rightarrow \beta, \gamma \rightarrow \beta\})$$

se simplifie (toujours informellement) en $\alpha \sqcup (\bot \rightarrow \beta)$. Ces termes sont deux prétypes différents ; cependant nous souhaiterions les identifier. Pour cela, nous allons définir une relation de congruence sur les prétypes, et l'utiliser pour définir les types comme des classes de congruence.

Définition 2.2 *Soit $\equiv$ la congruence engendrée par les identités suivantes :*

$$\begin{aligned}
\sqcup\varnothing &\equiv \bot \\
\sqcup\{\tau\} &\equiv \tau \\
\sqcup(\{\sqcup S\} \cup S') &\equiv \sqcup(S \cup S') \\
\sqcup(\{\bot\} \cup S) &\equiv \sqcup S \\
\sqcup(\{\top\} \cup S) &\equiv \top \\
\sqcup(\{\tau_0 \to \tau_1, \tau'_0 \to \tau'_1\} \cup S) &\equiv \sqcup(\{(\tau_0 \sqcap \tau'_0) \to (\tau_1 \sqcup \tau'_1)\} \cup S)
\end{aligned}$$

(plus 6 identités symétriques concernant $\sqcap$).

Les deux premières identités traitent les cas particuliers où l'opérateur possède zéro ou un argument. La troisième représente l'associativité. Enfin, les trois dernières sont des règles de calcul et proviennent de la proposition 1.5.

Définition 2.3 *L'ensemble des* types simples*, noté $\mathcal{T}$, est le quotient $p\mathcal{T}/\equiv$. Les ensembles des* pos-types *et des* neg-types*, notés respectivement $\mathcal{T}^+$ et $\mathcal{T}^-$, sont définis de façon similaire.*

Nous parlerons de types *tout court lorsque la distinction sera sans importance ou claire d'après le contexte.*

Ainsi, de façon formelle, un type est une classe de prétypes ; l'on pourra le dénoter par n'importe lequel de ses éléments. Cependant, pour pouvoir raisonner sur les types, nous allons isoler, dans chaque classe, un élément particulier appelé *forme normale.*

Proposition 2.1 *On définit un système de réécriture en orientant chacune des équations de la définition 2.2 de la gauche vers la droite. Alors, ce système est confluent et nœthérien. En d'autres termes, tout type admet une forme normale.*

Démonstration. Pour prouver que ce système de réécriture est localement confluent, nous utilisons l'outil automatique RRL [32]. En plus de notre syntaxe de termes, nous devons définir quelques notions de théorie des ensembles. Voici notre fichier de définitions :

```
;; -------------------------------------------------------------------------
;; Ce fichier RRL permet de prouver que les types admettent une forme
;; normale vis-à-vis des règles de réécriture.

;; -------------------------------------------------------------------------
;; Un petit morceau de théorie des ensembles.
;; Il faut également déclarer que cup (l'opérateur d'union ensembliste) est
;; associatif et commutatif, à l'aide de l'interface interactive de RRL.

[empty : set]
[s : term -> set]
[cup : set, set -> set]

cup(X, empty) := X

;; -------------------------------------------------------------------------
;; Voici nos règles de réécriture.

[glb : set -> term]
[lub : set -> term]
[arr : term, term -> term]
[bot : term]
[top : term]

lub(empty) := bot
```

```
lub(s(X)) := X
lub(cup(s(lub(X)),Y)) := lub(cup(X,Y))
lub(cup(s(bot), X)) := lub(X)
lub(cup(s(top), X)) := top
lub(cup(cup(s(arr(X0, X1)), s(arr(Y0, Y1))), Z)) :=
  lub(cup(s(arr(glb(cup(s(X0),s(Y0))),lub(cup(s(X1),s(Y1))))),Z))

glb(empty) := top
glb(s(X)) := X
glb(cup(s(glb(X)),Y)) := glb(cup(X, Y))
glb(cup(s(top), X)) := glb(X)
glb(cup(s(bot), X)) := bot
glb(cup(cup(s(arr(X0, X1)), s(arr(Y0, Y1))), Z)) :=
  glb(cup(s(arr(lub(cup(s(X0),s(Y0))),glb(cup(s(X1),s(Y1))))),Z))
```

Confronté à ceci, RRL demande à l'utilisateur d'orienter manuellement 4 équations, puis annonce un succès, ce qui prouve que le système est localement confluent. Pour prouver qu'il n'existe aucune suite de réécritures infinie, définissons la taille d'un prétype :

$$\begin{aligned}
\text{taille}(\bot) &= 1 \\
\text{taille}(\top) &= 1 \\
\text{taille}(\alpha) &= 1 \\
\text{taille}(\tau_0 \to \tau_1) &= 5 + \text{taille}(\tau_0) + \text{taille}(\tau_1) \\
\text{taille}(\sqcup S) &= 2 + \Sigma_{\tau \in S}\, \text{taille}(\tau) \\
\text{taille}(\sqcap S) &= 2 + \Sigma_{\tau \in S}\, \text{taille}(\tau)
\end{aligned}$$

On vérifie alors aisément que chacune des règles de réécriture diminue strictement la taille du type. Par conséquent, le système est nœthérien. Etant localement confluent, il est donc confluent. (Merci à Cesar Muñoz d'avoir suggéré – et facilité – l'utilisation de RRL.) □

Les formes normales peuvent être caractérisées de la façon suivante.

Proposition 2.2 *Un prétype τ est une forme normale ssi chaque occurrence dans τ des constructeurs $\sqcup$ et $\sqcap$ vérifie les conditions suivantes :*

- *le nombre d'arguments du constructeur est $n \geq 2$;*
- *au moins $n-1$ arguments sont des variables de types ;*
- *si l'un des arguments n'est pas une variable de types, alors il s'agit d'un terme dont le constructeur de tête n'est pas parmi $\bot$, $\top$, $\sqcup$ et $\sqcap$.*

Démonstration. Il est facile de vérifier que si l'une des conditions ci-dessus est violée, alors l'une des règles de réécriture s'applique ; et réciproquement, que si toutes les conditions sont remplies, alors aucune des règles ne s'applique. □

Dorénavant, lorsque nous raisonnerons sur la forme d'un type (c'est-à-dire sur sa structure en tant que terme), c'est sa forme normale qui sera implicitement considérée. Ainsi, les fonctions définies sur les prétypes par analyse de leur structure s'étendent implicitement aux types.

2.3 Définitions auxiliaires

La notion de variables libres dans un type est définie de façon classique. Cependant, nous devrons parfois diviser les variables libres d'un terme en deux groupes, selon qu'elles apparaissent en position positive ou négative.

Définition 2.4 *L'ensemble des* variables libres positives *(resp.* négatives*) d'un prétype τ, noté* $\mathrm{fv}^+(\tau)$ *(resp.* $\mathrm{fv}^-(\tau)$*), est défini par*

$$\begin{array}{rlrl}
\mathrm{fv}^+(\alpha) & = \{\alpha\} & \mathrm{fv}^-(\alpha) & = \varnothing \\
\mathrm{fv}^+(\bot) & = \varnothing & \mathrm{fv}^-(\bot) & = \varnothing \\
\mathrm{fv}^+(\top) & = \varnothing & \mathrm{fv}^-(\top) & = \varnothing \\
\mathrm{fv}^+(\tau_0 \to \tau_1) & = \mathrm{fv}^-(\tau_0) \cup \mathrm{fv}^+(\tau_1) & \mathrm{fv}^-(\tau_0 \to \tau_1) & = \mathrm{fv}^+(\tau_0) \cup \mathrm{fv}^-(\tau_1) \\
\mathrm{fv}^+(\sqcup S) & = \cup_{\tau \in S} \mathrm{fv}^+(\tau) & \mathrm{fv}^-(\sqcup S) & = \cup_{\tau \in S} \mathrm{fv}^-(\tau) \\
\mathrm{fv}^+(\sqcap S) & = \cup_{\tau \in S} \mathrm{fv}^+(\tau) & \mathrm{fv}^-(\sqcap S) & = \cup_{\tau \in S} \mathrm{fv}^-(\tau)
\end{array}$$

L'ensemble des variables libres *de τ, noté* $\mathrm{fv}(\tau)$*, est défini par*

$$\mathrm{fv}(\tau) = \mathrm{fv}^+(\tau) \cup \mathrm{fv}^-(\tau)$$

La hauteur d'un terme est définie de façon classique. On notera que les constructeurs $\sqcup$ et $\sqcap$ n'ont pas de hauteur intrinsèque, puisqu'ils représentent en fait des opérations qui conservent la hauteur des termes bruts.

Définition 2.5 *La* hauteur *d'un prétype τ, notée* $\mathrm{h}(\tau)$*, est définie par*

$$\begin{aligned}
\mathrm{h}(\alpha) &= 0 \\
\mathrm{h}(\bot) &= 0 \\
\mathrm{h}(\top) &= 0 \\
\mathrm{h}(\tau_0 \to \tau_1) &= 1 + \max\{\mathrm{h}(\tau_0), \mathrm{h}(\tau_1)\} \\
\mathrm{h}(\sqcup S) &= \max\{\mathrm{h}(\tau)\,;\, \tau \in S\} \\
\mathrm{h}(\sqcap S) &= \max\{\mathrm{h}(\tau)\,;\, \tau \in S\}
\end{aligned}$$

La définition suivante sera utilisée intensivement à la section 2.6, ainsi que dans certains énoncés concernant l'implication de contraintes.

Définition 2.6 *La* profondeur de la plus proche variable *dans un prétype τ, notée* $\mathrm{pppv}(\tau)$*, est définie par*

$$\begin{aligned}
\mathrm{pppv}(\alpha) &= 0 \\
\mathrm{pppv}(\bot) &= \infty \\
\mathrm{pppv}(\top) &= \infty \\
\mathrm{pppv}(\tau_0 \to \tau_1) &= 1 + \min\{\mathrm{pppv}(\tau_0), \mathrm{pppv}(\tau_1)\} \\
\mathrm{pppv}(\sqcup S) &= \min\{\mathrm{pppv}(\tau)\,;\, \tau \in S\} \\
\mathrm{pppv}(\sqcap S) &= \min\{\mathrm{pppv}(\tau)\,;\, \tau \in S\}
\end{aligned}$$

La définition suivante rappelle la notion de constructeur de tête. Si le constructeur de tête d'un type τ n'est ni $\sqcup$ ou $\sqcap$, ni une variable, alors toute substitution brute associera à τ un terme brut ayant le même constructeur de tête ; τ sera alors qualifié de *construit*.

Définition 2.7 *Le* constructeur de tête *d'un prétype τ est $\bot$, $\to$ ou $\top$, lorsque τ est de la forme $\bot$, $\tau_0 \to \tau_1$, ou $\top$, respectivement ; sinon, il est indéfini. On le note* $\mathrm{tête}(\tau)$*. τ est dit* construit *ssi* $\mathrm{tête}(\tau)$ *est défini.*

2.4 Substitutions brutes et renommages

Nous définissons ici les *substitutions brutes*. Elles établissent un lien entre types et types bruts, et ainsi donnent une signification à toutes les structures basées sur les types : graphes de contraintes, schémas de types, etc.

Définition 2.8 *Une* substitution brute *est une fonction (partielle) des variables de types vers les arbres bruts. Elle est dite* régulière *si son image ne contient que des arbres réguliers.*

Dorénavant, nous ne travaillerons qu'avec des substitutions totales et régulières (c'est-à-dire de domaine $\mathcal{V}$ et d'image incluse dans $\mathbb{T}$), sauf mention explicite du contraire.

Nous avons défini les substitutions brutes sur les variables. Etendons-les maintenant aux prétypes, puis aux types.

Définition 2.9 *Soit ρ une substitution brute de domaine $V \subset \mathcal{V}$. On l'étend à l'ensemble des prétypes τ tels que* $\mathrm{fv}(\tau) \subset V$ *en posant*

$$\begin{aligned}
\rho(\alpha) &= \rho(\alpha) \\
\rho(\bot) &= \bot \\
\rho(\top) &= \top \\
\rho(\tau_0 \to \tau_1) &= \rho(\tau_0) \to \rho(\tau_1) \\
\rho(\sqcup S) &= \sqcup \rho(S) \\
\rho(\sqcap S) &= \sqcap \rho(S)
\end{aligned}$$

On étend ensuite ρ aux types en définissant l'image d'un type comme l'image de sa forme normale.

Proposition 2.3 *Soient τ et τ' deux prétypes de la même classe, c'est-à-dire tels que $\tau \equiv \tau'$. Soit ρ une substitution brute. Alors $\rho(\tau)$ et $\rho(\tau')$, s'ils sont définis, sont égaux. Il en découle que les identités données dans la définition 2.9 sont également valides sur les types.*

Démonstration. Il suffit de vérifier que les équations qui définissent la congruence entre prétypes (cf. définition 2.2) sont des identités sur les types bruts. □

Définition 2.10 *Un* renommage *est une bijection entre deux sous-ensembles de $\mathcal{V}$.*

Les renommages s'étendent aisément aux prétypes et aux types.

2.5 Inclusion entre types

On pourrait considérer que les constructeurs $\sqcup$ et $\sqcap$ servent à coder un ensemble de types au sein d'un seul type. En effet, tout pos-type (resp. neg-type) peut s'écrire sous la forme $\sqcup S$ (resp. $\sqcap S$), où S est un ensemble de types simples. (Il suffit, pour vérifier cela, de repousser les constructeurs $\sqcup$ et $\sqcap$ vers le haut du terme.)

L'opération $\cup$ sur ces ensembles de termes correspond à l'opération $\sqcup$ sur les pos-types et à $\sqcap$ sur les neg-types. Quel est l'analogue de l'inclusion entre ensembles de termes ? Nous allons le définir ici. Il nous sera utile, en particulier, dans la définition de la clôture d'un graphe de contraintes.

Définition 2.11 *Un pos-type τ* contient *un pos-type τ' ssi $\tau \sqcup \tau' = \tau$. Symétriquement, un neg-type τ contient un neg-type τ' ssi $\tau \sqcap \tau' = \tau$. Dans les deux cas, on écrira $\tau' \lessdot \tau$.*

Rappelons que les types sont des classes de congruence, donc la comparaison entre deux types se fait en calculant d'abord leurs formes normales, puis en comparant celles-ci. Par exemple, $\alpha \sqcup (\beta \to (\gamma \sqcup \delta))$ contient $\bot$, α, et $\beta \to \delta$.

Proposition 2.4 *$\lessdot$ est une relation d'ordre sur $\mathcal{T}^+$ (respectivement, $\mathcal{T}^-$).*

Démonstration. Montrons que $\lessdot$ est un ordre sur $\mathcal{T}^+$. Elle est réflexive, car $\tau \sqcup \tau = \tau$. Elle est antisymétrique, car si $\tau \lessdot \tau'$ et $\tau' \lessdot \tau$, alors $\tau \sqcup \tau' = \tau = \tau'$. Elle est transitive, car si $\tau \lessdot \tau'$ et $\tau' \lessdot \tau''$, alors

$$\begin{aligned} \tau \sqcup \tau'' &= \tau \sqcup (\tau' \sqcup \tau'') \\ &= (\tau \sqcup \tau') \sqcup \tau'' \\ &= \tau' \sqcup \tau'' \\ &= \tau'' \end{aligned}$$

donc $\tau \lessdot \tau''$. Le résultat concernant $\mathcal{T}^-$ est symétrique. □

2.6 Systèmes d'équations de types

Le but de cette section est d'étudier certains systèmes d'équations entre types, et en particulier, d'établir un isomorphisme entre les types bruts et les systèmes d'équations *contractifs*. Il s'agit là d'un résultat classique [14]. Cependant, il est légèrement généralisé ici, du fait de la présence des constructeurs $\sqcup$ et $\sqcap$ dans les équations. C'est pourquoi nous en détaillons la démonstration. Celle-ci reste cependant fort classique.

Nous introduisons d'abord une structure d'espace métrique sur les arbres bruts. Puis, nous définissons les systèmes d'équations contractifs et établissons l'isomorphisme mentionné ci-dessus.

2.6.1 Propriétés métriques des arbres

Définition 2.12 *On définit une distance d sur l'ensemble $\mathbb{T}_\infty$ des arbres bruts, en posant $d(\tau, \tau') = 2^{-l}$, où $l = \min\{|\,p\,| \,;\, \tau(p) \neq \tau'(p)\}$, avec la convention que $2^{-\infty} = 0$. De plus, étant un donné un ensemble fini I, on définit une distance (également notée d) sur $\mathbb{T}_\infty^I$ par*

$$d((\tau_i)_{i\in I}, (\tau'_i)_{i\in I}) = \max\{d(\tau_i, \tau'_i) \,;\, i \in I\}$$

Soit V un sous-ensemble fini de $\mathcal{V}$. L'ensemble des substitutions brutes (non nécessairement régulières) de domaine V est $\mathbb{T}_\infty^V$. Il est donc également muni d'une distance par cette définition.

Munis de cette distance, $\mathbb{T}_\infty$ et $\mathbb{T}_\infty^I$ sont des espaces métriques complets. L'ensemble $\mathbb{T}_\infty$ des arbres bruts est la clôture (au sens topologique) de l'ensemble des arbres bruts finis.

Démonstration. On se référera à [10, 14]. □

Définition 2.13 *Soit $l \in \mathbb{N}^+ \cup \{\infty\}$. Deux arbres bruts τ et τ'* coïncident à l'ordre l *ssi*

$$\forall p \in \mathrm{dom}(\tau) \cap \mathrm{dom}(\tau') \quad |\,p\,| < l \Rightarrow \tau(p) = \tau'(p)$$

Proposition 2.5 *Deux arbres τ et τ' coïncident à l'ordre l ssi $d(\tau, \tau') \leq 2^{-l}$.*

Démonstration. Immédiat. □

Proposition 2.6 *Les fonctions $\sqcup$ et $\sqcap$, initialement définies comme éléments de $\mathbb{T}^2 \to \mathbb{T}$, s'étendent par continuité à $\mathbb{T}_\infty^2 \to \mathbb{T}_\infty$. Elles sont alors 1-contractantes.*

Toute assertion de sous-typage sur les types bruts, mettant en jeu les opérateurs $\sqcup$, $\sqcap$, et $\to$, est également vérifiée sur les arbres bruts. En particulier, $\mathbb{T}_\infty$ est un treillis.

Démonstration. Rappelons qu'une fonction $f : E \mapsto F$ (où E et F sont des espaces métriques) est dite 1-contractante ssi

$$\forall x, x' \in E \quad d_F(f(x), f(x')) \leq d_E(x, x')$$

Montrons d'abord que $\sqcup$ et $\sqcap$ sont uniformément continues sur $\mathbb{T}^2$. Plus précisément, nous allons montrer que pour tous arbres réguliers τ_0, τ'_0, τ_1 et τ'_1,

$$d(\tau_0 \sqcup \tau_1, \tau'_0 \sqcup \tau'_1) \leq d((\tau_0, \tau_1), (\tau'_0, \tau'_1))$$
$$d(\tau_0 \sqcap \tau_1, \tau'_0 \sqcap \tau'_1) \leq d((\tau_0, \tau_1), (\tau'_0, \tau'_1))$$

Nous avons, par définition de la distance,

$$d((\tau_0, \tau_1), (\tau'_0, \tau'_1)) = \max\{d(\tau_0, \tau'_0), d(\tau_1, \tau'_1)\}$$

Donc, notre but est équivalent à l'assertion suivante : « pour tout $l \in \mathbb{N}^+ \cup \{\infty\}$, si τ_0 coïncide avec τ'_0 à l'ordre l et τ_1 coïncide avec τ'_1 à l'ordre l, alors $\tau_0 \sqcup \tau_1$ coïncide avec $\tau'_0 \sqcup \tau'_1$ à l'ordre l, et $\tau_0 \sqcap \tau_1$ coïncide avec $\tau'_0 \sqcap \tau'_1$ à l'ordre l. »

Or, cet énoncé se vérifie aisément en considérant la définition de $\sqcup$ et $\sqcap$ en tant que produits d'automates. Les transitions de l'automate produit dépendent directement de celles des automates de départ ; donc, quand on lui soumet un chemin p, l'automate produit fournit un label qui ne peut dépendre que des labels fournis par les automates de départ lorsqu'on leur soumet des préfixes de p.

Ainsi, $\sqcup$ et $\sqcap$ sont 1-contractantes sur $\mathbb{T}^2$. Donc, elles peuvent s'étendre par continuité à sa clôture, qui est $\mathbb{T}^2_\infty$. Les fonctions étendues sont également 1-contractantes.

La fonction $\rightarrow$, fonction binaire des arbres bruts dans les arbres bruts, est $\frac{1}{2}$-contractante. On note que $\leq$, considérée comme une fonction binaire des arbres bruts dans les booléens, n'est pas continue. Cependant, on vérifie aisément qu'elle est inférieurement continue.

Cela implique que toute assertion de sous-typage sur les types bruts, mettant en jeu les opérateurs $\sqcup$, $\sqcap$, et $\rightarrow$, passe aux arbres bruts. □

2.6.2 Systèmes d'équations contractifs

Au cours de cette étude des systèmes contractifs, nous considérerons souvent des substitutions brutes irrégulières. Jusqu'à présent, celles-ci n'étaient définies que sur les variables. (En effet, l'extension réalisée par la définition 2.9 ne concerne que les substitutions brutes régulières.) Nous pouvons à présent les étendre aux prétypes, grâce à notre extension des opérations $\sqcup$ et $\sqcap$ (cf. proposition 2.6).

Par ailleurs, il n'y a ici aucune raison de restreindre l'usage des constructeurs $\sqcup$ et $\sqcap$, aussi travaillerons-nous avec des bi-prétypes.

Lemme 2.7 *Soit τ un bi-prétype. Soient ρ, ρ' deux substitutions brutes (éventuellement irrégulières) dont le domaine contient* $\mathrm{fv}(\tau)$. *Alors*

$$d(\rho(\tau), \rho'(\tau)) \leq 2^{-\mathrm{pppv}(\tau)}.\, d(\rho, \rho')$$

Démonstration. Par induction sur la structure de τ.

- τ est de la forme α. Alors le but devient $d(\rho(\alpha), \rho'(\alpha)) \leq d(\rho, \rho')$, qui est une conséquence directe de la définition 2.12.
- τ est égal à $\bot$ ou $\top$. Alors $\rho(\tau) = \rho'(\tau)$, donc le membre gauche du but est 0 et le but est vérifié.
- τ est de la forme $\tau_0 \rightarrow \tau_1$. Alors

$$\begin{aligned}
d(\rho(\tau), \rho'(\tau)) &= d(\rho(\tau_0 \rightarrow \tau_1), \rho'(\tau_0 \rightarrow \tau_1)) \\
&= d(\rho(\tau_0) \rightarrow \rho(\tau_1), \rho'(\tau_0) \rightarrow \rho'(\tau_1)) \\
&= \tfrac{1}{2} \max\{d(\rho(\tau_0), \rho'(\tau_0)), d(\rho(\tau_1), \rho'(\tau_1))\} \\
&\leq \tfrac{1}{2} \max\{2^{-\mathrm{pppv}(\tau_0)}, 2^{-\mathrm{pppv}(\tau_1)}\}.\, d(\rho, \rho') \\
&= 2^{-\mathrm{pppv}(\tau_0 \rightarrow \tau_1)}.\, d(\rho, \rho') \\
&= 2^{-\mathrm{pppv}(\tau)}.\, d(\rho, \rho')
\end{aligned}$$

L'hypothèse d'induction est utilisée pour passer de la troisième à la quatrième ligne.

- τ est de la forme $\sqcup S$. D'après la proposition 2.6, $\sqcup$ est une fonction binaire 1-contractante sur les arbres bruts, et elle est associative. Cette associativité permet d'étendre aisément la propriété de 1-contraction à la fonction n-aire $\sqcup$, pour tout $n \in \mathbb{N}^+$. Cette propriété est utilisée, avec $n = |S|$, pour passer de la première à la deuxième ligne dans ce qui suit :

$$\begin{aligned} d(\rho(\tau), \rho'(\tau)) &= d(\sqcup\rho(S), \sqcup\rho'(S)) \\ &\leq \max\{d(\rho(\tau), \rho'(\tau))\,;\, \tau \in S\} \\ &\leq \max\{2^{-\mathrm{pppv}(\tau)}\,;\, \tau \in S\}.\, d(\rho, \rho') \\ &= 2^{-\mathrm{pppv}(\tau)}.\, d(\rho, \rho') \end{aligned}$$

L'hypothèse d'induction est utilisée pour passer de la deuxième à la troisième ligne.

- τ est de la forme $\sqcap S$. Ce cas est symétrique au précédent. □

Définition 2.14 *Soit V un sous-ensemble fini de $\mathcal{V}$. Un* système contractif *de domaine V est une fonction S de V dans $p\mathcal{T}^{\pm}$, telle que pour tout $\alpha \in V$, $S(\alpha)$ est un bi-prétype construit et à variables dans V. Une substitution brute (éventuellement irrégulière) ρ est* solution *de S ssi* $\mathrm{dom}(\rho) = V$ *et*

$$\forall \alpha \in V \quad \rho(\alpha) = \rho(S(\alpha))$$

Proposition 2.8 *Soit S un système contractif. Alors toute solution de S est régulière.*

Démonstration. Soient ρ une solution de S et $\alpha \in V$, où $V = \mathrm{dom}(S)$. On vérifie aisément que tout sous-arbre de $\rho(\alpha)$ est l'image par ρ d'un sous-terme de $S(\beta)$, pour un certain $\beta \in V$. Par conséquent, $\rho(\alpha)$ admet un nombre fini de sous-arbres, et ρ est une substitution brute régulière. □

Théorème 2.1 *Soit S un système contractif de domaine V. Alors S admet une unique solution.*

Démonstration. Définissons S_∞ par

$$\begin{aligned} \mathbb{T}_\infty^V &\to \mathbb{T}_\infty^V \\ (\tau_\alpha)_{\alpha \in V} &\mapsto (S(\alpha)[\alpha \leftarrow \tau_\alpha]_{\alpha \in V})_{\alpha \in V} \end{aligned}$$

Nous allons à présent montrer que S_∞ est $\frac{1}{2}$-contractante. Par définition de S_∞,

$$d(S_\infty((\tau_\alpha)_{\alpha \in V}), S_\infty((\tau'_\alpha)_{\alpha \in V})$$

est égal à

$$\max\{d(S(\alpha)[\alpha \leftarrow \tau_\alpha]_{\alpha \in V}, S(\alpha)[\alpha \leftarrow \tau'_\alpha]_{\alpha \in V})\,;\, \alpha \in V\}$$

Nous pouvons à présent appliquer le lemme 2.7 au terme $S(\alpha)$ et aux substitutions brutes (éventuellement irrégulières) $\rho = [\alpha \leftarrow \tau_\alpha]_{\alpha \in V}$ et $\rho' = [\alpha \leftarrow \tau'_\alpha]_{\alpha \in V}$. Il en ressort que l'expression ci-dessus est inférieure ou égale à

$$\max\{2^{-\mathrm{pppv}(S(\alpha))}\,;\, \alpha \in V\}.\, d(\rho, \rho')$$

Or, d'après la définition 2.14, chaque $S(\alpha)$ est un bi-prétype construit. Par conséquent, $\mathrm{pppv}(S(\alpha))$ vaut au moins 1, et l'expression en question est inférieure ou égale à

$$\tfrac{1}{2} d(\rho, \rho')$$

Enfin, par définition de ρ et de ρ', on a

$$\begin{aligned} d(\rho, \rho') &= d([\alpha \leftarrow \tau_\alpha]_{\alpha \in V}, [\alpha \leftarrow \tau'_\alpha]_{\alpha \in V}) \\ &= \max\{d(\tau_\alpha, \tau'_\alpha)\,;\, \alpha \in V\} \\ &= d((\tau_\alpha)_{\alpha \in V}, (\tau'_\alpha)_{\alpha \in V}) \end{aligned}$$

Nous avons donc démontré

$$d(S_\infty((\tau_\alpha)_{\alpha\in V}), S_\infty((\tau'_\alpha)_{\alpha\in V}) \leq \tfrac{1}{2} d((\tau_\alpha)_{\alpha\in V}, (\tau'_\alpha)_{\alpha\in V})$$

ce qui établit que S_∞ est $\frac{1}{2}$-contractante.

S_∞ est une fonction contractante d'un espace métrique complet vers lui-même. Par conséquent, elle admet un unique point fixe. Pour conclure, il ne reste plus qu'à remarquer que

$$\begin{array}{ll} & \rho \text{ est solution du système } S \\ \iff & \forall \alpha \in V \quad \rho(\alpha) = \rho(S(\alpha)) \\ \iff & \forall \alpha \in V \quad \rho(\alpha) = S(\alpha)[\alpha \leftarrow \rho(\alpha)]_{\alpha\in V} \\ \iff & (\rho(\alpha))_{\alpha\in V} = S_\infty((\rho(\alpha))_{\alpha\in V}) \\ \iff & (\rho(\alpha))_{\alpha\in V} \text{ est un point fixe de } S_\infty \end{array}$$

□

Pour obtenir un isomorphisme entre systèmes contractifs et types bruts, il ne reste plus qu'à démontrer l'énoncé réciproque, qui ne pose pas de difficulté.

Proposition 2.9 *Soit ρ une substitution brute régulière, de domaine fini V. Alors, il existe un système contractif S, dont le domaine contient V et dont l'unique solution, restreinte à V, coïncide avec ρ.*

Démonstration. Chaque $\rho(\alpha)$, où $\alpha \in V$, est un arbre régulier, donc admet un nombre fini de sous-arbres. Associons une variable de types à chacun d'eux ; en particulier, on associera α au sous-arbre $\rho(\alpha)$. On définit alors un système contractif en exprimant la relation entre chaque nœud et ses fils. Clairement, la solution de S coïncide avec ρ sur V. □

Chapitre 3

Contraintes

Une contrainte est simplement une inéquation formelle entre deux types. Elle exige que le premier soit un sous-type du second, et *contraint* ainsi les variables qui apparaissent dans ces types, en restreignant l'ensemble des valeurs qu'elles peuvent prendre. La notion de contrainte est centrale à notre théorie, puisque c'est par une conjonction de contraintes que nous approximerons le flot de données d'un programme. Le programme sera considéré comme correct si et seulement si ces contraintes admettent une solution.

Nous définissons d'abord la notion de contrainte (section 3.1). Nous introduisons ensuite l'opération de *décomposition* de contraintes (section 3.2), qui réduit une contrainte complexe en un ensemble de contraintes dites élémentaires. Nous expliquons ensuite comment représenter les contraintes de façon efficace au sein d'un *graphe de contraintes* (section 3.3). Enfin, la section 3.4 introduit un critère syntaxique, nommé *clôture*, permettant de déterminer si un tel graphe admet une solution.

3.1 Définitions

Définition 3.1 *Une* contrainte *est un couple formé d'un pos-type $\tau \in \mathcal{T}^+$ et d'un neg-type $\tau' \in \mathcal{T}^-$, noté $\tau \leq \tau'$.*

Définition 3.2 *Soit $k \in \mathbb{N}^+ \cup \{\infty\}$. Une substitution brute ρ est une k-*solution *de la contrainte $\tau \leq \tau'$ ssi $\rho(\tau) \leq_k \rho(\tau')$. Une* solution *est une ∞-solution. Une k-solution d'un ensemble de contraintes est une k-solution de chacun de ses éléments. Une contrainte, ou un ensemble de contraintes, est* soluble *ssi il admet une solution.*

Enfin, mentionnons que certaines fonctions élémentaires sur les types s'étendent aux contraintes :

Définition 3.3 *On pose*

$$\begin{aligned} \mathrm{fv}(\tau \leq \tau') &= \mathrm{fv}(\tau) \cup \mathrm{fv}(\tau') \\ \mathrm{h}(\tau \leq \tau') &= \max\{\mathrm{h}(\tau), \mathrm{h}(\tau')\} \\ \mathrm{pppv}(\tau \leq \tau') &= \min\{\mathrm{pppv}(\tau), \mathrm{pppv}(\tau')\} \end{aligned}$$

3.2 Décomposition de contraintes

La relation de sous-typage définie sur les types bruts est induite par leur structure de termes : comparer deux arbres revient à effectuer une comparaison élémentaire sur leurs constructeurs de tête, puis à comparer leurs sous-arbres (cf. proposition 1.2). Il en découle que toute contrainte $\tau \leq \tau'$ mettant en jeu deux types construits est soit clairement insoluble (si tête(τ) $\not\leq_b$ tête(τ')), soit équivalente à un ensemble (éventuellement vide) de contraintes entre les sous-termes de τ et τ'. Cette constatation permet de réduire toute contrainte à un ensemble de contraintes dites *élémentaires*. Nous appellerons ce procédé *décomposition structurelle*.

Définition 3.4 *Une contrainte $\tau \leq \tau'$ est* élémentaire *ssi les conditions suivantes sont vérifiées :*

- *τ et τ' sont des variables ou des types construits ;*
- *l'un au moins de $\{\tau, \tau'\}$ est une variable.*

Exemple. $\alpha \leq \beta$ et $\alpha \leq \beta \to \top$ sont élémentaires. $\alpha \sqcup \beta \leq \gamma$ et $\alpha_0 \to \alpha_1 \leq \beta_0 \to \beta_1$ ne le sont pas ; elles peuvent être décomposées, comme le montre la définition suivante.

Proposition 3.1 *Les règles suivantes – dont l'ordre est significatif – définissent une fonction* subc*, qui à toute contrainte soluble c associe un ensemble de contraintes élémentaires :*

$$\begin{aligned}
\mathrm{subc}(\sqcup S \leq \tau') &= \cup_{\tau \in S}\, \mathrm{subc}(\tau \leq \tau') \\
\mathrm{subc}(\tau \leq \sqcap S) &= \cup_{\tau' \in S}\, \mathrm{subc}(\tau \leq \tau') \\
\mathrm{subc}(c) &= \{c\} \textit{ si } c \textit{ est élémentaire} \\
\mathrm{subc}(\bot \leq \tau) &= \varnothing \\
\mathrm{subc}(\tau \leq \top) &= \varnothing \\
\mathrm{subc}(\tau_0 \to \tau_1 \leq \tau'_0 \to \tau'_1) &= \mathrm{subc}(\tau'_0 \leq \tau_0) \cup \mathrm{subc}(\tau_1 \leq \tau'_1)
\end{aligned}$$

Démonstration. Cette définition est bien fondée, puisque si c est une contrainte soluble, alors les contraintes apparaissant en argument de subc dans les membres droits sont de taille strictement inférieure et sont également solubles. De plus, on vérifie que ces règles couvrent tous les cas possibles ; en effet, si aucune règle ne s'applique à la contrainte $\tau \leq \tau'$, alors nécessairement tête(τ) $\not\leq_b$ tête(τ'), ce qui contredit le fait que la contrainte est supposée soluble. □

Proposition 3.2 *Pour tout $k \in \mathbb{N}^+ \cup \{\infty\}$, toute k-solution de* subc(c) *est $(k+d)$-solution de c, où $d =$* pppv(c)*. La réciproque est vraie quand $k = \infty$, i.e. toute solution de c est solution de* subc(c)*.*

Démonstration. Soit $k \in \mathbb{N}^+ \cup \{\infty\}$. Nous allons à présent démontrer, par induction sur la structure de c, que toute k-solution de subc(c) est $(k + \mathrm{pppv}(c))$-solution de c. Distinguons les cas suivants, dans l'ordre :

- c est de la forme $\sqcup S \leq \tau'$ ou $\tau \leq \sqcap S$. D'après la proposition 2.2, au moins un des éléments de S est une variable. Par conséquent, $\mathrm{pppv}(c) = 0$ et le résultat est immédiat.
- c est élémentaire. Alors un des membres est une variable. Il en découle que $\mathrm{pppv}(c) = 0$ et on conclut de la même manière.
- c est de la forme $\bot \leq \tau$ ou $\tau \leq \top$. Toute substitution brute est j-solution de c, quelle que soit la valeur de j ; d'où le résultat.
- c est de la forme $\tau_0 \to \tau_1 \leq \tau'_0 \to \tau'_1$. Soient c_0 et c_1 les contraintes $\tau'_0 \leq \tau_0$ et $\tau_1 \leq \tau'_1$, respectivement. Nous avons

$$\begin{aligned}
d &= \min\{\mathrm{pppv}(\tau_0 \to \tau_1), \mathrm{pppv}(\tau'_0 \to \tau'_1)\} \\
&= 1 + \min\{\mathrm{pppv}(\tau_0), \mathrm{pppv}(\tau_1), \mathrm{pppv}(\tau'_0), \mathrm{pppv}(\tau'_1)\} \\
&= 1 + \min\{\mathrm{pppv}(c_0), \mathrm{pppv}(c_1)\} \\
&= 1 + \min\{d_0, d_1\}
\end{aligned}$$

Par hypothèse d'induction, pour $i \in \{0,1\}$, toute k-solution de $\mathrm{subc}(c_i)$ est $(k+d_i)$-solution de c_i. Par conséquent,

$$\begin{array}{rl} & \rho \text{ est } k\text{-solution de } \mathrm{subc}(c) \\ \iff & \forall i \in \{0,1\} \quad \rho \text{ est } k\text{-solution de } \mathrm{subc}(c_i) \\ \Rightarrow & \forall i \in \{0,1\} \quad \rho \text{ est } (k+d_i)\text{-solution de } c_i \\ \Rightarrow & \rho \text{ est } (k+d)\text{-solution de } c \end{array}$$

La preuve de l'énoncé réciproque, dans le cas où $k = \infty$, ne présente pas de difficulté. $\square$

3.3 Graphes de contraintes

Au cours de l'analyse d'un programme, nous aurons bien sûr besoin de manipuler des conjonctions de contraintes. L'approche la plus simple, utilisée dans de nombreuses présentations de systèmes à base de types contraints, est de regrouper les contraintes au sein d'un ensemble. Cependant, nous avons constaté qu'il est possible de réduire toute contrainte à une conjonction de contraintes élémentaires (cf. section 3.2), et que deux bornes portant sur la même variable peuvent être combinées à l'aide des constructeurs $\sqcup$ et $\sqcap$ (cf. section 2.1). Il est possible de tirer parti de ces observations pour représenter une conjonction de contraintes par une structure mieux adaptée, que nous nommerons *graphe de contraintes.*

Définition 3.5 *Un* graphe de contraintes *C, de domaine $V \subset \mathcal{V}$, est composé de*

- *une relation réflexive entre éléments de V, notée $\leq_C$;*
- *pour tout $\alpha \in V$, un pos-type construit $C^{\downarrow}(\alpha) \in \mathcal{T}^+$ et un neg-type construit $C^{\uparrow}(\alpha) \in \mathcal{T}^-$, à variables dans V.*

Soit $k \in \mathbb{N}^+ \cup \{\infty\}$. Une substitution brute ρ est k-solution *de C ssi pour tous $\alpha, \beta \in V$:*

- *$\alpha \leq_C \beta$ implique $\rho(\alpha) \leq_k \rho(\beta)$;*
- *$\rho(C^{\downarrow}(\alpha)) \leq_k \rho(\alpha) \leq_k \rho(C^{\uparrow}(\alpha))$.*

On écrira $\rho \vdash C$ pour indiquer que ρ est solution de C.

Ayant défini la notion de solution, nous pouvons définir l'implication de contraintes. Cette notion est relativement centrale, bien qu'en partie éclipsée par la notion plus générale de comparaison polymorphe entre schémas de types. Elle sera étudiée en détail au chapitre 8.

Définition 3.6 *Soit $k \in \mathbb{N}^+ \cup \{\infty\}$. Un graphe de contraintes C* k-implique *une contrainte c ssi toute k-solution de C est k-solution de c. Ce fait sera noté $C \Vdash_k c$. L'*implication de contraintes *est, par définition, l'∞-implication, et est notée $\Vdash$.*

La propriété suivante permet d'établir une assertion d'implication de contraintes en procédant par approximations finies.

Proposition 3.3 *Si, pour tout k fini, on a $C \Vdash_k c$, alors $C \Vdash c$.*

Démonstration. Soient ρ une solution de c et $k \in \mathbb{N}^+$. ρ est, en particulier, k-solution de C, et $C \Vdash_k c$, donc ρ est k-solution de c. Puisque cela est vrai pour tout k, ρ est solution de c. $\square$

Au cours de nos manipulations de graphes de contraintes, il sera nécessaire de pouvoir déterminer si un graphe donné "contient" une contrainte donnée, et également de pouvoir "ajouter" une contrainte donnée à un graphe donné. Si nous avions choisi de travailler avec des ensembles de contraintes, il suffirait d'utiliser les notions classiques d'appartenance et d'union d'ensembles. Ici, nous devons donner des définitions légèrement plus complexes. Cependant, celles-ci restent fondées sur de simples critères syntaxiques. Le reste de cette section est consacré à leur définition.

Définition 3.7 *Une graphe de contraintes C* contient *une contrainte élémentaire c ssi l'une des conditions suivantes est vérifiée :*

- *c est de la forme $\alpha \leq \beta$, et $\alpha \leq_C \beta$;*
- *c est de la forme $\alpha \leq \tau$, où τ est un neg-type construit, et $C^\uparrow(\alpha)$ contient τ ;*
- *c est de la forme $\tau \leq \alpha$, où τ est un pos-type construit, et $C^\downarrow(\alpha)$ contient τ.*

C contient une contrainte quelconque c ssi $\text{subc}(c)$ *est défini et que C contient chacun de ses éléments.*

Notons que cette définition fait appel à la notion d'inclusion entre types introduite par la définition 2.11.

La proposition suivante confirme qu'il s'agit bien là, comme nous l'avons annoncé, de l'analogue de l'appartenance au sens des ensembles.

Proposition 3.4 *Soit $k \in \mathbb{N}^+ \cup \{\infty\}$. Si un graphe de contraintes C contient une contrainte c, alors toute k-solution de C est $(k+d)$-solution de c, où $d = \text{pppv}(c)$. En particulier, pour $k = \infty$, nous avons $C \Vdash c$.*

Démonstration. Soit ρ une k-solution de C. Supposons d'abord c élémentaire. Alors $\text{pppv}(c)$ est égal à 0, donc il nous faut montrer que ρ est k-solution de c. Trois cas se présentent, selon la forme de c, comme dans la définition 3.7.

- c est de la forme $\alpha \leq \beta$. Alors $\alpha \leq_C \beta$, d'après la définition 3.7. Donc, $\rho(\alpha) \leq_k \rho(\beta)$, et ρ est k-solution de c.
- c est de la forme $\alpha \leq \tau$, où τ est un neg-type construit. Alors $C^\uparrow(\alpha)$ contient τ ; c'est-à-dire, $C^\uparrow(\alpha) \sqcap \tau = C^\uparrow(\alpha)$. Ceci implique $\rho(C^\uparrow(\alpha)) \sqcap \rho(\tau) = \rho(C^\uparrow(\alpha))$, que l'on peut écrire $\rho(C^\uparrow(\alpha)) \leq \rho(\tau)$. Par ailleurs, puisque ρ est k-solution de C, nous avons $\rho(\alpha) \leq_k \rho(C^\uparrow(\alpha))$. $\leq_k$ contient $\leq_\infty$ et est transitive, donc $\rho(\alpha) \leq_k \rho(\tau)$, et ρ est k-solution de c.
- c est de la forme $\tau \leq \alpha$, où τ est un pos-type construit. Cas symétrique du précédent.

Nous avons montré que la proposition est vraie pour toute contrainte c élémentaire. Passons au cas général. Alors, $\text{subc}(c)$ est défini et C contient toute contrainte $c' \in \text{subc}(c)$. La proposition peut être appliquée à chaque c', donc ρ est k-solution de $\text{subc}(c)$. D'après la proposition 3.2, ρ est alors $(k+d)$-solution de c. $\square$

Passons à présent au second problème mentionné ci-dessus, celui de l'ajout d'une contrainte à un graphe.

Définition 3.8 *L'*ajout *d'une contrainte élémentaire c à un graphe de contraintes C, noté $C + c$, est le graphe de contraintes D défini comme suit :*

- *Si c est de la forme $\alpha \leq \beta$, alors $\leq_D = \{(\alpha, \beta)\} \cup \leq_C$, $D^\uparrow = C^\uparrow$ et $D^\downarrow = C^\downarrow$.*
- *Si c est de la forme $\alpha \leq \tau$ où τ est un neg-type construit, alors $\leq_D = \leq_C$, $D^\uparrow = C^\uparrow + [\alpha \mapsto \tau \sqcap C^\uparrow(\alpha)]$ et $D^\downarrow = C^\downarrow$.*
- *Si c est de la forme $\tau \leq \alpha$ où τ est un pos-type construit, alors $\leq_D = \leq_C$, $D^\uparrow = C^\uparrow$ et $D^\downarrow = C^\downarrow + [\alpha \mapsto \tau \sqcup C^\downarrow(\alpha)]$.*

(Si $\text{fv}(c) \not\subseteq V$, où $V = \text{dom}(C)$, alors C est d'abord étendu à $\text{fv}(c) \cup V$ en posant $\alpha \leq_C \alpha$, $C^\downarrow(\alpha) = \bot$ et $C^\uparrow(\alpha) = \top$ pour tout $\alpha \in \text{fv}(c) \setminus V$.)

Si c_1 et c_2 sont deux contraintes élémentaires, alors $(C + c_1) + c_2 = (C + c_2) + c_1$. Cela nous permet de définir l'ajout d'une contrainte quelconque c à un graphe C, également notée $C + c$, comme l'ajout de tous les éléments de $\text{subc}(c)$ *à C.*

La proposition suivante formalise l'intuition que cette opération correspond bien à l'ajout d'une contrainte.

Proposition 3.5 *Les solutions de $C + c$ sont exactement les solutions communes à C et c.*

Démonstration. Analyse par cas aisée. □

Enfin, nous étendons les renommages aux graphes de contraintes de la façon (fort prévisible) suivante :

Définition 3.9 *Soient ρ un renommage et C un graphe de contraintes de domaine V, tel que $V \subset \mathrm{dom}(\rho)$. Alors $\rho(C)$ est le graphe de contraintes D, de domaine $\rho(V)$, définit comme suit :*

- $\alpha \leq_D \beta \iff \rho^{-1}(\alpha) \leq_C \rho^{-1}(\beta)$;
- $D^{\uparrow} = \rho \circ C^{\uparrow} \circ \rho^{-1}$ *et* $D^{\downarrow} = \rho \circ C^{\downarrow} \circ \rho^{-1}$.

3.4 Solubilité d'un graphe de contraintes

Nous nous intéressons à présent à déterminer si un graphe de contraintes donné admet une solution. Cette opération est à la base de notre système de typage, puisqu'un programme sera considéré comme bien typé si et seulement si le graphe de contraintes qui lui est associé est soluble.

Nous définissons d'abord une propriété des graphes de contraintes, dite *clôture*, suffisante pour garantir l'existence d'une solution. Dans un deuxième temps, nous montrons que si un graphe est soluble, alors il existe un graphe clos qui lui est équivalent. Cette assertion est effective, et fournit donc un algorithme de décision du problème de solubilité d'un graphe de contraintes.

On notera que ce que nous appelons ici graphe «clos» est en général qualifié de «clos et cohérent» dans la littérature [16, 39, 41]. Ici, clôture et cohérence sont confondues en une seule notion. Cela simplifie légèrement la théorie, mais ne conduit à aucune différence pratique.

Définition 3.10 *Un graphe de contraintes C de domaine V est* clos *ssi les conditions suivantes sont vérifiées :*

- $\leq_C$ *est transitive ;*
- *pour tous $\alpha, \beta \in V$ tels que $\alpha \leq_C \beta$, $C^{\downarrow}(\beta)$ contient $C^{\downarrow}(\alpha)$ et $C^{\uparrow}(\alpha)$ contient $C^{\uparrow}(\beta)$;*
- *pour tout $\alpha \in V$, C contient $C^{\downarrow}(\alpha) \leq C^{\uparrow}(\alpha)$.*

Nous parlerons parfois de clôture forte*, plutôt que de clôture, par opposition aux notions de clôture moins restrictives (clôture faible, clôture simple) que nous développerons par la suite.*

Dans la définition ci-dessus, les deux premières conditions correspondent à une propriété de transitivité, tandis que la troisième est une combinaison de transitivité et de décomposition structurelle.

La transitivité de $\leq_C$ n'est en fait pas utilisée dans la preuve du théorème 3.1, qui établit que tout graphe clos est soluble. Nous l'incluons cependant dans la définition des graphes clos, car elle est importante du point de vue de l'implémentation. Elle permet, en particulier, d'implémenter l'algorithme de clôture de façon plus efficace. Par ailleurs, toujours du point de vue de l'implémentation, elle constitue un prérequis pour la phase de canonisation (cf. chapitre 11), qui suit la phase de clôture, et il est donc intéressant que les deux phases puissent s'enchaîner sans calculs intermédiaires.

Le lecteur intéressé par un critère de solubilité plus faible pourra consulter la définition de la *clôture faible* (définition 9.4). Celle-ci fait intervenir des mécanismes plus évolués, comme une axiomatisation de l'implication de contraintes, et ne nous sera utile que du point de vue théorique.

Théorème 3.1 *Tout graphe de contraintes clos admet une solution.*

Démonstration. Le principe de la preuve consiste à construire un système d'équations plus précis que le graphe de contraintes fourni, puis à en exhiber une solution et à vérifier qu'il s'agit également d'une solution du graphe de contraintes. Ici, nous choisissons d'imposer l'égalité entre chaque variable de types et sa borne inférieure construite. D'autres choix sont possibles. On pourrait choisir d'imposer l'égalité de chaque variable à sa borne supérieure construite, ou encore, en général, à tout type construit prouvablement compris entre ces deux bornes. Notre choix est donc arbitraire. Cela ne pose pas de problème, puisque notre but est de produire une solution quelconque, non de les énumérer toutes, ce qui serait une tâche beaucoup plus difficile.

Soit C un graphe de contraintes clos, de domaine V. Alors $C^{\downarrow}$ est un système contractif de domaine V ; d'après le théorème 2.1, il admet une solution ρ. ρ est une substitution brute de domaine V. Nous l'étendons à $\mathcal{V}$ de façon quelconque. Vérifions à présent que ρ est solution de C.

D'abord, soient $\alpha, \beta \in V$ tels que $\alpha \leq_C \beta$. C étant clos, $C^{\downarrow}(\beta)$ contient $C^{\downarrow}(\alpha)$. Par définition, cela signifie que $C^{\downarrow}(\beta) \sqcup C^{\downarrow}(\alpha) = C^{\downarrow}(\beta)$. En appliquant ρ à cette équation, on obtient

$$\begin{array}{cc} & \rho(C^{\downarrow}(\beta)) \sqcup \rho(C^{\downarrow}(\alpha)) = \rho(C^{\downarrow}(\beta)) \\ \iff & \rho(C^{\downarrow}(\alpha)) \leq \rho(C^{\downarrow}(\beta)) \\ \iff & \rho(\alpha) \leq \rho(\beta) \end{array}$$

Ensuite, soit $\alpha \in V$. Il est immédiat que $\rho(C^{\downarrow}(\alpha)) \leq \rho(\alpha)$, puisque ces expressions sont égales. Il reste à vérifier que $\rho(C^{\downarrow}(\alpha)) \leq \rho(C^{\uparrow}(\alpha))$. Nous ne pouvons le faire directement ; au lieu de cela, nous allons montrer, par induction sur $k \in \mathbb{N}^+$, que

$$\forall \alpha \in V \quad \rho(C^{\downarrow}(\alpha)) \leq_k \rho(C^{\uparrow}(\alpha))$$

L'assertion est vérifiée pour $k = 0$, puisque $\leq_0$ est uniformément vraie. Supposons qu'elle le soit pour un certain $k \in \mathbb{N}^+$. Alors ρ est k-solution de C. Soit $\alpha \in V$. Puisque C est clos, C contient la contrainte $C^{\downarrow}(\alpha) \leq C^{\uparrow}(\alpha)$, que nous noterons c. D'après la proposition 3.4, ρ est $(k+d)$-solution de c, où $d = \mathrm{pppv}(c)$. Or, puisque $C^{\downarrow}(\alpha)$ et $C^{\uparrow}(\alpha)$ sont des types construits, nous avons $d \geq 1$, donc ρ est $(k+1)$-solution de c. Ceci conclut la preuve par induction.

Grâce à la proposition 1.3, il découle du résultat ci-dessus que

$$\forall \alpha \in V \quad \rho(C^{\downarrow}(\alpha)) \leq \rho(C^{\uparrow}(\alpha))$$

d'où nous concluons que ρ est solution de C. □

Ce théorème constitue la base théorique de l'algorithme de clôture, qui permet de décider si un graphe de contraintes donné est soluble. Plutôt que de donner cet algorithme ici, nous en retardons la description jusqu'au chapitre 7. En effet, la preuve de l'algorithme utilise l'invariant des petits termes, exposé au chapitre 6.

La propriété de clôture est ainsi nommée à cause de la façon dont elle est définie, c'est-à-dire en dérivant de nouvelles conséquences logiques à partir des contraintes existantes et en vérifiant qu'elles sont en fait déjà présentes dans le graphe. Cependant, nous aurions également pu parler de graphes *résolus*, non seulement parce que tout graphe clos admet une solution, mais aussi parce qu'un tel graphe constitue en fait la meilleure description de son propre ensemble de solutions. En d'autres termes, on ne connaît aucun moyen finitaire d'énumérer toutes les solutions d'un graphe de contraintes clos, hormis la donnée du graphe lui-même.

Chapitre 4

Schémas de types

COMME CELUI DE ML, notre système de typage autorise le polymorphisme, introduit par la construction `let`. Un jugement de typage associera donc à un programme non pas simplement un type, mais un *schéma de types*, lequel représente (en première approximation) l'ensemble de ses *instances brutes*.

Cependant, suivant l'exemple de Trifonov et Smith [49], nous adoptons une présentation assez différente de celle de ML. Notre but est de faire en sorte que dans tout schéma de types, *toutes* les variables, sans exception, soient universellement quantifiées. En ML, les variables ne sont pas quantifiées lors de leur introduction ; elles ne le deviennent qu'après une opération de *généralisation* explicite, associée à la construction `let`. Ici, tout schéma de types est en quelque sorte généralisé au maximum, et l'opération `let` se contente d'introduire un schéma existant dans l'environnement, sans qu'aucune généralisation soit nécessaire.

L'avantage de cette décision est de donner naissance à un système *sans variables partagées*. Par exemple, un schéma de types n'ayant aucune variable libre, on peut parler de sa dénotation, c'est-à-dire (grosso modo) de l'ensemble de ses instances brutes, sans devoir préciser sous quel environnement. Pour comparer deux schémas de types, c'est-à-dire déterminer si l'un est plus général que l'autre, il suffira de comparer leur dénotations, toujours indépendamment de tout environnement. Par ailleurs, on remarquera que dans notre système de typage (introduit au chapitre 5), deux branches distinctes d'une même dérivation ne partagent aucune variable de types. Ainsi, le système devient en quelque sorte plus modulaire, plus élémentaire. Cela mène à une simplification considérable, et à une meilleure compréhension, de la théorie.

En ML, il est incorrect de généraliser une variable si celle-ci apparaît libre dans l'environnement. Alors, comment pouvons-nous prétendre généraliser toutes les variables ? La solution est de déplacer l'environnement pour le rendre partie intégrante du schéma de types. Ce procédé est connu sous le nom de λ-lifting ; son fonctionnement sera détaillé par les règles de typage (cf. chapitre 5). Plus précisément, les informations de typage concernant les variables liées par `let` restent stockées dans un environnement externe, tandis que celles qui concernent les variables liées par λ apparaissent dans un *contexte* interne au schéma de types.

Nous définissons d'abord les schémas de types (section 4.1), puis introduisons une relation de *comparaison polymorphe* entre schémas (section 4.2).

4.1 Schémas de types

Définition 4.1 *On se donne un ensemble dénombrable de* λ-identificateurs*, notés* $x, y, \ldots$ *Les* contextes *sont définis par*

$$\begin{array}{rcl} A & ::= & \varnothing \\ & \mid & A; x : \tau \end{array}$$

où τ est un neg-type.

Définissons quelques notations classiques :

Définition 4.2 *On pose*

$$
\begin{array}{ll}
(A; x : \tau)(x) = \tau & (A; x : \tau) \setminus x = A \\
(A; y : \tau)(x) = A(x) & (A; y : \tau) \setminus x = (A \setminus x); y : \tau
\end{array}
$$

$$
\begin{array}{rlrl}
\mathrm{fv}(\varnothing) &= \varnothing & \mathrm{dom}(\varnothing) &= \varnothing \\
\mathrm{fv}(A; x : \tau) &= \mathrm{fv}(A) \cup \mathrm{fv}(\tau) & \mathrm{dom}(A; x : \tau) &= \mathrm{dom}(A); x
\end{array}
$$

Définition 4.3 *Un* schéma de types *est de la forme*

$$\sigma ::= A \Rightarrow \tau \mid C$$

où A est un contexte, τ un pos-type et C un graphe de contraintes dont le domaine contient $\mathrm{fv}(A) \cup \mathrm{fv}(\tau)$. *(Le symbole | doit être interprété ici comme un litéral, non comme un choix.) σ est* clos *ssi C est clos. σ est* simple *ssi il ne contient que des types simples, c'est-à-dire s'il ne contient aucune occurrence de $\sqcup$ ou $\sqcap$.*

Intuitivement parlant, il faut considérer ici toutes les variables apparaissant dans un schéma de types comme universellement quantifiées. Cependant, nous ne ferons apparaître aucun quantificateur explicitement. Formellement parlant, aucune α-conversion implicite ne sera autorisée sur les schémas de types ; l'α-conversion sera traitée de façon explicite. Cette décision a plusieurs avantages : elle apporte plus de clarté, donc plus de rigueur, et elle permet de décrire de façon explicite le traitement des variables fraîches et des renommages.

Chaque schéma de types contient un contexte, qui décrit les hypothèses faites par l'expression considérée sur son environnement. En d'autres termes, si l'on type une expression ouverte, c'est-à-dire contenant des variables libres, alors le contexte inféré indiquera quels types ces variables doivent avoir pour que l'expression soit correcte. L'exemple le plus trivial en est l'expression ouverte `x`, à laquelle on peut associer le schéma de types

$$(\varnothing; x : \alpha) \Rightarrow \alpha \mid \varnothing$$

Ce schéma exprime une tautologie : si `x` a le type α, alors `x` a le type α. On notera que cela est vrai pour tout α ; il est donc correct de considérer ce schéma comme universellement quantifié. Pour plus de détails sur le fonctionnement des contextes, on se réfèrera à la description des règles de typage (section 5.3).

4.2 Comparaison de schémas

Nous allons à présent définir une relation de *comparaison polymorphe* entre schémas de types. Elle est basée sur le sous-typage entre types bruts, mais tient compte du fait qu'un schéma de types contient des variables universellement quantifiées. Cette relation joue un rôle fondamental dans notre théorie, puisqu'elle permettra ensuite de justifier l'ensemble des transformations et des simplifications que nous introduirons sur les schémas de types.

La définition de cette relation est fort simple. Elle provient de l'idée qu'un schéma de types n'est en fait qu'une façon de décrire un ensemble de typages bruts, que nous appellerons sa *dénotation*. Pour comparer deux schémas, il suffit de comparer leurs dénotations, par simple inclusion ensembliste.

Quelle est la dénotation d'un schéma $\sigma = A \Rightarrow \tau \mid C$? En d'autres termes, quels typages bruts ce schéma représente-t-il ? Au moins toutes ses instances brutes, c'est-à-dire tous les typages bruts obtenus par substitution. Bien sûr, il n'est légal d'appliquer une substitution brute à σ que si elle vérifie les contraintes de C ; donc, les instances brutes de σ sont les

$\rho(A \Rightarrow \tau)$, où ρ parcourt les solutions de C. Qui plus est, on remarque que les typages bruts sont ordonnés par sous-typage ; étant donné un typage brut correct, le cône qu'il engendre ne contiendra également que des typages corrects.

Formalisons à présent cette discussion.

Définition 4.4 *Les* contextes bruts *sont définis par*

$$\begin{array}{lcl} A & ::= & \varnothing \\ & \mid & A; x : \tau \end{array}$$

où τ est un type brut. La relation de sous-typage est étendue, point par point, aux contextes bruts de même domaine. Un typage brut *est de la forme*

$$A \Rightarrow \tau$$

où A est un contexte brut et τ un type brut. La relation de sous-typage est étendue aux typages bruts, de façon contravariante pour le contexte et covariante pour le type.

Définition 4.5 *Soit $A \Rightarrow \tau \mid C$ un schéma de types. Sa* dénotation *est l'union des cônes engendrés par ses instances brutes, c'est-à-dire*

$$\{A' \Rightarrow \tau' \,;\, \exists \rho \vdash C \quad \rho(A \Rightarrow \tau) \leq A' \Rightarrow \tau'\}$$

où $\rho(A \Rightarrow \tau)$ représente le typage brut $\rho(A) \Rightarrow \rho(\tau)$.

Définition 4.6 *Etant donnés deux schémas de types σ_1 et σ_2, on dit que le premier est* plus général *que le second, et on écrit $\sigma_1 \leq^{\forall} \sigma_2$, ssi la dénotation du premier contient celle du second.*

En d'autres termes, σ_1 est plus général que σ_2 ssi pour toute instance brute de σ_2, il existe une instance brute de σ_1 qui lui est inférieure. De façon formelle,

$$(A_1 \Rightarrow \tau_1 \mid C_1) \leq^{\forall} (A_2 \Rightarrow \tau_2 \mid C_2)$$

est donc équivalent à

$$\forall \rho_2 \vdash C_2 \quad \exists \rho_1 \vdash C_1 \quad \rho_1(A_1 \Rightarrow \tau_1) \leq \rho_2(A_2 \Rightarrow \tau_2)$$

On écrira $\sigma =^{\forall} \sigma'$ lorsque $\sigma \leq^{\forall} \sigma'$ et $\sigma' \leq^{\forall} \sigma$.

Enfin, pour pouvoir faire référence de façon succincte aux contraintes engendrées par la comparaison de deux schémas, nous donnons la définition suivante :

Définition 4.7 *La notation $A \leq A'$, où A et A' sont deux contextes de même domaine, est définie par*

$$\begin{array}{c} \varnothing \leq \varnothing = \varnothing \\ (A; x : \tau) \leq (A'; x : \tau') = (A \leq A') \cup \mathrm{subc}(\tau \leq \tau') \end{array}$$

Soient $A \Rightarrow \tau \mid C$ et $A' \Rightarrow \tau' \mid C'$ deux schémas. La notation $A \Rightarrow \tau \leq A' \Rightarrow \tau'$ représentera l'ensemble $(A' \leq A) \cup \mathrm{subc}(\tau \leq \tau')$.

La définition de la relation $\leq^{\forall}$ est due à Trifonov et Smith [49]. On remarque qu'elle généralise à la fois les assertions de solubilité et d'implication. En effet, l'assertion $\exists \rho \vdash C$, qui exprime que C est soluble, peut s'écrire $\top \mid C \leq^{\forall} \top \mid \varnothing$. Quant à l'assertion $C \Vdash \alpha \leq \beta$, elle est équivalente à $\gamma \rightarrow \gamma \mid \varnothing \leq^{\forall} \alpha \rightarrow \beta \mid C$. La décidabilité de l'implication de contraintes étant actuellement un problème ouvert, il en va de même de la comparaison entre schémas de types. Nous donnerons cependant, au chapitre 9 un algorithme de décision incomplet pour cette dernière. Comme on peut s'y attendre, cet algorithme généralisera à la fois l'algorithme de clôture, qui permet de décider la solubilité des graphes de contraintes, et l'algorithme (incomplet) d'implication de contraintes.

Dans [43], nous avions présenté une relation de comparaison moins puissante, qui ne tenait pas compte du fait que certaines variables étaient universellement quantifiées. Elle était définie à l'aide de la relation d'implication de contraintes. Elle est à présent inutile, sauf dans la preuve de correction du typage, où elle joue un rôle auxiliaire (cf. définition 5.8). Elle est suffisante pour justifier les simplifications à base de substitutions. Cependant, du fait qu'elle ne considère pas les variables comme quantifiées, elle interdit de nombreuses opérations ; par exemple les renommages, mais aussi l'élimination des variables inaccessibles, qui était justifiée dans [43] par une réécriture de l'ensemble de la dérivation de typage. C'est pourquoi Trifonov et Smith introduisent la relation $\leq^\forall$: les renommages, ainsi que le dépoussiérage (qui généralise l'élimination des variables inaccessibles) peuvent à présent se justifier par une simple application de la règle de sous-typage. On intègre ainsi au système des propriétés qui auparavant étaient méta-théoriques.

Chapitre 5

Système de typage

La raison d'être première d'un système de typage est d'éliminer tous les programmes qui provoquent une erreur à l'exécution. (La seconde pourrait être l'utilisation des types à des fins de documentation – la simplification entre alors en jeu – mais cela ne nous intéresse pas ici.)

Un système de typage consiste habituellement en un prédicat dit *jugement de typage*, à trois paramètres Γ, e et σ, noté $\Gamma \vdash e : \sigma$. Il est défini comme le plus petit prédicat stable par un certain jeu de *règles de typage*, c'est-à-dire qu'un jugement de typage est valide si et seulement si il peut être *dérivé* à l'aide de ces règles. Un programme e sera alors considéré comme correct si et seulement si il est bien typé dans l'environnement vide, c'est-à-dire si le prédicat $\varnothing \vdash e : \sigma$ est vérifié pour un certain σ.

Un système de typage doit être *sûr* : aucun programme déclaré correct ne doit provoquer d'erreur à l'exécution. Pour donner un sens à cette propriété, il faut donner une spécification formelle de l'exécution. Nous donnons donc une *sémantique opérationnelle*, qui présente l'exécution comme une série de *réductions*, c'est-à-dire de réécritures du texte du programme.

Enfin, pour présenter un intérêt pratique, un système de typage doit être *décidable* : on veut pouvoir vérifier automatiquement qu'un programme est bien typé. Mieux, ici, on souhaite réaliser *l'inférence de types*, qui consiste à déterminer si un programme est bien typé et à découvrir son type sans aucune indication de la part du programmeur. Le système de typage devra alors être accompagné d'une description de l'algorithme d'inférence. Pour permettre la programmation modulaire, il faudra que le type inféré soit le (ou plutôt un) type le plus général du programme fourni. Cela nécessite qu'un tel type existe en général, c'est-à-dire que le système de typage admette des *types principaux*.

Dans ce chapitre, nous exposons notre système de typage et montrons qu'il répond à tous les critères généraux mentionnés ci-dessus. Nous définissons d'abord rapidement le langage (section 5.1). Vient ensuite, après quelques préliminaires techniques (section 5.2), l'exposé du système de typage (section 5.3). La section 5.4 décrit un second système de typage, dit « simple », équivalent au premier en termes de programmes acceptés, mais moins puissant en termes de jugements de typage corrects. Ce système sera utilisé dans la preuve de correction. La section 5.5 décrit un troisième système, lui aussi équivalent au précédent mais plus restreint, et qui constitue en fait une spécification de l'algorithme d'inférence. Une fois les trois systèmes présentés, nous montrons leur équivalence (section 5.6) ; de plus, nous montrons l'existence de types principaux et vérifions qu'ils sont calculés par l'algorithme d'inférence. Enfin, la section 5.7 se charge de définir la sémantique et de vérifier que le système de typage est correct vis-à-vis de celle-ci.

5.1 Langage

Le langage qui nous intéresse ici est le noyau de ML, c'est-à-dire un λ-calcul doté d'une construction `let`. Pour plus de simplicité, nous séparons les identificateurs liés par λ de ceux liés par `let`, en les plaçant dans deux classes syntaxiques distinctes.

Définition 5.1 *On suppose donné un ensemble dénombrable de* `let`-identificateurs*, notés* $X, Y, \ldots$ *Les* expressions *sont définies par*

$$e ::= x \mid \lambda x.e \mid e\,e \mid X \mid \texttt{let}\, X = e\, \texttt{in}\ e$$

5.2 Définitions préliminaires

Nos règles de typage traitent l'environnement de façon légèrement inhabituelle. La partie de l'environnement associée aux variables introduites par la construction `let` apparaît, de façon classique, à gauche du signe $\vdash$ dans les jugements de typage. En revanche, la partie de l'environnement correspondant aux variables introduites par la construction λ est considérée comme faisant partie du schéma de types, et apparaît donc à droite du symbole $\vdash$. Cette présentation, parfois appelée λ-lifting, demande quelques définitions préliminaires, qui font l'objet de cette section.

Notons qu'une présentation du λ-*lifting* légèrement plus simple que la nôtre est possible, à condition de faire l'hypothèse que deux λ-abstractions distinctes ne lient pas la même variable. Etant donné un programme quelconque, on peut produire par renommage un programme équivalent qui satisfait cette propriété. Cependant, celle-ci n'est pas conservée par β-réduction. Cette présentation se révèle donc inadaptée lorsque l'on désire formuler un théorème de correction du typage vis-à-vis de la sémantique, et c'est pourquoi nous ne la développerons pas ici. Décrivons-la cependant de façon succincte. (Pour plus de détails, on pourra se référer à [49].) Comme ici, les environnements ne concernent que les variables X liées par `let`. Quant aux contextes inférés, ils ne mentionnent que les variables effectivement utilisées par l'expression, et non toutes les variables liées par λ, comme c'est le cas dans notre système. Les opérations « lift » et « unlift » disparaissent, puisque le phénomène de capture est éliminé. Cette présentation est donc moins « bureaucratique » ; nous l'adoptons d'ailleurs dans notre implémentation, pour des raisons d'efficacité. Il suffit, dans l'implémentation, de renommer au vol certaines variables pour satisfaire l'hypothèse d'unicité des noms de variables liées par λ.

Définition 5.2 *Les* environnements *sont définis par*

$$\begin{array}{rcl} \Gamma & ::= & \varnothing \\ & \mid & \Gamma; x \\ & \mid & \Gamma; X : \sigma \end{array}$$

Un environnement Γ associe un schéma de types σ à toute variable X liée par une construction `let`. Notons qu'il n'associe aucune information de type aux variables x liées par λ ; la présence de ces variables dans l'environnement ne sert qu'à gérer correctement les liaisons. L'association d'un type aux variables liées par λ se fait au sein du schéma de types.

Définition 5.3 *Le* domaine *et le* λ-domaine *d'un environnement sont définis par*

$$\begin{array}{rclcrcl} \mathrm{dom}(\varnothing) & = & \varnothing & \qquad & \mathrm{dom}_\lambda(\varnothing) & = & \varnothing \\ \mathrm{dom}(\Gamma; x) & = & \mathrm{dom}(\Gamma); x & & \mathrm{dom}_\lambda(\Gamma; x) & = & \mathrm{dom}_\lambda(\Gamma); x \\ \mathrm{dom}(\Gamma; X : \sigma) & = & \mathrm{dom}(\Gamma); X & & \mathrm{dom}_\lambda(\Gamma; X : \sigma) & = & \mathrm{dom}_\lambda(\Gamma) \end{array}$$

Définition 5.4 *La* soustraction *d'un λ-identificateur x à un environnement Γ, notée $\Gamma \setminus x$, est définie par*

$$\begin{array}{rcl} (\Gamma; x) \setminus x & = & \Gamma \\ (\Gamma; y) \setminus x & = & (\Gamma \setminus x); y \\ (\Gamma; X : \sigma) \setminus x & = & (\Gamma \setminus x); X : \sigma \end{array}$$

Les fonctions auxiliaires suivantes sont nécessaires lorsque l'on désire introduire ou éliminer une λ-abstraction.

Définition 5.5 *On définit*

$$\begin{aligned}
\mathrm{lift}_x(\varnothing) &= \varnothing \\
\mathrm{lift}_x(\Gamma; y) &= \mathrm{lift}_x(\Gamma); y \\
\mathrm{lift}_x(\Gamma; X : A \Rightarrow \tau \mid C) &= \mathrm{lift}_x(\Gamma); X : (A; x : \top) \Rightarrow \tau \mid C \\
\mathrm{unlift}_x(\varnothing) &= \varnothing \\
\mathrm{unlift}_x(\Gamma; y) &= \mathrm{unlift}_x(\Gamma); y \\
\mathrm{unlift}_x(\Gamma; X : A \Rightarrow \tau \mid C) &= \mathrm{unlift}_x(\Gamma); X : (A \setminus x) \Rightarrow \tau \mid C
\end{aligned}$$

Etant donnés un environnement Γ, une variable x et deux types τ et τ', on désire définir un contexte $\mathrm{single}_{(x,\tau,\tau')}(\Gamma)$, de même domaine que Γ, qui à x associe τ et à toute autre variable associe τ'. Cette définition sera utilisée dans la règle d'inférence (VAR$_{\mathrm{I}}$).

Définition 5.6 *Etant donnés une variable x et deux types τ, τ', on définit*

$$\begin{aligned}
\mathrm{single}_{(x,\tau,\tau')}(\Gamma; x) &= \mathrm{uni}_{\tau'}(\Gamma); x : \tau \\
\mathrm{single}_{(x,\tau,\tau')}(\Gamma; y) &= \mathrm{single}_{(x,\tau,\tau')}(\Gamma); y : \tau' \\
\mathrm{single}_{(x,\tau,\tau')}(\Gamma; X : \sigma) &= \mathrm{single}_{(x,\tau,\tau')}(\Gamma)
\end{aligned}$$

où $\mathrm{uni}_{\tau'}$ *est donnée par*

$$\begin{aligned}
\mathrm{uni}_{\tau'}(\varnothing) &= \varnothing \\
\mathrm{uni}_{\tau'}(\Gamma; x) &= \mathrm{uni}_{\tau'}(\Gamma); x : \tau' \\
\mathrm{uni}_{\tau'}(\Gamma; X : \sigma) &= \mathrm{uni}_{\tau'}(\Gamma)
\end{aligned}$$

5.3 Règles de typage

Les règles de typage sont données par la figure 5.1 page suivante.

Tout jugement de typage est de la forme $\Gamma \vdash e : \sigma$. On vérifie que tous les contextes A apparaissant dans ce jugement, que ce soit dans Γ ou dans σ, ont pour domaine $\mathrm{dom}_\lambda(\Gamma)$. L'opération lift_x utilisée dans la règle (ABS) a pour objet de faire respecter cet invariant.

Les règles de typage décrivent la façon dont l'analyse du programme engendre des contraintes de sous-typage. Reste à vérifier que ces contraintes admettent une solution, comme l'indique la définition suivante.

Définition 5.7 *Un expression e est* bien typée *dans un environnement Γ ssi il existe un schéma de types σ, de dénotation non vide, tel que $\Gamma \vdash e : \sigma$.*

Rappelons que la dénotation d'un schéma $A \Rightarrow \tau \mid C$ est non vide si et seulement si C admet une solution. Pour déterminer si un programme est bien typé, il faut donc non seulement construire une dérivation de typage, mais également vérifier que le graphe de contraintes obtenu est soluble. Pour cela, un algorithme, fondé sur un calcul de *clôture*, sera fourni au chapitre 7.

Une règle est associée à chaque construction syntaxique ; de plus, la règle (SUB), dite « règle de sous-typage », permet de remanier le schéma de types à tout instant, de façon extrêmement flexible, puisqu'elle utilise toute la puissance de la relation de comparaison polymorphe entre schémas de types. En particulier, cette règle autorise des α-conversions arbitraires, conformément au fait que toutes les variables d'un schéma de types sont considérées comme universellement quantifiées. Par ailleurs, toutes nos méthodes de simplification se présenteront sous la forme d'algorithmes acceptant un schéma σ et produisant un schéma

$$\frac{\mathrm{dom}(A) = \mathrm{dom}_\lambda(\Gamma)}{\Gamma \vdash x : A \Rightarrow A(x) \mid C} \quad (\textsc{Var})$$

$$\frac{\mathrm{lift}_x(\Gamma); x \vdash e : (A; x : \tau) \Rightarrow \tau' \mid C}{\Gamma \vdash \lambda x.e : A \Rightarrow (\tau \to \tau') \mid C} \quad (\textsc{Abs})$$

$$\frac{\Gamma \vdash e_1 : A \Rightarrow (\tau_2 \to \tau) \mid C \qquad \Gamma \vdash e_2 : A \Rightarrow \tau_2 \mid C}{\Gamma \vdash e_1\, e_2 : A \Rightarrow \tau \mid C} \quad (\textsc{App})$$

$$\frac{\Gamma(X) = \sigma}{\Gamma \vdash X : \sigma} \quad (\textsc{LetVar})$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \qquad \Gamma; X : \sigma_1 \vdash e_2 : A_2 \Rightarrow \tau_2 \mid C_2 \qquad \sigma_1 \leq^\forall A_2 \Rightarrow \top \mid C_2}{\Gamma \vdash \mathtt{let}\, X = e_1 \,\mathtt{in}\; e_2 : A_2 \Rightarrow \tau_2 \mid C_2} \quad (\textsc{Let})$$

$$\frac{\Gamma \vdash e : \sigma \qquad \sigma \leq^\forall \sigma'}{\Gamma \vdash e : \sigma'} \quad (\textsc{Sub})$$

FIG. 5.1: Règles de typage

σ' équivalent ; une simple application de la règle (SUB) sera donc suffisante pour en justifier la correction. Bien sûr, en contrepartie, la puissance de la règle (SUB) est gênante dans la preuve de correction du typage : à cause de cette règle, il est très difficile de montrer directement la stabilité du typage par réduction. Aussi serons-nous amenés, pour les besoins de la démonstration, à introduire un second jeu de règles, appelées règles de typage « simples », où la règle de sous-typage est sensiblement affaiblie.

Ces règles de typage sont très voisines de celles de Trifonov et Smith [49]. Les différences principales résident dans le traitement du λ-*lifting*, dont nous donnons une description plus lourde mais mieux adaptée à la preuve de stabilité du typage par réduction, et dans la formulation de la règle (LET).

Le premier de ces points a été mentionné dans la section 5.2, où nous avons brièvement décrit les différences entre les deux présentations possibles du λ-*lifting*. Cependant, nous n'avons pas abordé le principe même du λ-*lifting*, c'est-à-dire la façon dont ce mécanisme permet d'obtenir un système de typage au comportement similaire à celui de ML, tout en manipulant exclusivement des variables universellement quantifiées. Donnons donc brièvement un exemple de son fonctionnement. La question est : comment attribuer un type « monomorphe » à une expression, quand on ne dispose que de variables universellement quantifiées ? Nous allons y répondre, de façon légèrement informelle. Considérons l'expression

$$\lambda x.\mathtt{let}\, Y = x \,\mathtt{in}\; (Y, Y)$$

Typons cette expression en ML. Le type de Y est une variable monomorphe α. Les deux utilisations de Y ne donnent donc lieu à aucune instantiation, et le type de l'expression est $\alpha \to \alpha \times \alpha$. Dans notre système, au contraire, le type de Y est $(x : \alpha) \Rightarrow \alpha$, d'après la règle (VAR). Ici, α est (implicitement) universellement quantifiée. Les deux utilisations de Y pourraient donc donner lieu, si on était libre d'utiliser la règle (SUB) pour effectuer des renommages, à deux schémas $(x : \beta) \Rightarrow \beta$ et $(x : \gamma) \Rightarrow \gamma$. Cependant, la règle de typage des paires exige que ses deux branches partagent un même contexte. (Notre langage ne contient en fait pas les paires ; mais la règle de construction des paires peut se dériver de la règle d'application, dans laquelle on retrouve cette même condition : les deux branches doivent utiliser le même contexte A.) Nécessairement, β et γ sont donc une seule et même variable, et le type de la paire (Y, Y) est $(x : \beta) \Rightarrow \beta \times \beta$. Une fois la λ-abstraction effectuée, l'expression complète reçoit le type $\beta \to \beta \times \beta$, comme attendu. En résumé, toutes les variables apparaissant dans le contexte se comportent en fait de façon monomorphe ; cela provient de la condition de partage des contextes imposée à chaque fois que deux branches de l'arbre d'inférence se rejoignent. Le système est donc correct, mais parvient à éliminer la notion de variable non quantifiée.

Le second point est la formulation de la règle (LET). Trifonov et Smith [49] proposent une règle plus simple, qui s'écrit

$$\frac{\Gamma \vdash e_1 : \sigma_1 \qquad \Gamma; X : \sigma_1 \vdash e_2 : \sigma_2}{\Gamma \vdash \mathtt{let}\, X = e_1 \;\mathtt{in}\; e_2 : \sigma_2} \; (\text{LET'})$$

Cette règle est d'une élégante simplicité. Malheureusement, elle est incorrecte vis-à-vis de la sémantique d'appel par valeur. Le problème se manifeste lorsque la variable X n'est pas utilisée dans l'expression e_2. Par exemple, dans l'expression

$$\lambda x.\mathtt{let}\, X = x + 1 \;\mathtt{in}\; x$$

la variable X reçoit le schéma $(x : \mathtt{int}) \Rightarrow \mathtt{int}$, qui rend compte du fait que x est nécessairement un entier. Cependant, X n'étant pas utilisée dans le corps de la construction `let`, cette information est perdue, et l'expression complète obtient le schéma de types $\alpha \to \alpha$, qui est incorrect. Pour éviter ce problème, Trifonov et Smith [49] proposent de remplacer la construction $\mathtt{let}\, X = e_1 \;\mathtt{in}\; e_2$ par $\mathtt{let}\, X = e_1 \;\mathtt{in}\; (\lambda\_.e_2)\, X$ lorsque X n'apparaît pas dans e_2. On ajoute ainsi explicitement une occurrence de X, qui n'a pas d'effet sur la sémantique mais conduit à un typage correct. Cette solution est acceptable en pratique, mais encore une fois, se prête mal aux preuves de stabilité du typage par réduction. Aussi préférons-nous introduire une règle (LET) légèrement plus lourde, mais correcte. Pour expliquer la formulation de celle-ci, il suffit de remarquer qu'elle correspond exactement à ce que l'on obtiendrait si on utilisait les règles (LET'), (SUB), etc. pour typer l'expression $\mathtt{let}\, X = e_1 \;\mathtt{in}\; (\lambda\_.e_2)\, X$.

5.4 Règles de typage « simples »

Les règles de typage introduites précédemment sont puissantes. En particulier, la règle (SUB), basée sur la relation de comparaison polymorphe entre schémas de types, est très flexible. Paradoxalement, cela est gênant lorsque l'on désire montrer la correction du système de typage vis-à-vis de la sémantique. En effet, la règle (SUB) est si puissante que sa correction n'est pas immédiate ; le problème se manifeste principalement lors de la preuve du lemme de β-réduction.

Les règles d'inférence, que nous allons bientôt introduire, sont moins flexibles ; en particulier, elles ne possèdent pas d'analogue à la règle (SUB). On pourrait donc imaginer de prouver la correction des règles d'inférence et d'en déduire ensuite celle des règles initiales. Cependant, les règles d'inférence sont relativement complexes, car elles doivent décrire avec précision l'algorithme d'inférence ; la preuve de correction en serait inutilement compliquée.

Nous introduisons donc ici un troisième jeu de règles, dites « simples », données par la figure 5.2 page suivante. Elles sont aussi simples d'expression que les règles de typage, mais moins puissantes, car elles n'utilisent pas la comparaison polymorphe entre schémas.

Ce nouveau jeu de règles est écrit à partir de la remarque suivante. La règle (SUB) originale a deux vocations principales : le renommage d'un schéma de types et la modification de son graphe de contraintes. (Une troisième, et fondamentale, utilisation de la comparaison polymorphe entre schémas est la simplification du schéma de types ; mais celle-ci ne nous intéresse pas ici puisqu'elle n'affecte pas l'ensemble des programmes typables.) Ces deux utilisations peuvent être séparées et décrites de façon plus élémentaire. Le renommage peut

$$\frac{\mathrm{dom}(A) = \mathrm{dom}_\lambda(\Gamma)}{\Gamma \vdash_S x : A \Rightarrow A(x) \mid C} \quad (\mathrm{VAR_S})$$

$$\frac{\mathrm{lift}_x(\Gamma); x \vdash_S e : (A; x : \tau) \Rightarrow \tau' \mid C}{\Gamma \vdash_S \lambda x.e : A \Rightarrow (\tau \to \tau') \mid C} \quad (\mathrm{ABS_S})$$

$$\frac{\Gamma \vdash_S e_1 : A \Rightarrow (\tau_2 \to \tau) \mid C \qquad \Gamma \vdash_S e_2 : A \Rightarrow \tau_2 \mid C}{\Gamma \vdash_S e_1\, e_2 : A \Rightarrow \tau \mid C} \quad (\mathrm{APP_S})$$

$$\frac{\Gamma(X) = \sigma \qquad \rho \text{ renommage de } \sigma}{\Gamma \vdash_S X : \rho(\sigma)} \quad (\mathrm{LETVAR_S})$$

$$\frac{\begin{array}{c}\Gamma \vdash_S e_1 : \sigma_1 \qquad \Gamma; X : \sigma_1 \vdash_S e_2 : A_2 \Rightarrow \tau_2 \mid C_2 \\ \sigma_1 \leq^m A_2 \Rightarrow \top \mid C_2\end{array}}{\Gamma \vdash_S \mathtt{let}\, X = e_1 \,\mathtt{in}\, e_2 : A_2 \Rightarrow \tau_2 \mid C_2} \quad (\mathrm{LET_S})$$

$$\frac{\Gamma \vdash_S e : \sigma \qquad \sigma \leq^m \sigma'}{\Gamma \vdash_S e : \sigma'} \quad (\mathrm{SUB_S})$$

FIG. 5.2: Règles de typage « simples »

être explicitement intégré à la règle d'accès à l'environnement (LETVAR$_S$). La modification du graphe de contraintes peut être réalisée à l'aide d'une règle (SUB$_S$) affaiblie, dans laquelle la comparaison polymorphe entre schémas est remplacée par une comparaison dite *monomorphe*, définie comme suit.

Définition 5.8 *La* comparaison monomorphe entre schémas de types, *notée $\leq^m$, est définie comme suit. Etant donnés deux schémas de types $\sigma_1 = (A_1 \Rightarrow \tau_1 \mid C_1)$ et $\sigma_2 = (A_2 \Rightarrow \tau_2 \mid C_2)$, $\sigma_1 \leq^m \sigma_2$ ssi*

$$C_2 \Vdash C_1 + (A_1 \Rightarrow \tau_1 \leq A_2 \Rightarrow \tau_2)$$

Lemme 5.1 *$\sigma_1 \leq^m \sigma_2$ implique $\sigma_1 \leq^\forall \sigma_2$.*

Démonstration. Supposons $\sigma_1 \leq^m \sigma_2$. Soit ρ_2 une solution de C_2. Alors, d'après la définition de l'implication de contraintes, ρ_2 satisfait également C_1 et $A_1 \Rightarrow \tau_1 \leq A_2 \Rightarrow \tau_2$. Par conséquent, ρ_2 est un témoin de l'assertion $\sigma_1 \leq^\forall \sigma_2$. □

Notons que la règle (SUB$_S$) est celle que l'on trouvait dans [43] (modulo l'utilisation du λ-*lifting*).

5.5 Règles d'inférence

Les règles de typage présentées précédemment ne définissent pas directement un algorithme. Premièrement, elles ne sont pas dirigées par la syntaxe, puisque la règle (SUB) peut être appliquée à tout moment. Deuxièmement, la règle (APP) impose des contraintes de partage entre ses prémisses : A, τ_2 et C apparaissent dans les deux prémisses. Cela nécessite de faire à l'avance le bon choix lorsque l'on applique la règle (VAR).

Nous donnons ici un jeu de règles d'inférence de types. Celles-ci sont dirigées par la syntaxe et ne présentent aucune contrainte de partage. De plus, le mécanisme de création de

$$\frac{\alpha \notin F \qquad A = \mathrm{single}_{(x,\alpha,\top)}(\Gamma)}{[F]\ \Gamma \vdash_{\mathrm{I}} x : [F \cup \{\alpha\}]\ A \Rightarrow \alpha \mid \varnothing} \quad (\textsc{Var}_{\mathrm{I}})$$

$$\frac{[F]\ \mathrm{lift}_x(\Gamma); x \vdash_{\mathrm{I}} e : [F']\ (A; x : \tau) \Rightarrow \tau' \mid C}{[F]\ \Gamma \vdash_{\mathrm{I}} \lambda x.e : [F']\ A \Rightarrow (\tau \to \tau') \mid C} \quad (\textsc{Abs}_{\mathrm{I}})$$

$$\frac{\begin{array}{c}[F]\ \Gamma \vdash_{\mathrm{I}} e_1 : [F']\ A_1 \Rightarrow \tau_1 \mid C_1 \\ [F']\ \Gamma \vdash_{\mathrm{I}} e_2 : [F'']\ A_2 \Rightarrow \tau_2 \mid C_2 \qquad \alpha \notin F''\end{array}}{\begin{array}{c}[F]\ \Gamma \vdash_{\mathrm{I}} e_1\, e_2 : [F'' \cup \{\alpha\}]\ (A_1 \sqcap A_2) \Rightarrow \alpha \mid C \\ \text{où } C = (C_1 \cup C_2) + (\tau_1 \leq \tau_2 \to \alpha)\end{array}} \quad (\textsc{App}_{\mathrm{I}})$$

$$\frac{\Gamma(X) = \sigma \qquad \rho \text{ renommage de } \sigma \qquad \mathrm{im}(\rho) \cap F = \varnothing}{[F]\ \Gamma \vdash_{\mathrm{I}} X : [F \cup \mathrm{im}(\rho)]\ \rho(\sigma)} \quad (\textsc{LetVar}_{\mathrm{I}})$$

$$\frac{\begin{array}{c}[F]\ \Gamma \vdash_{\mathrm{I}} e_1 : [F']\ A_1 \Rightarrow \tau_1 \mid C_1 \\ [F']\ \Gamma; X : A_1 \Rightarrow \tau_1 \mid C_1 \vdash_{\mathrm{I}} e_2 : [F'']\ A_2 \Rightarrow \tau_2 \mid C_2\end{array}}{[F]\ \Gamma \vdash_{\mathrm{I}} \mathtt{let}\, X = e_1\ \mathtt{in}\ e_2 : [F'']\ (A_1 \sqcap A_2) \Rightarrow \tau_2 \mid C_1 \cup C_2} \quad (\textsc{Let}_{\mathrm{I}})$$

FIG. 5.3: Règles d'inférence de types

« variables fraîches » y est explicitement spécifié. Nous vérifierons que grâce à ces propriétés, les règles d'inférence décrivent directement un algorithme.

Remarquons que si nos règles de typage sont très proches de celles de Trifonov et Smith [49], nos règles d'inférence présentent des différences plus sensibles. En effet, nous les formulons de façon plus rigoureuse, décrivant ainsi directement un algorithme. De plus, elles seront raffinées à deux reprises, aux chapitres 6 et 12, de façon à respecter certains invariants.

Les règles d'inférence sont données par la figure 5.3. Commentons les modifications apportées par rapport aux règles de typage. (On ignorera pour l'instant les annotations de la forme « $[F]$ ».) De façon synthétique, l'unique différence fondamentale est la disparition de la règle de sous-typage, qui a été combinée avec la règle d'application.

- La règle ($\textsc{Var}_{\mathrm{I}}$) place l'hypothèse $x : \alpha$ dans le contexte. Sous cette hypothèse, l'expression x a le type α. Les variables autres que x ne sont pas utilisées, et reçoivent donc le type $\top$ dans le contexte inféré.
- La règle de λ-abstraction type le corps de l'abstraction dans un environnement étendu. Notons que ce nouvel environnement n'associe pas d'information de type à la variable x. A l'inverse, c'est dans le contexte inféré que l'on trouve quel type τ est attendu pour x. La règle ($\textsc{Abs}_{\mathrm{I}}$) retire ce type du contexte et le fait passer à gauche du symbole $\to$. Cette règle est essentiellement identique à ($\textsc{Abs}$).
- La règle ($\textsc{App}_{\mathrm{I}}$) infère séparément un schéma de types pour chacun des deux composants de l'expression. Puis, elle réunit leurs exigences en calculant l'intersection de leurs contextes et l'union de leurs graphes de contraintes (en l'occurrence, il s'agit d'une simple union au sens de la théorie des ensembles, puisque les graphes ne partagent aucune variable de types, comme nous le verrons plus loin). Ensuite, elle ajoute la contrainte $\tau_1 \leq \tau_2 \to \alpha$. Pourquoi ? Le type inféré pour l'opérande gauche est τ_1. Par ailleurs, cet opérande se voit passer un argument de type τ_2, et doit renvoyer un résultat de type α, puisque tel est le type choisi pour l'expression complète. Donc, l'opérande gauche a le type τ_1, mais est utilisé avec le type $\tau_2 \to \alpha$. Grâce au sous-typage, il n'est pas nécessaire d'exiger que ces deux types soient égaux ; il suffit de

demander que le premier soit sous-type du second, ce que nous effectuons en ajoutant la contrainte appropriée au graphe de contraintes.

– La règle (LETVAR$_\mathrm{I}$) est essentiellement identique à la règle (LETVAR$_\mathrm{S}$) : elle décharge le schéma de types de l'environnement, et en fait une copie de façon à permettre le polymorphisme. Cette opération de copie doit être explicitement spécifiée ici, comme dans les règles de typage « simples », du fait de l'absence de la règle (SUB), qui permettait un renommage à tout moment.
– Enfin, la règle associée à la construction `let` a été modifiée de façon similaire à la règle d'application, c'est-à-dire qu'elle calcule l'intersection des contextes et l'union des graphes de contraintes des deux branches. L'équivalent de cette opération était effectué, dans la règle (LET) initiale, par la troisième prémisse. Rappelons que celle-ci s'obtenait naturellement en typant l'expression $\mathtt{let}\, X = e_1 \;\mathtt{in}\; (\lambda\_.e_2)\, X$ à l'aide d'une règle (LET') simplifiée. Il en va en fait de même ici : on peut dire que les opérations d'intersection des contextes et d'union des graphes de contraintes proviennent essentiellement de la règle (APP$_\mathrm{I}$) utilisée pour typer l'application $(\lambda\_.e_2)\, X$.

Nous pouvons maintenant expliquer les annotations « $[F]$ ». Les jugements d'inférence sont de la forme

$$[F]\; \Gamma \vdash_\mathrm{I} e : [F']\; \sigma$$

Le lecteur l'aura deviné, ces annotations servent à traiter explicitement la question des variables « fraîches ». F et F' sont des ensembles (finis) de variables de type qui ont déjà été utilisées et ne doivent pas être réutilisées. F est un argument de l'algorithme d'inférence, tandis que F' en est un résultat. Grâce à ces annotations, les variables fraîches sont traitées de façon entièrement formelle. Bien que relativement lourdes, elles sont très simples d'usage. Rappelons que toute α-conversion implicite des schémas de types est interdite. Cela est nécessaire pour donner un sens à ces annotations, et cela permet aux règles d'inférence de refléter très fidèlement l'implémentation.

F étant, par hypothèse, l'ensemble de variables de type « sales », choisir une variable « fraîche » revient simplement à la choisir en dehors de F ; d'où les prémisses de la forme $\alpha \notin F$. En contrepartie, une fois qu'une variable fraîche α a été utilisée, elle ne peut plus être réutilisée ailleurs ; le nouvel ensemble de variables « sales » sera donc $F \cup \{\alpha\}$.

Notons que dans la règle d'application (APP$_\mathrm{I}$), l'ensemble de variables sales F', obtenu en résultat de la première prémisse, est passé en argument à la seconde prémisse. (La règle (LET$_\mathrm{I}$) procède de façon similaire.) En conséquence, deux sous-arbres distincts de l'arbre d'inférence ne partagent aucune variable de types. Cette propriété, formalisée par le lemme 5.2 plus loin, est une différence importante vis-à-vis d'un système d'inférence classique comme celui de ML, où *l'environnement* est est un *argument* partagé par des sous-arbres distincts. Ici, au contraire, le *contexte* est un *résultat* et il n'y a aucun partage. Cette propriété est essentielle pour la simplification de schémas de types, puisqu'elle nous permet de travailler uniquement sur des schémas de types sans variables libres. Ainsi, la simplification ne s'intéresse qu'à un unique schéma, et n'a pas à prendre en compte un environnement de variables partagées.

Pour conclure l'exposé des règles d'inférence, donnons un exemple de leur fonctionnement.

Exemple. On désire inférer le schéma de types du λ-terme $\lambda fx.f\,(f\,x)$, qui compose une fonction f avec elle-même. Le terme étant une λ-abstraction par rapport aux deux variables f et x, la dérivation se terminera par deux instances de la règle (ABS$_\mathrm{I}$) ; le corps du programme, c'est-à-dire l'expression $f\,(f\,x)$, devra donc être typé dans l'environnement $\Gamma = (\varnothing; f; x)$.

On utilise d'abord la règle (VAR$_\mathrm{I}$) pour typer la première occurrence de f :

$$[\varnothing]\; \Gamma \vdash_\mathrm{I} f : [\{v_1\}]\; (\varnothing; f : v_1; x : \top) \Rightarrow v_1 \mid \varnothing$$

Puis, on fait de même pour la seconde occurrence de f et pour l'occurrence de x. On prend garde, à chaque fois, à garder trace de l'ensemble de variables « sales » fourni par les calculs précédents ; ainsi, on obtient toujours des variables fraîches.

$$[\{v_1\}]\ \Gamma \vdash_{\mathrm{I}} f : [\{v_1, v_2\}]\ (\varnothing; f : v_2; x : \top) \Rightarrow v_2 \mid \varnothing$$
$$[\{v_1, v_2\}]\ \Gamma \vdash_{\mathrm{I}} x : [\{v_1, v_2, v_3\}]\ (\varnothing; f : \top; x : v_3) \Rightarrow v_3 \mid \varnothing$$

On peut alors utiliser la règle ($\textsc{App}_{\mathrm{I}}$) pour typer l'application $(f\,x)$. Celle-ci engendre une première contrainte de sous-typage. (Pour plus de brièveté, on représente ici le graphe de contraintes par un ensemble.)

$$[\{v_1\}]\ \Gamma \vdash_{\mathrm{I}} (f\,x) : [\{v_1, v_2, v_3, v_4\}]\ (\varnothing; f : v_2; x : v_3) \Rightarrow v_4 \mid \{v_2 \leq v_3 \rightarrow v_4\}$$

Cela fait, on passe à la seconde application, qui engendre une nouvelle contrainte :

$$[\varnothing]\ \Gamma \vdash_{\mathrm{I}} f\,(f\,x) : [V]\ (\varnothing; f : v_1 \sqcap v_2; x : v_3) \Rightarrow v_5 \mid C$$
$$\text{où } V = \{v_1, v_2, v_3, v_4, v_5\}$$
$$\text{et } C = \{v_2 \leq v_3 \rightarrow v_4,\ v_1 \leq v_4 \rightarrow v_5\}$$

On peut maintenant appliquer deux fois la règle ($\textsc{Abs}_{\mathrm{I}}$), qui retire la dernière entrée du contexte et l'utilise pour créer un type de fonction :

$$[\varnothing]\ (\varnothing; f) \vdash_{\mathrm{I}} \lambda x.f\,(f\,x) : [V]\ (\varnothing; f : v_1 \sqcap v_2) \Rightarrow v_3 \rightarrow v_5 \mid C$$
$$[\varnothing]\ \varnothing \vdash_{\mathrm{I}} \lambda fx.f\,(f\,x) : [V]\ \varnothing \Rightarrow (v_1 \sqcap v_2) \rightarrow v_3 \rightarrow v_5 \mid C$$

Pour vérifier que le λ-terme est bien typé, il ne reste plus qu'à vérifier que le graphe de contraintes C admet une solution, ce qui est immédiat, puisqu'il est clos. Une fois prouvée la sûreté du typage, nous aurons donc la garantie que ce programme ne provoque pas d'erreur d'exécution. Notons que le type inféré n'est pas très lisible, ce qui indique qu'une simplification est nécessaire. Une fois appliquées nos méthodes de simplification internes (canonisation, dépoussiérage) et externes (c'est-à-dire uniquement destinées à l'affichage, cf. section 15.2), nous obtiendrons le jugement de typage

$$\varnothing \vdash \lambda fx.f\,(f\,x) : (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta) \mid \{\beta \leq \alpha\}$$

qui fournit une spécification claire de notre λ-terme : il attend une fonction pouvant être composée avec elle-même (comme l'indique la contrainte $\beta \leq \alpha$), et renvoie une fonction du même type.

5.6 Equivalence des trois jeux de règles

Dans cette section, nous montrons que les trois systèmes présentés plus haut sont équivalents. Cela implique, en particulier, qu'ils acceptent le même ensemble de programmes. Plus précisément, nous vérifions que chaque système est *correct* et *complet* vis-à-vis de ceux qui le précèdent. Nous en déduisons que dans le système de typage original, toute expression admet un *type principal*, lequel est calculé par l'algorithme d'inférence.

L'équivalence des trois systèmes se résume en trois théorèmes : correction des règles de typage simples vis-à-vis des règles de typage, correction des règles d'inférence vis-à-vis des règles de typage simple, et complétude des règles d'inférence vis-à-vis des règles de typage.

Théorème 5.1 *Les règles de typage simples sont correctes vis-à-vis des règles de typage ; c'est-à-dire,*

$$\Gamma \vdash_{\mathrm{S}} e : \sigma$$

implique

$$\Gamma \vdash e : \sigma$$

Démonstration. Par induction sur la structure de la dérivation hypothèse. Il y a un cas par règle de typage simple. Les cas (Var$_\mathrm{S}$), (Abs$_\mathrm{S}$) et (App$_\mathrm{S}$) sont immédiats. Le cas (LetVar$_\mathrm{S}$) découle de la remarque que pour tout renommage ρ, on a $\sigma \leq^\forall \rho(\sigma)$; le but peut donc être obtenu en faisant suivre (LetVar) par une instance de (Sub). Les cas (Let$_\mathrm{S}$) et (Sub$_\mathrm{S}$) découlent du fait que la relation $\leq^m$ est incluse dans $\leq^\forall$ (lemme 5.1). □

Théorème 5.2 *Les règles d'inférence sont correctes vis-à-vis des règles de typage simples ; c'est-à-dire,*

$$[F]\ \Gamma \vdash_\mathrm{I} e : [F']\ \sigma$$

implique

$$\Gamma \vdash_\mathrm{S} e : \sigma$$

Démonstration. Par induction sur la structure de la dérivation hypothèse. Il y a un cas par règle d'inférence. Chacun des cas reprend exactement les notations de la figure 5.3, ce qui nous permet d'omettre l'hypothèse.

- (Var$_\mathrm{I}$) Il suffit de vérifier que le contexte $\mathrm{single}_{(x,\alpha,\top)}(\Gamma)$ a pour domaine $\mathrm{dom}_\lambda(\Gamma)$, et que sa valeur en x est α, ce qui est immédiat d'après la définition 5.6.
- (Abs$_\mathrm{I}$) Immédiat en appliquant l'hypothèse d'induction à la prémisse et en appliquant (Abs$_\mathrm{S}$).
- (App$_\mathrm{I}$) En appliquant l'hypothèse d'induction à chacune des prémisses, nous obtenons $\Gamma \vdash_\mathrm{S} e_1 : A_1 \Rightarrow \tau_1 \mid C_1$ et $\Gamma \vdash_\mathrm{S} e_2 : A_2 \Rightarrow \tau_2 \mid C_2$. Par concision, définissons $A = A_1 \sqcap A_2$ et $C = (C_1 \cup C_2) + (\tau_1 \leq \tau_2 \to \alpha)$. Il est alors aisé de vérifier que

$$\begin{array}{l} A_1 \Rightarrow \tau_1 \mid C_1 \leq^m A \Rightarrow \tau_2 \to \alpha \mid C \\ A_2 \Rightarrow \tau_2 \mid C_2 \leq^m A \Rightarrow \tau_2 \mid C \end{array}$$

  Cela nous permet d'appliquer deux fois (Sub$_\mathrm{S}$) :

$$\frac{\Gamma \vdash_\mathrm{S} e_1 : A_1 \Rightarrow \tau_1 \mid C_1}{\Gamma \vdash_\mathrm{S} e_1 : A \Rightarrow \tau_2 \to \alpha \mid C}\ (\textsc{Sub}_\mathrm{S})$$

$$\frac{\Gamma \vdash_\mathrm{S} e_2 : A_2 \Rightarrow \tau_2 \mid C_2}{\Gamma \vdash_\mathrm{S} e_2 : A \Rightarrow \tau_2 \mid C}\ (\textsc{Sub}_\mathrm{S})$$

  Nous pouvons alors conclure par

$$\frac{\Gamma \vdash_\mathrm{S} e_1 : A \Rightarrow \tau_2 \to \alpha \mid C \qquad \Gamma \vdash_\mathrm{S} e_2 : A \Rightarrow \tau_2 \mid C}{\Gamma \vdash_\mathrm{S} e_1\, e_2 : A \Rightarrow \alpha \mid C}\ (\textsc{App}_\mathrm{S})$$

- (LetVar$_\mathrm{I}$) Immédiat.
- (Let$_\mathrm{I}$) Similaire au cas (App$_\mathrm{I}$). □

Avant de démontrer le théorème de complétude, nous devons introduire quelques lemmes techniques. Le premier formalise l'utilisation de F et F' pour la gestion des variables fraîches.

Lemme 5.2 *Si* $[F]\ \Gamma \vdash_\mathrm{I} e : [F']\ \sigma$, *alors* $\mathrm{fv}(\sigma) \subset F' \setminus F$.

Démonstration. Induction aisée sur la structure de la dérivation hypothèse. □

Le lemme suivant établit que si une expression est typable dans un certain environnement, alors elle est *a fortiori* typable dans tout environnement plus fin.

Lemme 5.3 *Si* $\Gamma \vdash e : \sigma$ *et* $\Gamma' \leq^\forall \Gamma$, *alors* $\Gamma' \vdash e : \sigma$.

Démonstration. (Précisons qu'il s'agit ici d'une comparaison définie point par point, entre environnements de même domaine.) Par induction sur la structure de la dérivation hypothèse. Détaillons le cas

– (LetVar) Nous avons $\Gamma(X) = \sigma$. Puisque $\Gamma' \leq^\forall \Gamma$, ceci implique $\Gamma'(X) = \sigma'$ où $\sigma' \leq^\forall \sigma$. Nous pouvons écrire

$$\frac{\Gamma'(X) = \sigma'}{\Gamma' \vdash X : \sigma'} \ (\textsc{LetVar})$$

et conclure par

$$\frac{\Gamma' \vdash X : \sigma' \qquad \sigma' \leq^\forall \sigma}{\Gamma' \vdash X : \sigma} \ (\textsc{Sub})$$

Les autres cas sont immédiats. □

Nous pouvons maintenant établir le théorème principal :

Théorème 5.3 *Les règles d'inférence sont complètes vis-à-vis des règles de typage. C'est-à-dire, si*

$$\Gamma \vdash e : \sigma$$

alors, pour tout $F \subset \mathcal{V}$ fini, il existe un $F' \subset \mathcal{V}$ fini et un schéma $\sigma' \leq^\forall \sigma$ tels que

$$[F]\,\Gamma \vdash_{\mathrm{I}} e : [F']\,\sigma'$$

Démonstration. Par induction sur la structure de la dérivation hypothèse. Il y a un cas par règle de typage. Chacun des cas reprend exactement les notations de la figure 5.1, ce qui nous permet d'omettre l'hypothèse.

– (Var) Soit $\alpha \notin F$; soient $F' = F \cup \{\alpha\}$, $A' = \mathrm{single}_{(x,\alpha,\top)}(\Gamma)$ et $\sigma' = A' \Rightarrow \alpha \mid \varnothing$. Alors

$$\frac{\alpha \notin F \qquad A' = \mathrm{single}_{(x,\alpha,\top)}(\Gamma)}{[F]\,\Gamma \vdash_{\mathrm{I}} x : [F']\,\sigma'} \ (\textsc{Var}_{\mathrm{I}})$$

Il reste à vérifier que $\sigma' \leq^\forall A \Rightarrow A(x) \mid C$. D'après la définition de la comparaison entre schémas, et d'après la définition 5.6, cette assertion est équivalente à : pour toute solution ρ de C, il existe une substitution brute ρ', de domaine $\{\alpha\}$, telle que $\rho(A(x)) \leq \rho'(\alpha)$ et $\rho'(\alpha) \leq \rho(A(x))$. Assertion vérifiée, le témoin étant $\rho' : \alpha \mapsto \rho(A(x))$.

– (Abs) En appliquant l'hypothèse d'induction à la prémisse, nous obtenons

$$[F]\,\mathrm{lift}_x(\Gamma); x \vdash_{\mathrm{I}} e : [F']\,\sigma_0$$

où $\sigma_0 \leq^\forall (A; x : \tau) \Rightarrow \tau' \mid C$. σ_0 est donc nécessairement de la forme $(A_0; x : \tau_0) \Rightarrow \tau'_0 \mid C_0$. Nous pouvons écrire

$$\frac{[F]\,\mathrm{lift}_x(\Gamma); x \vdash_{\mathrm{I}} e : [F']\,(A_0; x : \tau_0) \Rightarrow \tau'_0 \mid C_0}{[F]\,\Gamma \vdash_{\mathrm{I}} \lambda x.e : [F']\,A_0 \Rightarrow \tau_0 \to \tau'_0 \mid C_0} \ (\textsc{Abs}_{\mathrm{I}})$$

Il reste à vérifier que

$$A_0 \Rightarrow \tau_0 \to \tau'_0 \mid C_0 \leq^\forall A \Rightarrow \tau \to \tau' \mid C$$

qui découle aisément de notre hypothèse

$$(A_0; x : \tau_0) \Rightarrow \tau'_0 \mid C_0 \leq^\forall (A; x : \tau) \Rightarrow \tau' \mid C$$

- (App) L'hypothèse d'induction, appliquée à la première prémisse, donne $[F]\,\Gamma \vdash_{\mathrm{I}} e_1 : [F']\,\sigma'_1$ où $\sigma'_1 \leq^{\forall} A \Rightarrow \tau_2 \rightarrow \tau \mid C$. En appliquant l'hypothèse d'induction à la seconde prémisse (cette fois avec F' en entrée, au lieu de F), on obtient $[F']\,\Gamma \vdash_{\mathrm{I}} e_2 : [F'']\,\sigma'_2$ où $\sigma'_2 \leq^{\forall} A \Rightarrow \tau_2 \mid C$. Pour $i \in \{1, 2\}$, le schéma σ'_i est de la forme $A'_i \Rightarrow \tau'_i \mid C'_i$. Soit $\alpha \notin F''$. Nous pouvons écrire

$$\frac{[F]\,\Gamma \vdash_{\mathrm{I}} e_1 : [F']\,\sigma'_1 \qquad [F']\,\Gamma \vdash_{\mathrm{I}} e_2 : [F'']\,\sigma'_2 \qquad \alpha \notin F''}{[F]\,\Gamma \vdash_{\mathrm{I}} e_1\, e_2 : [F'' \cup \{\alpha\}]\, A' \Rightarrow \alpha \mid C'} \;(\textsc{App}_{\mathrm{I}})$$

où $A' = A'_1 \sqcap A'_2$ et $C' = (C'_1 \cup C'_2) + (\tau'_1 \leq \tau'_2 \rightarrow \alpha)$. Il reste à vérifier que

$$A' \Rightarrow \alpha \mid C' \leq^{\forall} A \Rightarrow \tau \mid C$$

Soit ρ une solution de C. D'après les deux assertions de comparaison de schémas établies ci-dessus, il existe des solutions ρ'_i de C'_i, pour $i \in \{1, 2\}$, telles que

$$\rho'_1(A'_1 \Rightarrow \tau'_1) \leq \rho(A \Rightarrow \tau_2 \rightarrow \tau)$$
$$\rho'_2(A'_2 \Rightarrow \tau'_2) \leq \rho(A \Rightarrow \tau_2)$$

A présent, deux applications successives du lemme 5.2 montrent que $\mathrm{fv}(\sigma_1)$, $\mathrm{fv}(\sigma_2)$ et $\{\alpha\}$ sont disjoints deux à deux. Par conséquent, il existe une substitution ρ' qui coïncide avec ρ'_i sur $\mathrm{fv}(\sigma_i)$ pour $i \in \{1, 2\}$ et dont la valeur en α est $\rho(\tau)$. Nous allons maintenant vérifier que ρ' est le témoin désiré, c'est-à-dire que ρ' est solution de C' et que

$$\rho'(A' \Rightarrow \alpha) \leq \rho(A \Rightarrow \tau)$$

Pour $i \in \{1, 2\}$, ρ' satisfait C'_i car elle coïncide avec ρ'_i sur $\mathrm{fv}(\sigma_i)$. De plus, nous avons $\rho'_1(\tau'_1) \leq \rho(\tau_2 \rightarrow \tau)$ et $\rho'_2(\tau'_2) \leq \rho(\tau_2)$. En combinant ces inéquations, nous obtenons $\rho'_1(\tau'_1) \leq \rho'_2(\tau'_2) \rightarrow \rho(\tau)$. Puisque $\rho(\tau) = \rho'(\alpha)$, ceci n'est autre que $\rho'(\tau'_1) \leq \rho'(\tau'_2 \rightarrow \alpha)$, ce qui signifie que ρ' satisfait $\tau'_1 \leq \tau'_2 \rightarrow \alpha$. Nous avons vérifié que ρ' est une solution de C'.
Nous avons $\rho(A) \leq \rho'_1(A'_1)$ et $\rho(A) \leq \rho'_2(A'_2)$. Ceci peut être réécrit en $\rho(A) \leq \rho'(A'_1)$ et $\rho(A) \leq \rho'(A'_2)$, qui impliquent $\rho(A) \leq \rho'(A')$, puisque $A' = A'_1 \sqcap A'_2$. Finalement, $\rho'(\alpha) \leq \rho(\tau)$ est satisfait par définition de ρ'.
- (LetVar) Nous avons $\Gamma(X) = \sigma$. Soit ρ un renommage de σ tel que $\mathrm{im}(\rho) \cap F = \varnothing$. Alors

$$\frac{\Gamma(X) = \sigma \qquad \rho \text{ renommage de } \sigma \qquad \mathrm{im}(\rho) \cap F = \varnothing}{[F]\,\Gamma \vdash_{\mathrm{I}} X : [F \cup \mathrm{im}(\rho)]\,\rho(\sigma)} \;(\textsc{LetVar}_{\mathrm{I}})$$

Il reste à vérifier que $\rho(\sigma) \leq^{\forall} \sigma$, ce qui est immédiat puisque l'α-conversion est contenue dans la relation d'équivalence entre schémas de types.
- (Let) En appliquant l'hypothèse d'induction à la première prémisse, nous obtenons $[F]\,\Gamma \vdash_{\mathrm{I}} e_1 : [F']\,\sigma'_1$, où $\sigma'_1 \leq^{\forall} \sigma_1$. En appliquant le lemme 5.3 à la seconde prémisse, nous obtenons $\Gamma; X : \sigma'_1 \vdash e_2 : A_2 \Rightarrow \tau_2 \mid C_2$. En appliquant l'hypothèse d'induction à ce jugement, on obtient $[F']\,\Gamma; X : \sigma'_1 \vdash_{\mathrm{I}} e_2 : [F'']\,\sigma'_2$, où $\sigma'_2 \leq^{\forall} A_2 \Rightarrow \tau_2 \mid C_2$. Nous pouvons alors écrire

$$\frac{[F]\,\Gamma \vdash_{\mathrm{I}} e_1 : [F']\,\sigma'_1 \qquad [F']\,\Gamma; X : \sigma'_1 \vdash_{\mathrm{I}} e_2 : [F'']\,\sigma'_2}{[F]\,\Gamma \vdash_{\mathrm{I}} \texttt{let}\, X = e_1 \;\texttt{in}\; e_2 : [F'']\,(A'_1 \sqcap A'_2) \Rightarrow \tau'_2 \mid C'_1 \cup C'_2} \;(\textsc{Let}_{\mathrm{I}})$$

où $\sigma'_i = (A'_i \Rightarrow \tau'_i \mid C'_i)$ pour $i \in \{1, 2\}$. Il reste à vérifier que

$$(A'_1 \sqcap A'_2) \Rightarrow \tau'_2 \mid C'_1 \cup C'_2 \leq^{\forall} A_2 \Rightarrow \tau_2 \mid C_2$$

De $\sigma'_1 \leq^{\forall} \sigma_1$ et de la troisième prémisse, on déduit, par transitivité de $\leq^{\forall}$, que $\sigma'_1 \leq^{\forall} A_2 \Rightarrow \top \mid C_2$. Par ailleurs, rappelons que $\sigma'_2 \leq^{\forall} A_2 \Rightarrow \tau_2 \mid C_2$. Le résultat en découle aisément (modulo, comme dans le cas (App), une application du lemme 5.2).

– (Sub) Une application de l'hypothèse d'induction donne $[F]\ \Gamma \vdash_{\mathrm{I}} e : [F']\ \sigma''$, où $\sigma'' \leq^{\forall} \sigma$. Par ailleurs, nous avons $\sigma \leq^{\forall} \sigma'$, d'où $\sigma'' \leq^{\forall} \sigma'$ par transitivité de $\leq^{\forall}$. □

Le théorème de complétude indique que si un programme est typable et admet un typage σ, alors il existe une dérivation, écrite à l'aide des règles d'inférence, qui fournit un typage plus fin. Cet énoncé peut être renforcé en remarquant que les règles d'inférence sont déterministes, c'est-à-dire que deux dérivations d'inférence quelconques (pour le même programme) fournissent des schémas de types équivalents :

Lemme 5.4 *Si* $[F_1]\ \Gamma \vdash_{\mathrm{I}} e : [F_1']\ \sigma_1$ *et* $[F_2]\ \Gamma \vdash_{\mathrm{I}} e : [F_2']\ \sigma_2$*, alors* $\sigma_1 =^{\forall} \sigma_2$*.*

Démonstration. Laissée au lecteur. □

Voici alors l'énoncé amélioré :

Théorème 5.4 *Soient* Γ *un environnement et* e *une expression. Alors,*

- *soit* e *n'est pas typable dans* Γ*. Alors il n'existe aucune dérivation d'inférence pour* e *dans* Γ*.*
- *soit* e *est typable dans* Γ*. Alors il existe une dérivation d'inférence pour* e *dans* Γ *; de plus, toute dérivation de la sorte produit un schéma de types optimal pour* e *dans* Γ*, au sens de la relation* $\leq^{\forall}$*.*

Démonstration. Si e n'est pas typable, alors il ne peut exister aucune dérivation d'inférence, sans quoi la correction des règles d'inférence vis-à-vis des règles de typage serait contredite (celle-ci s'obtient par combinaison des théorèmes 5.1 et 5.2).

Sinon, nous avons $\Gamma \vdash e : \sigma$ pour un certain schéma de types σ. D'après le théorème 5.3, il existe une dérivation d'inférence

$$[\varnothing]\ \Gamma \vdash_{\mathrm{I}} e : [F']\ \sigma'$$

où $\sigma' \leq^{\forall} \sigma$.

Soit σ'' tel que $\Gamma \vdash e : \sigma''$. Alors, on peut appliquer le raisonnement ci-dessus à σ''. Mais, d'après le lemme 5.4, toutes les dérivations d'inférence pour e dans Γ produisent des schémas de types équivalents à σ' ; il en découle que $\sigma' \leq^{\forall} \sigma''$. Par conséquent, σ' est un schéma de types optimal pour e dans Γ. □

Ce dernier théorème indique que les règles d'inférence décrivent directement un algorithme d'inférence, dirigé par la syntaxe. L'algorithme rejette le programme si celui-ci n'est pas typable, et produit un typage optimal dans le cas contraire.

5.7 Correction du typage

Il nous reste à montrer que le système de typage est sain, c'est-à-dire qu'un programme bien typé ne peut provoquer d'erreur à l'exécution. Dans un premier temps, nous définissons formellement la notion d'exécution en fournissant une sémantique opérationnelle de notre langage. Ensuite, nous démontrons que la réduction préserve le typage, et nous en déduisons la correction du système de typage.

5.7.1 Sémantique opérationnelle

Une sémantique opérationnelle à petits pas consiste simplement en une description syntaxique de l'exécution des programmes, c'est-à-dire un ensemble de règles de réécriture sur les programmes. Les deux règles fondamentales sont la réduction des rédex β et `let`. On y ajoute une règle de réduction sous contexte, laquelle spécifie la stratégie d'évaluation ; nous utilisons ici une stratégie par valeur.

Définition 5.9 *Les* valeurs *forment un sous-ensemble des expressions irréductibles, défini par*

$$v ::= \lambda x.e$$

Notre langage étant réduit au λ-calcul pur, on notera que les expressions irréductibles sans variables libres coïncident ici avec les valeurs. Cela n'est pas vrai dans des langages plus riches ; par exemple, si l'on ajoute des constantes entières au langage, alors l'expression $2\,(\lambda x.x)$ est irréductible, mais n'est pas une valeur ; elle correspond à une erreur d'exécution. Dans notre langage restreint, aucune erreur d'exécution n'est possible. Cependant, cela n'affecte en rien la preuve de stabilité du typage par réduction ; aussi n'avons-nous pas jugé nécessaire d'enrichir le langage. L'ajout de constantes au langage sera brièvement évoqué lors de la preuve de correction du typage (cf. théorème 5.6).

Définition 5.10 *La* substitution sans capture *d'une variable x par une expression e' dans une expression e, notée $[e'/x]\,e$, est définie par*

$$\begin{aligned}
[e'/x]\,x &= e' \\
[e'/x]\,y &= y \\
[e'/x]\,X &= X \\
[e'/x]\,\lambda x.e &= \lambda x.e \\
[e'/x]\,\lambda y.e &= \lambda y.[e'/x]\,e \\
[e'/x]\,(e_1\,e_2) &= ([e'/x]\,e_1)\,([e'/x]\,e_2) \\
[e'/x]\,(\mathtt{let}\,X = e_1\ \mathtt{in}\ e_2) &= \mathtt{let}\,X = [e'/x]\,e_1\ \mathtt{in}\ [e'/x]\,e_2
\end{aligned}$$

Similairement, la substitution sans capture d'une variable X par une expression e' dans une expression e, notée $[e'/X]\,e$, est définie par

$$\begin{aligned}
[e'/X]\,x &= x \\
[e'/X]\,X &= e' \\
[e'/X]\,Y &= Y \\
[e'/X]\,\lambda x.e &= \lambda x.[e'/X]\,e \\
[e'/X]\,(e_1\,e_2) &= ([e'/X]\,e_1)\,([e'/X]\,e_2) \\
[e'/X]\,(\mathtt{let}\,X = e_1\ \mathtt{in}\ e_2) &= \mathtt{let}\,X = [e'/X]\,e_1\ \mathtt{in}\ e_2 \\
[e'/X]\,(\mathtt{let}\,Y = e_1\ \mathtt{in}\ e_2) &= \mathtt{let}\,Y = [e'/X]\,e_1\ \mathtt{in}\ [e'/X]\,e_2
\end{aligned}$$

Définition 5.11 *On se munit des* règles de réduction *fondamentales suivantes :*

$$\begin{aligned}
(\lambda x.e)\,v &\longrightarrow [v/x]\,e \\
\mathtt{let}\,X = v\ \mathtt{in}\ e &\longrightarrow [v/X]\,e
\end{aligned}$$

auxquelles on ajoute une règle dite de réduction sous contexte *:*

$$e \longrightarrow e' \Rightarrow E[e] \longrightarrow E[e']$$

Les contextes de réduction *sont définis de façon à refléter la stratégie d'évaluation par valeur :*

$$E ::= [] \mid E\,e \mid v\,E \mid \mathtt{let}\,X = E\ \mathtt{in}\ e$$

5.7.2 Stabilité du typage par réduction

L'objet de cette section est de démontrer que le typage est conservé par réduction. Il s'agit d'une section essentiellement technique. Nous conclurons et commenterons le résultat dans la section suivante.

Enonçons d'abord un lemme de clôture très simple, qui indique que le typage des termes clos est indépendant de l'environnement.

Lemme 5.5 *Soit e un terme clos (c'est-à-dire sans variables libres). Soient Γ, A, τ, C tels que* $\mathrm{dom}(A) = \mathrm{dom}_\lambda(\Gamma)$. *Alors*

$$\Gamma \vdash_S e : A \Rightarrow \tau \mid C \iff \varnothing \vdash_S e : \varnothing \Rightarrow \tau \mid C$$

Nous allons maintenant établir un lemme concernant la stabilité du typage par une étape de β-réduction. Du fait que nous nous intéressons au système de typage « simple », doté d'une règle de sous-typage affaiblie, ce résultat est relativement aisé. Il est d'ailleurs classique ; on en trouve l'analogue dans [16], quoique sans démonstration.

Lemme 5.6 *Soient e une expression et v une valeur telles que $\Gamma \vdash_S e : A \Rightarrow \tau \mid C$ et $\varnothing \vdash_S v : \varnothing \Rightarrow \tau_{\mathrm{in}} \mid C_{\mathrm{in}}$. Soit $x \in \mathrm{dom}(A)$. On suppose que $C \Vdash C_{\mathrm{in}} + (\tau_{\mathrm{in}} \leq A(x))$. Alors*

$$\mathrm{unlift}_x(\Gamma) \setminus x \vdash_S [v/x]\, e : (A \setminus x) \Rightarrow \tau \mid C$$

Démonstration. Par induction sur la structure de la dérivation de typage de e. Il y a donc un cas par règle de typage « simple ». Par brièveté, posons $\Gamma' = \mathrm{unlift}_x(\Gamma) \setminus x$.

- ($\textsc{Var}_S$) L'expression est une variable. Deux cas se présentent, selon que cette variable est x ou non.
  - $e = x$. Alors $\tau = A(x)$ et $[v/x]\, e = v$. D'après le lemme 5.5, nous avons

    $$\Gamma' \vdash_S v : (A \setminus x) \Rightarrow \tau_{\mathrm{in}} \mid C_{\mathrm{in}}$$

    d'où l'on peut déduire, par ($\textsc{Sub}_S$),

    $$\Gamma' \vdash_S v : (A \setminus x) \Rightarrow \tau \mid C$$

  - $e = y \neq x$. Alors $\tau = A(y) = (A \setminus x)(y)$, donc la règle ($\textsc{Var}_S$) donne $\Gamma' \vdash_S y : (A \setminus x) \Rightarrow \tau \mid C$.
- ($\textsc{Abs}_S$) L'expression e est de la forme $\lambda y.a$, et sa dérivation de typage se termine par

  $$\frac{\mathrm{lift}_y(\Gamma); y \vdash_S a : (A; y : \tau_0) \Rightarrow \tau_1 \mid C}{\Gamma \vdash_S \lambda y.a : A \Rightarrow \tau \mid C}\ (\textsc{Abs}_S)$$

  où $\tau = \tau_0 \rightarrow \tau_1$. Deux cas se présentent, selon que y est ou non égal à x.
  - $y = x$. La variable x apparaît alors deux fois dans l'environnement $\mathrm{lift}_x(\Gamma); x$, dans lequel on a typé la prémisse. La première occurrence est donc inutile. Un lemme de capture (proche parent du lemme 5.5) nous permet de réécrire cette prémisse

    $$\mathrm{lift}_x(\mathrm{unlift}_x(\Gamma) \setminus x); x \vdash_S a : (A \setminus x; x : \tau_0) \Rightarrow \tau_1 \mid C$$

    d'où l'on peut déduire, grâce à la règle ($\textsc{Abs}_S$),

    $$\Gamma' \vdash_S \lambda x.a : (A \setminus x) \Rightarrow \tau \mid C$$

  - $y \neq x$. L'application de l'hypothèse d'induction à la prémisse donne

    $$\mathrm{unlift}_x(\mathrm{lift}_y(\Gamma); y) \setminus x \vdash_S [v/x]\, a : ((A; y : \tau_0) \setminus x) \Rightarrow \tau_1 \mid C$$

    Légèrement réécrite, cette assertion permet d'appliquer à nouveau ($\textsc{Abs}_S$) :

    $$\frac{\mathrm{lift}_y(\Gamma'); y \vdash_S [v/x]\, a : ((A \setminus x); y : \tau_0) \Rightarrow \tau_1 \mid C}{\Gamma' \vdash_S [v/x]\, e : (A \setminus x) \Rightarrow \tau \mid C}\ (\textsc{Abs}_S)$$

– ($\textsc{App}_\mathrm{S}$) L'expression e est de la forme $e_1\, e_2$, et sa dérivation de typage se termine par

$$\frac{\Gamma \vdash_\mathrm{S} e_1 : A \Rightarrow \tau_2 \to \tau \mid C \qquad \Gamma \vdash_\mathrm{S} e_2 : A \Rightarrow \tau_2 \mid C}{\Gamma \vdash_\mathrm{S} e : A \Rightarrow \tau \mid C}\ (\textsc{App}_\mathrm{S})$$

Appliquons l'hypothèse de récurrence à chaque prémisse, et nous pouvons écrire

$$\frac{\begin{array}{c}\Gamma' \vdash_\mathrm{S} [v/x]\, e_1 : (A \setminus x) \Rightarrow \tau_2 \to \tau \mid C \\ \Gamma' \vdash_\mathrm{S} [v/x]\, e_2 : (A \setminus x) \Rightarrow \tau_2 \mid C\end{array}}{\Gamma' \vdash_\mathrm{S} [v/x]\, e : (A \setminus x) \Rightarrow \tau \mid C}\ (\textsc{App}_\mathrm{S})$$

– ($\textsc{LetVar}_\mathrm{S}$) e est une variable X. Le schéma $A \Rightarrow \tau \mid C$ s'écrit alors $\rho(\Gamma(X))$ pour un certain renommage ρ. Par définition de Γ', on en déduit que $\rho(\Gamma'(X))$ est précisément $(A \setminus x) \Rightarrow \tau \mid C$. On peut donc utiliser ($\textsc{LetVar}_\mathrm{S}$) pour obtenir $\Gamma' \vdash_\mathrm{S} X : (A \setminus x) \Rightarrow \tau \mid C$.

– ($\textsc{Let}_\mathrm{S}$) L'expression e est de la forme $\mathtt{let}\, X = e_1 \,\mathtt{in}\ e_2$, et sa dérivation de typage se termine par

$$\frac{\begin{array}{c}\Gamma \vdash_\mathrm{S} e_1 : \sigma_1 \qquad \Gamma; X : \sigma_1 \vdash_\mathrm{S} e_2 : \sigma_2 \\ \sigma_1 \leq^m A \Rightarrow \top \mid C\end{array}}{\Gamma \vdash_\mathrm{S} e : \sigma_2}\ (\textsc{Let}_\mathrm{S})$$

où $\sigma_2 = A \Rightarrow \tau \mid C$. Notons $\sigma_1 = A_1 \Rightarrow \tau_1 \mid C_1$. De $\sigma_1 \leq^m A \Rightarrow \top \mid C$, on déduit $C \Vdash A(x) \leq A_1(x)$. En conjonction avec $C \Vdash C_\mathrm{in} + (\tau_\mathrm{in} \leq A(x))$, cela implique $C \Vdash C_\mathrm{in} + (\tau_\mathrm{in} \leq A_1(x))$. Nous pouvons donc appliquer l'hypothèse d'induction à la première prémisse, ce qui donne

$$\Gamma' \vdash_\mathrm{S} [v/x]\, e_1 : (A_1 \setminus x) \Rightarrow \tau_1 \mid C_1$$

Appliquons à présent l'hypothèse d'induction à la seconde prémisse. Nous obtenons

$$\mathrm{unlift}_x(\Gamma; X : \sigma_1) \setminus x \vdash_\mathrm{S} [v/x]\, e_2 : (A \setminus x) \Rightarrow \tau \mid C$$

Notons que l'environnement utilisé dans cette assertion n'est autre que $\Gamma'; X : (A_1 \setminus x) \Rightarrow \tau_1 \mid C_1$. Nous pouvons à présent conclure :

$$\frac{\begin{array}{c}\Gamma' \vdash_\mathrm{S} [v/x]\, e_1 : (A_1 \setminus x) \Rightarrow \tau_1 \mid C_1 \\ \Gamma'; X : (A_1 \setminus x) \Rightarrow \tau_1 \mid C_1 \vdash_\mathrm{S} [v/x]\, e_2 : (A \setminus x) \Rightarrow \tau \mid C \\ (A_1 \setminus x) \Rightarrow \tau_1 \mid C_1 \leq^m (A \setminus x) \Rightarrow \top \mid C\end{array}}{\Gamma' \vdash_\mathrm{S} [v/x]\, e : (A \setminus x) \Rightarrow \tau \mid C}\ (\textsc{Let}_\mathrm{S})$$

– ($\textsc{Sub}_\mathrm{S}$) La dérivation de typage de e se termine par

$$\frac{\Gamma \vdash_\mathrm{S} e : A' \Rightarrow \tau' \mid C' \qquad A' \Rightarrow \tau' \mid C' \leq^m A \Rightarrow \tau \mid C}{\Gamma \vdash_\mathrm{S} e : A \Rightarrow \tau \mid C}\ (\textsc{Sub}_\mathrm{S})$$

La deuxième prémisse, en conjonction avec l'hypothèse $C \Vdash C_\mathrm{in} + (\tau_\mathrm{in} \leq A(x))$, implique $C \Vdash C_\mathrm{in} + (\tau_\mathrm{in} \leq A'(x))$. Nous pouvons donc appliquer l'hypothèse de récurrence à la première prémisse. Nous concluons alors comme suit :

$$\frac{\begin{array}{c}\Gamma' \vdash_\mathrm{S} [v/x]\, e : (A' \setminus x) \Rightarrow \tau' \mid C' \\ (A' \setminus x) \Rightarrow \tau' \mid C' \leq^m (A \setminus x) \Rightarrow \tau \mid C\end{array}}{\Gamma' \vdash_\mathrm{S} [v/x]\, e : (A \setminus x) \Rightarrow \tau \mid C}\ (\textsc{Sub}_\mathrm{S})$$

Ceci conclut le lemme de β-réduction. $\square$

Nous désirons maintenant établir un résultat similaire vis-à-vis de la `let`-réduction. Commençons par établir un lemme de renommage des variables de types. L'énoncé est similaire à celui que l'on trouve dans la théorie de ML, mais la démonstration en est quasi triviale, grâce à notre traitement plus simple de la quantification.

(Notons que ce lemme est immédiat dans le cas des règles de typage, puisque la comparaison polymorphe entre schémas de types contient les renommages. Mais nous nous intéressons ici aux règles de typage dites « simples », lesquelles sont moins puissantes ; une démonstration par induction est alors nécessaire.)

Lemme 5.7 *On suppose* $\Gamma \vdash_S e : \sigma$. *Soit* ρ *un renommage de* Γ *et de* σ. *Alors*

$$\rho(\Gamma) \vdash_S e : \rho(\sigma)$$

Démonstration. Par induction sur la structure de la dérivation hypothèse. Il y a donc un cas par règle de typage « simple ».

- ($\textsc{Var}_S$) L'expression e est une variable x, et σ est de la forme $A \Rightarrow A(x) \mid C$. Une application de ($\textsc{Var}_S$) montre que $\rho(\Gamma) \vdash_S e : \rho(A) \Rightarrow \rho(A(x)) \mid \rho(C)$ est un jugement valide.
- ($\textsc{LetVar}_S$) L'expression e est une variable X, et σ est de la forme $\psi(\Gamma(X))$ où ψ est un renommage. Alors le schéma $\rho(\sigma)$ peut s'écrire $\rho\psi\rho^{-1}(\rho\Gamma(X))$. $\rho\psi\rho^{-1}$ étant un renommage de $\rho\Gamma(X)$, la règle ($\textsc{LetVar}_S$) s'applique et donne $\rho(\Gamma) \vdash_S e : \rho(\sigma)$.
- ($\textsc{Let}_S$) Ce cas se traite par simple application de l'hypothèse d'induction. Nous le détaillons cependant pour souligner la différence avec le cas du langage ML.

  La dérivation de typage de e se termine par une règle de la forme

  $$\frac{\Gamma \vdash_S e_1 : \sigma_1 \qquad \Gamma; X : \sigma_1 \vdash_S e_2 : \sigma \qquad \sigma_1 \leq^m A \Rightarrow \top \mid C}{\Gamma \vdash_S e : \sigma} \ (\textsc{Let}_S)$$

  où $\sigma = A \Rightarrow \tau \mid C$. Il se peut que ρ ne soit pas défini sur $\mathrm{fv}(\sigma_1)$; on l'étend alors de façon à en faire un renommage défini sur $\mathrm{fv}(\sigma_1)$. Appliquons ensuite l'hypothèse d'induction à la première prémisse ; nous obtenons

  $$\rho(\Gamma) \vdash_S e_1 : \rho(\sigma_1)$$

  Faisons subir le même traitement à la seconde prémisse :

  $$\rho(\Gamma); X : \rho(\sigma_1) \vdash_S e_2 : \rho(\sigma)$$

  Enfin, de la troisième prémisse, on déduit aisément

  $$\rho(\sigma_1) \leq^m \rho(A \Rightarrow \top \mid C)$$

  Ces trois assertions établies, nous pouvons appliquer ($\textsc{Let}_S$), et nous obtenons le résultat attendu : $\rho(\Gamma) \vdash_S e : \rho(\sigma)$.
- Les cas ($\textsc{Abs}_S$), ($\textsc{App}_S$) et ($\textsc{Sub}_S$) s'obtiennent, de façon similaire, par application de l'hypothèse d'induction. □

Nous pouvons maintenant établir la stabilité du typage par `let`-réduction :

Lemme 5.8 *Soient* e *une expression et* v *une valeur telles que* $\Gamma_1; X : \sigma_X; \Gamma_2 \vdash_S e : \sigma$ *et* $\varnothing \vdash_S v : \varnothing \Rightarrow \tau_X \mid C_X$, *où* $\sigma_X = A_X \Rightarrow \tau_X \mid C_X$ *et* $X \notin \mathrm{dom}(\Gamma_2)$. *Alors*

$$\Gamma_1; \Gamma_2 \vdash_S [v/X]\, e : \sigma$$

Démonstration. Par induction sur la dérivation de typage de e. Les cas non triviaux sont :

– ($\textsc{LetVar}_{\mathrm{S}}$) Deux cas se présentent, selon que l'expression e est précisément X ou non. Seul le premier est digne d'intérêt ; supposons donc $e = X$. Alors $\sigma = \rho(\sigma_X)$ pour un certain renommage ρ. Par ailleurs, $[v/X]\,e$ est égal à v. Par application du lemme 5.7 à notre deuxième hypothèse, nous obtenons $\varnothing \vdash_{\mathrm{S}} v : \varnothing \Rightarrow \rho(\tau_X) \mid \rho(C_X)$. Le lemme 5.5 donne alors $\Gamma_1; \Gamma_2 \vdash_{\mathrm{S}} v : \rho(A_X) \Rightarrow \rho(\tau_X) \mid \rho(C_X)$ qui est précisément le résultat attendu.
– ($\textsc{Let}_{\mathrm{S}}$) L'expression e est de la forme $\texttt{let}\, Y = e_1 \texttt{ in } e_2$. Sa dérivation de typage se termine par une règle de la forme

$$\frac{\Gamma_1; X : \sigma_X; \Gamma_2 \vdash_{\mathrm{S}} e_1 : \sigma_1 \qquad \Gamma_1; X : \sigma_X; \Gamma_2; Y : \sigma_1 \vdash_{\mathrm{S}} e_2 : \sigma \qquad \sigma_1 \leq^m A \Rightarrow \top \mid C}{\Gamma_1; X : \sigma_X; \Gamma_2 \vdash_{\mathrm{S}} e : \sigma}\ (\textsc{Let}_{\mathrm{S}})$$

où $\sigma = A \Rightarrow \tau \mid C$. Par application de l'hypothèse d'induction à la première prémisse, on obtient $\Gamma_1; \Gamma_2 \vdash_{\mathrm{S}} [v/X]\,e_1 : \sigma_1$. Il faut à présent distinguer deux cas :
  – $Y = X$. Alors l'environnement qui apparaît dans la seconde prémisse ci-dessus contient deux occurrences de X. D'après un lemme de capture similaire au lemme 5.5), la première occurrence est superflue, et l'on a $\Gamma_1; \Gamma_2; X : \sigma_1 \vdash_{\mathrm{S}} e_2 : \sigma$. On peut alors conclure, en appliquant ($\textsc{Let}_{\mathrm{S}}$) :

$$\Gamma_1; \Gamma_2 \vdash_{\mathrm{S}} \texttt{let}\, X = [v/X]\,e_1 \texttt{ in } e_2 : \sigma$$

  – $Y \neq X$. On peut alors appliquer l'hypothèse d'induction à la seconde prémisse, ce qui donne $\Gamma_1; \Gamma_2; Y : \sigma_1 \vdash_{\mathrm{S}} [v/X]\,e_2 : \sigma$. On peut alors appliquer ($\textsc{Let}_{\mathrm{S}}$), d'où

$$\Gamma_1; \Gamma_2 \vdash_{\mathrm{S}} \texttt{let}\, Y = [v/X]\,e_1 \texttt{ in } [v/X]\,e_2 : \sigma$$

Ceci conclut le lemme de `let`-réduction. □

Munis des lemmes précédents, nous sommes à présent en mesure de démontrer la stabilité du typage « simple » par réduction.

Lemme 5.9 *Si $\Gamma \vdash_{\mathrm{S}} e : \sigma$ et $e \longrightarrow e'$, alors $\Gamma \vdash_{\mathrm{S}} e' : \sigma$.*

Démonstration. Par induction sur la structure de la dérivation de l'assertion $e \longrightarrow e'$. Dans chacun des cas ci-dessous, on supposera que la dérivation de typage de e ne se termine pas par la règle ($\textsc{Sub}_{\mathrm{S}}$), l'extension au cas général étant ensuite immédiate.
– $e = (\lambda x.a)\,v$ et $e' = [v/x]\,a$. On a mentionné que la dérivation de typage de e ne se termine pas par la règle ($\textsc{Sub}_{\mathrm{S}}$). Elle est donc nécessairement de la forme

$$\frac{\Gamma \vdash_{\mathrm{S}} \lambda x.a : A \Rightarrow \tau_0 \rightarrow \tau \mid C \qquad \Gamma \vdash_{\mathrm{S}} v : A \Rightarrow \tau_0 \mid C}{\Gamma \vdash_{\mathrm{S}} e : A \Rightarrow \tau \mid C}\ (\textsc{App}_{\mathrm{S}})$$

où $\sigma = A \Rightarrow \tau \mid C$. De surcroît, la règle ($\textsc{Sub}_{\mathrm{S}}$) commute librement avec la règle ($\textsc{Abs}_{\mathrm{S}}$) ; on peut donc supposer, quitte à la réécrire, que la dérivation de typage de $\lambda x.a$ est de la forme

$$\frac{\mathrm{lift}_x(\Gamma); x \vdash_{\mathrm{S}} a : (A; x : \tau_0) \Rightarrow \tau \mid C}{\Gamma \vdash_{\mathrm{S}} \lambda x.a : A \Rightarrow \tau_0 \rightarrow \tau \mid C}\ (\textsc{Abs}_{\mathrm{S}})$$

v étant un terme clos, le lemme 5.5 permet d'écrire $\varnothing \vdash_{\mathrm{S}} v : \varnothing \Rightarrow \tau_0 \mid C$. Enfin, notons que $C \Vdash C + (\tau_0 \leq (A; x : \tau_0)(x))$ – il s'agit d'une tautologie. Nous pouvons donc appliquer le lemme 5.6. Nous obtenons

$$\mathrm{unlift}_x(\mathrm{lift}_x(\Gamma); x) \setminus x \vdash_{\mathrm{S}} [v/x]\,a : ((A; x : \tau_0) \setminus x) \Rightarrow \tau \mid C$$

que l'on peut réécrire

$$\Gamma \vdash_{\mathrm{S}} e' : A \Rightarrow \tau \mid C$$

- $e = \texttt{let}\, X = v\, \texttt{in}\; a$ et $e' = [v/X]\, a$. La dérivation de typage de e est nécessairement de la forme
$$\frac{\Gamma \vdash_S v : \sigma_1 \qquad \Gamma; X : \sigma_1 \vdash_S a : \sigma \qquad \sigma_1 \leq^m A \Rightarrow \top \mid C}{\Gamma \vdash_S e : \sigma} \; (\textsc{Let}_S)$$
où $\sigma = A \Rightarrow \tau \mid C$. Notons $\sigma_1 = A_1 \Rightarrow \tau_1 \mid C_1$. D'après le lemme 5.5, nous avons $\varnothing \vdash_S v : \varnothing \Rightarrow \tau_1 \mid C_1$. Nous pouvons alors appliquer le lemme 5.8, et nous obtenons
$$\Gamma \vdash_S [v/X]\, a : \sigma$$
- $e = E[e_0]$, $e' = E[e'_0]$ et $e_0 \longrightarrow e'_0$, où E est un contexte de réduction. L'arbre de typage de e contient un jugement de typage de e_0. En appliquant l'hypothèse d'induction à celui-ci, on le remplace par un jugement identique concernant e'_0. L'arbre de typage complet fournit alors une dérivation de $\Gamma \vdash_S E[e'_0] : \sigma$. □

Finalement, voici le résultat principal de cette section : la stabilité du typage par réduction, dans le cas des règles de typage originales.

Théorème 5.5 *Si $\Gamma \vdash e : \sigma$ et $e \longrightarrow e'$, alors $\Gamma \vdash e' : \sigma$.*

Démonstration. Supposons $\Gamma \vdash e : \sigma$. La combinaison des théorèmes 5.3 et 5.2 indique que les règles de typages « simples » sont complètes vis-à-vis des règles de typage. Par conséquent, il existe un schéma de types σ' tel que $\sigma' \leq^\forall \sigma$ et $\Gamma \vdash_S e : \sigma'$. Le lemme 5.9 indique alors que $\Gamma \vdash_S e' : \sigma'$. Les règles de typage « simples » étant correctes par rapport aux règles de typage (théorème 5.1), ceci implique $\Gamma \vdash e' : \sigma'$. Par application de la règle (Sub), nous avons alors $\Gamma \vdash e' : \sigma$. □

5.7.3 Correction du typage vis-à-vis de la sémantique

Nous avons montré, dans la section précédente, que le typage est préservé par la réduction. C'est sur ce théorème fondamental que repose la correction du typage vis-à-vis de la sémantique. Avant de conclure, il ne reste plus qu'à démontrer la proposition suivante :

Proposition 5.10 *Soient une expression e irréductible et bien typée dans l'environnement vide, c'est-à-dire telle que $\varnothing \vdash e : \sigma$. Alors e est une valeur.*

Démonstration. Par induction sur la structure du terme e.

- Si e est une variable, alors e ne peut être bien typée dans l'environnement vide ; cas éliminé.
- Si e est une λ-abstraction, alors e est une valeur.
- Si e est une application $e_1\, e_2$, alors e_1 est elle-même irréductible, car $[]\, e_2$ est un contexte de réduction (définition 5.11). D'après l'hypothèse d'induction, e_1 est une valeur. On en déduit que $e_1\, []$ est également un contexte de réduction, et que, de même, e_2 est une valeur. Toute valeur étant une λ-abstraction (définition 5.9), e est un β-rédex. e n'est donc pas irréductible ; cas éliminé.
- Si e est de la forme $\texttt{let}\, X = e_1\, \texttt{in}\; e_2$, alors on montre comme précédemment que e_1 est une valeur. Par conséquent, e est un `let`-rédex, donc n'est pas irréductible ; cas éliminé.

La preuve est terminée. En fait, ce résultat est trivial, car le calcul considéré est le λ-calcul pur, dans lequel toute expression close irréductible est une valeur. En d'autres termes, il n'y a pas d'erreur d'exécution possible en λ-calcul pur. Cette proposition devient non triviale si l'on étend le calcul et le système de types considérés. Par exemple, voici ce que serait la démonstration après l'introduction de constantes entières et d'un type de base `int`. (On n'ajoute ici aucune primitive de manipulation des entiers. L'intérêt pratique de cette extension est donc limité, mais elle suffit à introduire la possibilité d'erreur.)

Un nouveau cas apparaît dans la preuve, celui où e est une constante. Les constantes étant des valeurs, le résultat est immédiat. Les autres cas sont inchangés, à l'exception de celui où e est une application. Alors, on ne peut plus affirmer que e_1 est une λ-abstraction ; il faut également examiner le cas où e_1 est une constante. Puisque l'application est bien typée, cette constante a reçu, dans l'environnement vide, un schéma de types de la forme $\varnothing \Rightarrow \tau_0 \to \tau_1 \mid C$. Or, toute dérivation de typage pour une constante entière est nécessairement constituée d'une application de la règle de typage des constantes (nouvellement introduite), éventuellement suivie par une ou plusieurs applications de la règle (Sub). Par conséquent, nous avons

$$\varnothing \Rightarrow \texttt{int} \mid \varnothing \leq^{\forall} \varnothing \Rightarrow \tau_0 \to \tau_1 \mid C$$

Rappelons qu'un programme n'est considéré comme bien typé qu'à la condition (implicite) que le schéma de types qui lui est associé admette une instance. Dans le cas présent, C admet donc une solution ρ. Par définition de la comparaison polymorphe entre schémas de types, nous avons alors

$$\texttt{int} \leq \rho(\tau_0) \to \rho(\tau_1)$$

Or ceci est impossible ; la relation de sous-typage a été définie de sorte que les constructeurs `int` et $\to$ soient incompatibles. Ce cas est donc éliminé. □

Nous pouvons à présent affirmer que notre système de types est sûr :

Théorème 5.6 *Soit e une expression bien typée dans l'environnement vide, c'est-à-dire telle que $\varnothing \vdash e : \sigma$. Supposons qu'une évaluation de e termine, c'est-à-dire que $e \longrightarrow^* e'$ où e' est une expression irréductible. Alors e' est une valeur et a le même type que e, c'est-à-dire que $\varnothing \vdash e' : \sigma$.*

Démonstration. L'assertion $\varnothing \vdash e' : \sigma$ provient du théorème 5.5. e' est donc irréductible et bien typée dans l'environnement vide. D'après la proposition 5.10, e' est une valeur (par opposition à une expression irréductible car erronée, qui constituerait une erreur d'exécution). □

Deuxième partie

Simplification

Chapitre 6

L'invariant des petits termes

Ce court chapitre a pour objet de montrer qu'il est possible, sans perte de puissance, d'imposer des restrictions assez fortes sur la forme des schémas de types que nous manipulons. L'adoption de ces restrictions nous permettra ensuite de simplifier de façon significative notre exposé théorique, et d'améliorer l'efficacité de notre implémentation.

L'idée de se restreindre à des petits termes n'est pas nouvelle, au moins du point de vue théorique. On la trouve, par exemple, dans la théorie de l'unification [31], où la présentation du problème sous forme de multi-équations de profondeur 1 au plus, plutôt que d'équations entre termes arbitraires, procède du même principe. Parmi les travaux plus proches du nôtre, ceux d'Aiken et Wimmers [5] et de Palsberg [39] utilisent une convention analogue. Cependant, il s'agit souvent d'une simple convenance théorique. Ici, l'invariant permettra de plus de renforcer le partage entre nœuds, par l'intermédiaire de l'algorithme de minimisation, et est donc utile du point de vue de l'implémentation.

Nous commençons par définir ce que nous appellerons l'invariant des petits termes (section 6.1). Puis, nous indiquons comment le faire respecter à faible coût (section 6.2). De plus, la section 6.3 décrit un algorithme qui rend tout schéma de types conforme à l'invariant. Enfin, en section 6.4, nous étudions les bénéfices retirés de ce choix.

6.1 Définition

Commençons par définir ces restrictions. Dans ce qui suit, nous nous intéressons à la forme des types utilisés ; nous parlerons donc de *termes*, plutôt que de types.

Définition 6.1 *Un terme est appelé* terme feuille *ssi sa hauteur est* 0 *et il n'est pas construit. De façon équivalente, ssi il s'obtient par combinaison, à l'aide de* $\sqcup$ *ou* $\sqcap$*, d'une ou plusieurs variables de type.*

Un terme est appelé petit terme *ssi sa hauteur est au plus* 1*, il est construit, et ses sous-termes sont des termes feuilles.*

Définition 6.2 *Un graphe de contraintes* C *vérifie l'*invariant des petits termes *ssi pour tout* $\alpha \in \mathrm{dom}(C)$*,* $C^{\downarrow}(\alpha)$ *et* $C^{\uparrow}(\alpha)$ *sont des petits termes.*

Un schéma de types $A \Rightarrow \tau \mid C$ *vérifie l'invariant des petits termes ssi*

- *tous les types apparaissant dans* A *sont des termes feuilles ;*
- τ *est un terme feuille ;*
- C *vérifie l'invariant des petits termes.*

6.2 Mise en œuvre

Il est très aisé de faire en sorte que tous les schémas de types produits par nos règles d'inférence vérifient cet invariant. Un examen des règles montre que seules deux d'entre elles sont susceptibles de le briser : (VAR$_\mathrm{I}$) et (ABS$_\mathrm{I}$). La première introduit un contexte A associant le type $\top$ à toutes les variables autres que x ; il nous suffit de remplacer $\top$ par une variable fraîche β. La seconde crée un type construit à l'aide du constructeur $\rightarrow$; il nous suffit de le déplacer vers le graphe de contraintes à l'aide d'une nouvelle contrainte.

$$\frac{\alpha, \beta \notin F \qquad A = \mathrm{single}_{(x,\alpha,\beta)}(\Gamma)}{[F]\ \Gamma \vdash_\mathrm{I} x : [F \cup \{\alpha, \beta\}]\ A \Rightarrow \alpha \mid \varnothing} \qquad (\textsc{Var}_\mathrm{I})$$

$$\frac{[F]\ \mathrm{lift}_x(\Gamma); x \vdash_\mathrm{I} e : [F']\ (A; x : \tau) \Rightarrow \tau' \mid C \qquad \alpha \notin F'}{[F]\ \Gamma \vdash_\mathrm{I} \lambda x.e : [F' \cup \{\alpha\}]\ A \Rightarrow \alpha \mid C + (\tau \rightarrow \tau' \leq \alpha)} \qquad (\textsc{Abs}_\mathrm{I})$$

FIG. 6.1: Règles d'inférence, respectant l'invariant des petits termes

Les règles modifiées sont données par la figure 6.1. Dans chaque cas, on vérifie aisément que le schéma de types produit par la nouvelle version de la règle est équivalent (au sens de la comparaison polymorphe) à celui produit par l'ancienne. Toutes les propriétés du système d'inférence sont donc conservées.

Théorème 6.1 *Si $[F]\ \Gamma \vdash_\mathrm{I} e : [F']\ \sigma$ est dérivable dans le système d'inférence de types modifié, alors σ vérifie l'invariant des petits termes.*

Démonstration. Par induction sur la structure de la dérivation. La preuve est aisée, et laissée au lecteur. □

6.3 Normalisation explicite

Il est donc possible de faire respecter l'invariant des petits termes sans pour autant devoir y consacrer une phase dédiée ; il suffit d'utiliser les règles d'inférence modifiées. Néanmoins, nous allons développer un algorithme de « normalisation », qui à tout schéma de types associe un schéma équivalent et respectant l'invariant. Du point de vue théorique, l'existence de cet algorithme confirme que nos restrictions sur la forme des schémas de types ne provoquent pas de perte d'expressivité. Du point de vue pratique, cet algorithme sera utilisé pour normaliser les schémas de types fournis par l'utilisateur, par exemple dans les signatures de modules.

Théorème 6.2 *Soit σ un schéma de types. Alors il existe un schéma de types σ', qui vérifie l'invariant des petits termes, tel que $\sigma =^\forall \sigma'$.*

Démonstration. La preuve est entièrement triviale ; le seul point remarquable est qu'il suffit d'ajouter des inéquations (par opposition à des équations) au graphe de contraintes. Un terme apparaissant en position positive (resp. négative) dans le graphe de contraintes est remplacé par une variable fraîche dont il devient la borne inférieure (resp. supérieure).

Considérons une paire formée d'un pos-type τ et d'un graphe de contraintes C. Son *pos-masquage* est la paire (α, C'), où $\alpha \notin \mathrm{dom}(C)$ et $C' = C + (\tau \leq \alpha)$. Son *neg-masquage* est défini de façon symétrique. Notons que cette opération produit un terme feuille.

Considérons une paire formée d'un pos-type construit τ et d'un graphe de contraintes C. Son *pos-découpage* est la paire

- (τ, C) si τ est de hauteur 0 ;

– $(\tau_0' \to \tau_1', C'')$ si $\tau = \tau_0 \to \tau_1$, (τ_0', C') est le neg-masquage de (τ_0, C), et (τ_1', C'') est le pos-masquage de (τ_1, C').

Son *neg-découpage* est défini de façon symétrique. Notons que cette opération produit un petit terme.

On définit la relation $\rightsquigarrow$ entre schémas de types de la façon suivante. Soit $\sigma = A \Rightarrow \tau \mid C$ un schéma de types.

1. Si τ n'est pas un terme feuille, soit (τ', C') le pos-masquage de (τ, C) ; alors $\sigma \rightsquigarrow A \Rightarrow \tau' \mid C'$.
2. Si A contient une entrée de la forme $x : \tau_x$ où τ_x n'est pas un terme feuille, soit (τ_x', C') le neg-masquage de (τ_x, C) ; soit A' identique à A, excepté que l'entrée $x : \tau_x$ est remplacée par $x : \tau_x'$; alors $\sigma \rightsquigarrow A' \Rightarrow \tau \mid C'$.
3. Si, pour un certain $\alpha \in \mathrm{dom}(C)$, $C^{\downarrow}(\alpha)$ n'est pas un petit terme, soit (τ', C') le pos-découpage de $(C^{\downarrow}(\alpha), C)$; soit C'' identique à C' à l'exception du fait que $C''^{\downarrow}(\alpha) = \tau'$. Alors $\sigma \rightsquigarrow A \Rightarrow \tau \mid C''$.
4. Si, pour un certain $\alpha \in \mathrm{dom}(C)$, $C^{\uparrow}(\alpha)$ n'est pas un petit terme, soit (τ', C') le neg-découpage de $(C^{\uparrow}(\alpha), C)$; soit C'' identique à C' à l'exception du fait que $C''^{\uparrow}(\alpha) = \tau'$. Alors $\sigma \rightsquigarrow A \Rightarrow \tau \mid C''$.

Nous pouvons à présent vérifier que si $\sigma \rightsquigarrow \sigma'$, alors $\sigma =^{\forall} \sigma'$. Il faut traiter chacun des cas ci-dessus.

1. Soit ρ' une solution de C'. Alors ρ' est également solution de C, car C' contient C. Nous prétendons que ρ' est un témoin de l'assertion $\sigma \leq^{\forall} \sigma'$. Il suffit de vérifier que $\rho'(\tau) \leq \rho'(\tau')$, ce qui est vrai car C' contient la contrainte $\tau \leq \tau'$.

   Réciproquement, soit ρ une solution de C. Soit ρ' l'extension de ρ qui à τ' associe $\rho(\tau)$ (τ' est une variable de types). Alors ρ' est solution de C', car la contrainte $\tau \leq \tau'$ est satisfaite par égalité. ρ' est témoin de l'assertion $\sigma' \leq^{\forall} \sigma$, car $\rho'(\tau') \leq \rho'(\tau)$ est également vérifiée.
2. Les autres cas sont basés sur le même principe, et laissés au lecteur.

On vérifie aisément qu'étant donné un schéma de types σ, toute séquence de réécriture selon $\rightsquigarrow$ ayant pour origine σ est finie. En effet, l'étape 1 ne peut être appliquée qu'une fois, l'étape 2 une fois par élément de A ; quant aux étapes 3 et 4, elles font décroître strictement le nombre de sous-termes qui violent l'invariant dans le graphe de contraintes.

Enfin, tout schéma de types σ' en forme normale vis-à-vis de $\rightsquigarrow$ vérifie l'invariant des petits termes. L'algorithme de normalisation consiste donc à appliquer $\rightsquigarrow$ autant de fois que possible. $\square$

Exemple. Considérons le schéma de types

$$(\varnothing; x : \alpha \to \beta) \Rightarrow \alpha \to \top$$

Il ne respecte pas l'invariant des petits termes, car des termes construits apparaissent dans son contexte et dans son corps. L'algorithme décrit ci-dessus remplace ces termes par des variables fraîches, tout en ajoutant les contraintes appropriées. On notera que le terme $\alpha \to \top$ n'est pas un petit terme, et doit donc lui-même être décomposé. On obtient ainsi

$$(\varnothing; x : \gamma) \Rightarrow \delta \mid \{\gamma \leq \alpha \to \beta,\ \alpha \to \epsilon \leq \delta,\ \top \leq \epsilon\}$$

6.4 Conséquences

Le bénéfice le plus évident de l'invariant des petits termes est la simplification qu'il apporte au niveau de la théorie. La structure des schémas de types étant à présent fort simple, l'exposé, la preuve et l'implémentation de plusieurs algorithmes (clôture, comparaison polymorphe de schémas, canonisation, minimisation) seront nettement facilités.

Les algorithmes de clôture (cf. chapitre 7) et de comparaison polymorphe de schémas (cf. chapitre 9) voient leur définition (en ce qui concerne le second) et leur preuve simplifiée. Fondamentalement, ils sont néanmoins indépendants de cet invariant.

L'algorithme de canonisation (cf. chapitre 11) est chargé d'éliminer toute occurrence des constructeurs $\sqcup$ et $\sqcap$. Grâce à l'invariant des petits termes, si un tel constructeur apparaît, alors ses arguments ne peuvent être que des variables de type. Le cas « hétérogène » où l'un des arguments est un type construit et les autres sont des variables est donc éliminé, ce qui simplifie théorie et implémentation de l'algorithme.

L'algorithme de minimisation (cf. chapitre 13), quant à lui, serait probablement nettement plus difficile à énoncer en l'absence de cette hypothèse. Cet algorithme cherche à détecter dans le graphe de contraintes des sous-termes, ou nœuds, jouant des rôles identiques, de façon à pouvoir les identifier. Or, l'adoption de l'invariant des petits termes revient essentiellement à étiqueter chaque nœud du graphe par une variable de types, comme le montre la façon dont fonctionne l'algorithme de normalisation exposé plus haut. Dans ces conditions, l'algorithme de minimisation peut raisonner simplement en termes de variables : relation d'équivalence entre variables, polarité d'une variable, identification de deux variables, etc. En l'absence de l'invariant, il faudrait étendre ces notions aux nœuds, ce qui serait redondant et, pour certaines, difficile. L'invariant fournit donc en fait un moyen simple d'exprimer le *partage* entre nœuds, et c'est là son principal intérêt. Le gain n'est pas uniquement théorique ; il est également pratique, puisque l'algorithme de minimisation sera ainsi à même d'identifier des nœuds arbitraires, et donc de réaliser un meilleur partage.

L'idée de manipuler uniquement des petits termes n'est pas nouvelle. On la trouve, en particulier, dans la théorie de l'unification [31]. Plutôt que d'utiliser des équations entre termes arbitraires, on y présente les problèmes sous forme d'ensembles de multi-équations dont tous les membres sont des variables, excepté éventuellement un terme construit, de profondeur 1 au plus. Les bénéfices sont les mêmes, à savoir simplification de la théorie et meilleures possibilités de partage. Parmi les travaux plus proches du nôtre, on retrouve une technique analogue chez Aiken et Wimmers [5] et chez Palsberg [39].

Dorénavant, nous supposerons donc que tous les schémas de types sur lesquels nous travaillons vérifient l'invariant des petits termes. Cela signifie que les nouvelles transformations que nous introduirons devront préserver cet invariant. Cela sera toujours le cas, car aucune d'entre elles ne *construira* de nouveaux types.

Notons que l'adoption de cet invariant interdit la simplification classique qui consiste à remplacer une variable par son unique borne [15, 4, 8, 43], dans le cas où celle-ci est un terme construit. En effet, cette substitution brise l'invariant, puisqu'elle fait disparaître une variable, créant ainsi des nœuds non étiquetés. Nous en déduisons que cette simplification est contraire à notre objectif d'efficacité, et nous la réservons donc à l'affichage. Nous reviendrons sur ce point dans la section 15.2.

Chapitre 7

Décision de la solubilité de contraintes

Voici à présent la définition, et la preuve, de l'algorithme de clôture. Celui-ci permet de décider si un graphe de contraintes donné admet ou non une solution. Il est basé sur la remarque que tout graphe de contraintes soluble est équivalent à un graphe clos. Le problème est donc, étant donné un graphe quelconque, de le compléter pour obtenir un graphe clos. Si ce procédé échoue, alors le graphe considéré est insoluble ; sinon, il admet une solution, que l'on peut déduire de sa forme close, d'après le théorème 3.1.

L'algorithme présente l'intérêt d'être *incrémental*, c'est-à-dire qu'il demande un graphe clos C et une nouvelle contrainte c, et produit (si possible) un graphe clos équivalent à $C + c$. Ainsi, dans une implémentation, les contraintes peuvent être ajoutées au fur et à mesure qu'elles sont engendrées par le moteur d'inférence de types, sans devoir à chaque étape reconstruire la clôture en partant de zéro.

Nous aurions pu présenter cet algorithme au chapitre 3, puisqu'il constitue la suite logique du théorème 3.1. Nous l'avons déplacé ici parce que sa preuve utilise l'invariant des petits termes, invariant qui n'était pas encore introduit alors. Nous supposerons, dans ce chapitre, que les graphes de contraintes manipulés vérifient l'invariant des petits termes.

Nous commençons par quelques lemmes préliminaires (section 7.1). Puis, nous définissons l'algorithme (section 7.2) et en prouvons la correction (section 7.3).

7.1 Préliminaires

Ces lemmes utilisent l'invariant des petits termes et permettront de simplifier l'énoncé de l'invariant de l'algorithme, ainsi que sa preuve.

Lemme 7.1 *Soient $(\tau_i)_{i\in I}$ (resp. $(\tau'_j)_{j\in J}$) une famille de petits pos-types (resp. neg-types). Alors*

$$\mathrm{subc}(\sqcup_{i\in I}\tau_i \leq \sqcap_{j\in J}\tau'_j) = \bigcup_{(i,j)\in I\times J} \mathrm{subc}(\tau_i \leq \tau'_j)$$

Démonstration. (Par cette égalité, on entend que ces deux expressions sont définies en même temps, et égales lorsqu'elles sont définies.) Le résultat est immédiat dans le cas des termes feuilles. On le généralise ensuite aux petits termes, ce qui demande une analyse par cas laborieuse mais sans difficulté. □

De ce résultat, on déduit en particulier :

Lemme 7.2 *Soient τ_1 et τ'_1 deux petits pos-types tels que τ'_1 contient τ_1. Soit τ_2 un neg-type. Alors, l'assertion*

$$\mathrm{subc}(\tau_1 \leq \tau_2) \subset \mathrm{subc}(\tau'_1 \leq \tau_2)$$

est vérifiée pour peu que son membre droit soit défini.

Démonstration. Supposons $\mathrm{subc}(\tau_1' \leq \tau_2)$ défini. D'après la définition 2.11, nous avons $\tau_1' = \tau_1 \sqcup \tau_1'$. Par conséquent, d'après le lemme 7.1, $\mathrm{subc}(\tau_1' \leq \tau_2)$ est égal à $\mathrm{subc}(\tau_1 \leq \tau_2) \cup \mathrm{subc}(\tau_1' \leq \tau_2)$. En d'autres termes, $\mathrm{subc}(\tau_1 \leq \tau_2) \subset \mathrm{subc}(\tau_1' \leq \tau_2)$ qui est le résultat attendu. □

Il existe bien sûr un lemme symétrique au précédent, dont nous ne donnerons pas l'énoncé par souci de brièveté.

7.2 Algorithme

Voici à présent la définition de l'algorithme de clôture incrémental.

Définition 7.1 *Un* état *de l'algorithme est une paire (C, Q), où C est un graphe de contraintes et Q un ensemble de contraintes de la forme $\alpha \leq \beta$, $\alpha \leq \tau$ ou $\tau \leq \beta$, où α, β sont des variables et τ est un petit terme. De plus, on demande que $\mathrm{fv}(Q) \subset \mathrm{dom}(C)$.*

L'algorithme passe *d'un état (C, Q) à un nouvel état comme suit. Si la liste d'attente Q est vide, on arrête l'algorithme, avec pour résultat C. Sinon, on choisit une contrainte c dans Q et on pose $Q' = Q \setminus \{c\}$. Trois cas se présentent alors, selon la forme de c.*

- *c est de la forme $\alpha \leq \beta$. Si $\alpha \leq_C \beta$, on passe à l'état (C, Q'). Sinon, on passe à l'état (D, Q''), où D et Q'' sont définis de la façon suivante :*
  - $\leq_D = \leq_C \cup \{(\alpha', \beta')\,;\, \alpha' \leq_C \alpha \wedge \beta \leq_C \beta'\}$;
  - *Si $\beta \leq_C \beta'$, alors $D^{\downarrow}(\beta') = C^{\downarrow}(\beta') \sqcup C^{\downarrow}(\alpha)$, sinon $D^{\downarrow}(\beta') = C^{\downarrow}(\beta')$;*
  - *Si $\alpha' \leq_C \alpha$, alors $D^{\uparrow}(\alpha') = C^{\uparrow}(\alpha') \sqcap C^{\uparrow}(\beta)$, sinon $D^{\uparrow}(\alpha') = C^{\uparrow}(\alpha')$;*
  - $Q'' = Q' \cup \mathrm{subc}(C^{\downarrow}(\alpha) \leq C^{\uparrow}(\beta))$.
- *c est de la forme $\alpha \leq \tau$. Si $C^{\uparrow}(\alpha)$ contient τ, on passe à l'état (C, Q'). Sinon, on passe à l'état (D, Q''), où D et Q'' sont définis de la façon suivante :*
  - $\leq_D = \leq_C$;
  - $D^{\downarrow} = C^{\downarrow}$;
  - *Si $\alpha' \leq_C \alpha$, alors $D^{\uparrow}(\alpha') = C^{\uparrow}(\alpha') \sqcap \tau$, sinon $D^{\uparrow}(\alpha') = C^{\uparrow}(\alpha')$;*
  - $Q'' = Q' \cup \mathrm{subc}(C^{\downarrow}(\alpha) \leq \tau)$.
- *c est de la forme $\tau \leq \beta$. Si $C^{\downarrow}(\beta)$ contient τ, on passe à l'état (C, Q'). Sinon, on passe à l'état (D, Q''), où D et Q'' sont définis de la façon suivante :*
  - $\leq_D = \leq_C$;
  - *Si $\beta \leq_C \beta'$, alors $D^{\downarrow}(\beta') = C^{\downarrow}(\beta') \sqcup \tau$, sinon $D^{\downarrow}(\beta') = C^{\downarrow}(\beta')$;*
  - $D^{\uparrow} = C^{\uparrow}$;
  - $Q'' = Q' \cup \mathrm{subc}(\tau \leq C^{\uparrow}(\beta))$.

Notons que l'algorithme peut échouer si la fonction subc est évaluée en dehors de son domaine. Cela signifie qu'une contrainte insoluble a été mise en évidence.

Exemple. Pour démontrer le fonctionnement de l'algorithme de clôture incrémental, calculons, à l'aide des règles d'inférence, le type de l'application $(\lambda x.x)\,3$. On supposera donné le schéma de types de la fonction identité $\lambda x.x$, qui est $\gamma \mid C_1$, où $C_1 = \varnothing + (\alpha \to \beta \leq \gamma) + (\alpha \leq \beta)$; ainsi que celui de l'expression 3, qui est $\epsilon \mid C_2$, où $C_2 = \varnothing + (\mathtt{int} \leq \epsilon)$. Pour typer l'application, il faut appliquer la règle (App$_{\mathrm{I}}$). Celle-ci indique que le schéma de types associé au résultat est $\delta \mid C$, où $C = (C_1 \cup C_2) + (\gamma \leq \epsilon \to \delta)$. L'union $C_1 \cup C_2$ ne donne lieu à aucun calcul, puisque ces deux graphes de contraintes sont de domaines disjoints. Reste à utiliser l'algorithme de clôture incrémental pour lui ajouter la contrainte $\gamma \leq \epsilon \to \delta$.

Notre description de l'algorithme est fonctionnelle, c'est-à-dire que chaque étape produit un nouveau graphe de contraintes. Pour plus de simplicité, nous rédigerons cet exemple sous forme impérative, c'est-à-dire que nous supposerons que l'algorithme effectue des modifications sur un graphe existant.

Soit donc, initialement, C le graphe défini par $\alpha \leq_C \beta$, $C^\downarrow(\gamma) = \alpha \to \beta$ et $C^\downarrow(\epsilon) = \texttt{int}$. La liste d'attente Q contient la contrainte $\gamma \leq \epsilon \to \delta$.

La première étape de l'algorithme retire donc cette contrainte de la queue. Elle met à jour la borne supérieure de γ en posant $C^\uparrow(\gamma) = \epsilon \to \delta$, puis elle ajoute à la liste d'attente les contraintes $\mathrm{subc}(\alpha \to \beta \leq \epsilon \to \delta)$, c'est-à-dire $\epsilon \leq \alpha$ et $\beta \leq \delta$, obtenues par transitivité sur γ.

La deuxième étape retire de la liste d'attente la contrainte $\epsilon \leq \alpha$. (On pourrait commencer par l'autre contrainte ; le résultat final serait le même.) Elle ajoute ses conséquences sur les variables : $\epsilon \leq_C \alpha$ et $\epsilon \leq_C \beta$. Puis, ϵ ayant une borne inférieure construite non triviale, cette borne vient contraindre α et β : on pose donc $C^\downarrow(\alpha) = C^\downarrow(\beta) = \texttt{int}$. Enfin, on doit ajouter à la liste d'attente les contraintes $\mathrm{subc}(C^\downarrow(\epsilon) \leq C^\uparrow(\alpha))$, c'est-à-dire $\mathrm{subc}(\texttt{int} \leq \top) = \varnothing$; il n'y a ici rien à ajouter.

La troisième étape retire de la liste d'attente la contrainte $\beta \leq \delta$. Elle ajoute ses conséquences sur les variables, à savoir $\epsilon \leq_C \delta$, $\alpha \leq_C \delta$ et $\beta \leq_C \delta$. β a maintenant $\texttt{int}$ pour borne inférieure construite ; aussi doit-on poser $C^\downarrow(\delta) = \texttt{int}$. Enfin, encore une fois, on constate qu'aucune contrainte n'est à ajouter à la liste d'attente. L'algorithme termine donc, et fournit un graphe C clos.

Rappelons que d'après la règle (App$_\mathrm{I}$), le schéma de types associé au λ-terme complet est $\delta \mid C$. δ est donc le type de ce terme, et on trouve dans C la contrainte $\texttt{int} \leq \delta$; cela indique que ce terme est un entier. Si, en anticipant quelque peu, on applique à ce schéma l'algorithme de dépoussiérage présenté au chapitre 10, on constate que toutes les autres contraintes sont éliminées. Elles ont donc joué un rôle intermédiaire pendant le calcul de clôture, mais ne sont plus nécessaires une fois obtenue l'information pertinente, à savoir $\texttt{int} \leq \delta$.

7.3 Correction

Proposition 7.3 *L'algorithme de clôture termine.*

Démonstration. Soit $(C_n, Q_n)_n$ une suite de transitions. On note que tous les C_n ont le même domaine V et vérifient l'invariant des petits termes. L'ensemble des graphes de contraintes de domaine V et vérifiant cet invariant est fini. On l'ordonne en posant que C est inférieur ou égal à D ssi les conditions suivantes sont vérifiées :

- $\leq_C \subset \leq_D$;
- pour tout $\alpha \in V$, $D^\downarrow(\alpha)$ contient $C^\downarrow(\alpha)$ et $D^\uparrow(\alpha)$ contient $C^\uparrow(\alpha)$.

On ordonne l'ensemble des états par ordre lexicographique, l'ordre sur les premières composantes étant celui défini ci-dessus, et l'ordre sur les deuxièmes composantes étant l'inverse de l'inclusion ensembliste. On vérifie alors aisément, d'après la définition de l'algorithme, que la suite $(C_n, Q_n)_n$ est strictement croissante. Sa première composante évolue dans un ensemble fini, et est donc constante à partir d'un certain rang. A partir de ce rang, l'ensemble Q_n décroît strictement ; donc, la suite est finie. L'algorithme termine. □

Lemme 7.4 *Si l'algorithme passe d'un état (C, Q) à un état (C', Q'), alors les solutions communes à C et à Q sont exactement les solutions communes à C' et à Q'.*

Démonstration. Simple vérification sur la définition de l'algorithme. □

Lemme 7.5 *Supposons qu'un état (C, Q) vérifie les propriétés suivantes :*

- *$\leq_C$ est transitive ;*

- *si* $\alpha \leq_C \beta$, *alors* $C^\downarrow(\beta)$ *contient* $C^\downarrow(\alpha)$ *et* $C^\uparrow(\alpha)$ *contient* $C^\uparrow(\beta)$ *;*
- *pour tout* $\gamma \in \mathrm{dom}(C)$, *l'ensemble* $\mathrm{subc}(C^\downarrow(\gamma) \leq C^\uparrow(\gamma))$ *est défini et inclus dans* $\leq_C \cup Q$.

Si, de cet état, l'algorithme passe à un nouvel état, alors celui-ci vérifie les mêmes propriétés.

Démonstration. (Pour bien comprendre la troisième propriété ci-dessus, on notera que grâce à l'invariant des petits termes, l'ensemble $\mathrm{subc}(C^\downarrow(\gamma) \leq C^\uparrow(\gamma))$ ne contient que des contraintes entre variables. Ainsi, chacune d'entre elles doit être soit inscrite dans C soit en attente dans la liste Q.)

Considérons d'abord la première de ces trois propriétés. Plaçons-nous dans le cas où la contrainte c choisie dans Q est de la forme $\alpha \leq \beta$, puisque dans les autres cas, le résultat est immédiat. Supposons $\alpha' \leq_D \beta' \leq_D \gamma'$. Quatre sous-cas se présentent, selon que chacune de ces deux relations est présente ou non dans $\leq_C$.

- $\alpha' \leq_C \beta' \leq_C \gamma'$. Le résultat découle de la transitivité de $\leq_C$ et du fait que $\leq_D$ contient $\leq_C$.
- $\alpha' \leq_C \beta'$, $\beta' \leq_C \alpha$ et $\beta \leq_C \gamma'$. Par transitivité de $\leq_C$, on a $\alpha' \leq_C \alpha$, d'où $\alpha' \leq_D \gamma'$ par définition de $\leq_D$.
- Les deux derniers sous-cas sont similaires au précédent.

Passons à la deuxième propriété. Nous ne considérerons que la première des deux assertions qu'elle contient, la seconde étant symétrique. Trois cas se présentent, selon la forme de la contrainte c choisie dans Q. Dans le premier cas, c est de la forme $\alpha \leq \beta$. Supposons $\alpha' \leq_D \beta'$. Deux sous-cas se distinguent, selon que cette relation est présente ou non dans $\leq_C$.

- $\alpha' \leq_C \beta'$. Il nous faut démontrer que $D^\downarrow(\beta') \sqcup D^\downarrow(\alpha') = D^\downarrow(\beta')$. Cela est aisé, en revenant à la définition de $D^\downarrow$ et en remarquant que si $\beta \leq_C \alpha'$, alors $\beta \leq_C \beta'$ par transitivité de $\leq_C$. (On tirera bien sûr parti de l'hypothèse que la propriété est vérifiée par C.)
- $\alpha' \leq_C \alpha$ et $\beta \leq_C \beta'$. Il nous faut démontrer que $C^\downarrow(\beta') \sqcup C^\downarrow(\alpha)$ contient $D^\downarrow(\alpha')$. D'après la définition de $D^\downarrow$, $D^\downarrow(\alpha')$ est égal soit à $C^\downarrow(\alpha')$, soit à $C^\downarrow(\alpha') \sqcup C^\downarrow(\alpha)$. Pour conclure, il suffit de vérifier que $C^\downarrow(\alpha)$ contient $C^\downarrow(\alpha')$, ce qui provient du fait que $\alpha' \leq_C \alpha$ et de notre hypothèse sur C.

Le second cas est trivial, puisque $D^\downarrow = C^\downarrow$. Le troisième cas s'obtient aisément par un raisonnement similaire à celui mené ci-dessus.

Considérons enfin la troisième propriété. Soit $\gamma \in \mathrm{dom}(D)$. Trois cas sont à distinguer, selon la forme de la contrainte c choisie dans Q. Dans le premier cas, c est de la forme $\alpha \leq \beta$. Quatre sous-cas se présentent alors, selon que les conditions $\gamma \leq_C \alpha$ et $\beta \leq_C \gamma$ sont vérifiées ou non. Ces sous-cas se traitent de façon similaire ; intéressons-nous par exemple à celui où $\gamma \leq_C \alpha$ et $\beta \not\leq_C \gamma$. Nous avons alors $D^\downarrow(\gamma) = C^\downarrow(\alpha)$ et $D^\uparrow(\gamma) = C^\uparrow(\gamma) \sqcap C^\uparrow(\beta)$. D'après le lemme 7.1, nous pouvons donc écrire

$$\mathrm{subc}(D^\downarrow(\gamma) \leq D^\uparrow(\gamma)) = \mathrm{subc}(C^\downarrow(\gamma) \leq C^\uparrow(\gamma)) \cup \mathrm{subc}(C^\downarrow(\gamma) \leq C^\uparrow(\beta))$$

Le premier des ensembles du membre droit est, d'après notre hypothèse, défini et inclus dans $\leq_C \cup Q$. Il est donc *a fortiori* inclus dans $\leq_D \cup Q''$. Considérons à présent le second. Par hypothèse, $C^\downarrow(\alpha)$ contient $C^\downarrow(\gamma)$. D'après le lemme 7.2, cela entraîne

$$\mathrm{subc}(C^\downarrow(\gamma) \leq C^\uparrow(\beta)) \subset \mathrm{subc}(C^\downarrow(\alpha) \leq C^\uparrow(\beta))$$

pour peu que ce dernier soit défini. Or celui-ci est défini ; en fait, il est inclus dans Q'', par définition de Q''. L'examen de ce sous-cas est donc terminé. Les deuxième et troisième cas se traitent de façon similaire. $\square$

Grâce aux lemmes précédents, nous pouvons établir le résultat suivant :

Théorème 7.1 *Soit C un graphe de contraintes clos. Soit c une contrainte de la forme $\alpha \leq \beta$, $\alpha \leq \tau$ ou $\tau \leq \beta$, où α, β sont des variables et τ est un petit terme, et telle que* $\mathrm{fv}(c) \subset \mathrm{dom}(C)$. *On exécute l'algorithme de clôture à partir de l'état initial $(C, \{c\})$. Alors*

- *si $C + c$ est soluble, alors l'algorithme produit un graphe de contraintes clos équivalent à $C + c$;*
- *dans le cas contraire, l'algorithme signale un échec.*

Démonstration. On vérifie d'abord que $(C, \{c\})$ est bien un état au sens de la définition 7.1. (Notons que la condition $\mathrm{fv}(c) \subset \mathrm{dom}(C)$ n'est pas restrictive en pratique, puisqu'on peut préalablement étendre le domaine de C.)

D'après la proposition 7.3, l'algorithme termine. Par conséquent, soit il fournit un résultat, soit il signale un échec.

S'il fournit un résultat, alors il a atteint un état de la forme $(D, \varnothing)$. On obtient, par application répétée du lemme 7.4, l'équivalence entre $C + c$ et D. De plus, C étant clos, l'état initial $(C, \{c\})$ vérifie l'invariant donné par le lemme 7.5. D'une application répétée de ce lemme, on déduit que D est lui-même clos. Tout graphe clos étant soluble, D est soluble, donc $C + c$ également.

Si, au contraire, l'algorithme signale un échec, alors une contrainte insoluble a été mise en évidence. Or toutes les contraintes manipulées sont logiquement impliquées par $C + c$. Par conséquent, ce dernier est également insoluble. $\square$

Nous avons donc mis au point un algorithme de clôture incrémental, qui permet d'ajouter une contrainte quelconque à un graphe clos pour obtenir un nouveau graphe clos. Cet algorithme peut bien sûr être utilisé de façon non incrémentale ; la clôture d'un ensemble, ou d'un graphe, de contraintes donné se calcule en ajoutant, une par une, toutes les contraintes qu'il contient au graphe vide. Le problème de la solubilité d'une conjonction de contraintes est donc décidable.

Nous ne démontrerons pas certaines propriétés, comme le déterminisme de l'algorithme (c'est-à-dire que son résultat ne dépend pas de l'ordre dans lequel on retire les contraintes de la queue) ou la minimalité du résultat fourni (c'est-à-dire qu'il s'agit du plus petit graphe clos contenant le graphe et la liste d'attente originaux). Ces propriétés sont vérifiées, mais ne nous seront pas utiles.

Chapitre 8

Décision de l'implication de contraintes

Nous tentons, dans ce chapitre, de mettre au point un algorithme capable de déterminer si un graphe de contraintes donné implique (au sens logique) une contrainte donnée. En d'autres termes, il s'agit d'établir une procédure de décision pour la relation d'implication de contraintes donnée par la définition 3.6.

Historiquement, nous avons présenté cet algorithme dans [43]. Il jouait alors un rôle clef dans le processus de simplification des types. En effet, le système de typage présenté dans cet article était plus proche de notre système dit « simple » (cf. section 5.4) ; en particulier, il était basé sur la relation d'implication entre contraintes. De plus, la simplification était fondée sur des heuristiques ; il était donc nécessaire de pouvoir vérifier, à l'exécution, la validité des jugements de typage produits par ces heuristiques. C'est là la raison pour laquelle nous avons développé cet algorithme. Malheureusement, celui-ci se révèle incomplet.

Nous nous étions donc intéressés à la comparaison monomorphe entre schémas de types, basée sur l'implication de contraintes. Trifonov et Smith [49] développent, de façon indépendante, un algorithme tout à fait similaire au nôtre. Puis, ils le généralisent au cas de la comparaison polymorphe entre schémas. Nous donnons une description formelle et une preuve de ce nouvel algorithme au chapitre 9. Il est également incomplet ; à l'heure actuelle, la décidabilité de ces relations de comparaisons reste un problème ouvert.

Actuellement, aucun de ces deux algorithmes n'est effectivement utilisé dans la mise en œuvre de notre typeur, puisque celle-ci ne contient plus aucune heuristique. Le second sert toujours, cependant, pour comparer le type inféré pour une expression à celui déclaré dans la signature de son module. Par ailleurs, il a une importance théorique, puisqu'il constitue l'explication fondamentale du processus de dépoussiérage, introduit au chapitre 10. Quant au premier algorithme, il conserve un intérêt, parce qu'il constitue une version simplifiée du second (donc une aide à sa compréhension), et aussi parce qu'il sert à sa définition.

Ce chapitre s'organise comme suit. La section 8.1 propose une axiomatisation de l'implication de contraintes, c'est-à-dire un système logique permettant de prouver certaines assertions d'implication. Nous en prouvons d'abord la correction (section 8.2) ; puis, nous vérifions que nous avons là la spécification d'un algorithme (section 8.4). Au cours de la section 8.5, nous montrons que cette axiomatisation est incomplète, et développons plusieurs contre-exemples. Enfin, la section 8.3 en donne une formulation simplifiée, dans le cas où les types manipulés sont simples.

8.1 Axiomatisation

Enonçons d'abord le problème. Etant donnés un graphe de contraintes C, de domaine V, et une contrainte c telle que $\mathrm{fv}(c) \in V$, a-t-on $C \Vdash c$?

Il est clair que si C n'admet aucune solution, alors la réponse est positive. D'après la théorème 7.1, ce cas est décidable ; nous pouvons donc supposer que C est un graphe de contraintes clos. (Notons que cette hypothèse de clôture est importante pour rendre l'algorithme aussi puissant que possible en pratique ; cependant, elle ne sera pas utilisée dans la preuve de correction.)

Etant données ces hypothèses, nous tentons d'axiomatiser la relation d'implication de contraintes de la façon suivante.

Définition 8.1 *La figure 8.1 page ci-contre définit un système de dérivation formel. Une dérivation dans ce système sera appelée* dérivation d'implication. *Les jugements sont de la forme $C, H \vdash \tau \leq \tau'$ où C est un graphe de contraintes clos, $(\tau, \tau') \in \mathcal{T}^- \times \mathcal{T}^+$, $H \subset \mathcal{T}^- \times \mathcal{T}^+$, et toutes les variables de type apparaissant dans le jugement appartiennent au domaine de C. On écrira $C \vdash \tau \leq \tau'$ pour signifier $C, \varnothing \vdash \tau \leq \tau'$.*

Dans les règles (Tauto) et (V-Elim), nous supposons que S et S' contiennent uniquement des variables et des types construits, comme la proposition 2.2 nous y autorise. Nous autorisons S et S' à contenir un nombre quelconque d'éléments, de façon à regrouper plusieurs cas. Enfin, $C^\uparrow$ (resp. $C^\downarrow$), qui a été défini uniquement pour les variables de type, est étendu ici aux types construits en posant $C^\uparrow(\tau) = \tau$ (resp. $C^\downarrow(\tau) = \tau$) quand τ est un type construit.

Ces règles de déduction sont extrêmement simples. Pour mieux les comprendre, on pourra commencer par considérer le cas où les constructeurs $\sqcup$ et $\sqcap$ sont proscrits. Les règles se simplifient alors pour donner le système de la figure 8.2.

Remarquons que nous généralisons ici l'algorithme de comparaison de deux types récursifs donné par Amadio et Cardelli [9]. Dans les deux cas, le problème consiste à déterminer si, sous un certain ensemble d'hypothèses, un certain τ est sous-type d'un certain τ'. Dans le cas où on compare deux types récursifs, l'ensemble d'hypothèses est simplement un système d'équations, dont l'unique solution définit les deux types. Ici, l'ensemble d'hypothèses est un graphe de contraintes, dont l'ensemble de solutions est *a priori* complexe, d'où une plus grande difficulté du problème. Néanmoins, comme nous allons le voir, nos règles restent extrêmement proches de celles d'Amadio et Cardelli.

Dans ces dernières, les jugements sont de la forme $\Sigma, \epsilon \supset \tau \leq \tau'$. Nos jugements se présentent sous une forme essentiellement identique, à savoir $C, H \vdash \tau \leq \tau'$. Les hypothèses sont représentées par l'ensemble d'équations ϵ dans le premier cas, et par le graphe de contraintes C dans le second. Quant aux paramètres Σ et H, ils jouent le rôle *d'historique* ; c'est-à-dire qu'ils servent à mémoriser les buts déjà rencontrés au cours de la dérivation, de façon à garantir la terminaison de l'algorithme.

Commentons à présent nos règles :

- La règle (Hist) permet de considérer comme acquis tout but présent dans l'historique, c'est-à-dire déjà rencontré précédemment. Cette règle permet donc à l'algorithme de raisonner par co-induction : si un ancien but réapparaît, c'est qu'il est possible de construire une dérivation infinie de ce but. On l'accepte alors immédiatement.
- La règle (Tauto) permet simplement d'utiliser une contrainte entre variables présente dans les hypothèses.
- La règle (V-Elim) entre en jeu lorsqu'une variable apparaît à profondeur nulle dans le but, comme l'indique la condition $(S \cup S') \cap \mathcal{V} \neq \varnothing$. Pour fixer les idées, considérons le cas simple où le but est de la forme $\alpha \leq \tau_0 \to \tau_1$. Le but ne peut plus subir de décomposition structurelle, puisque la variable α apparaît à profondeur nulle. La solution la plus naturelle est alors d'utiliser les hypothèses contenues dans C. Si nous pouvons prouver $C^\uparrow(\alpha) \leq \tau_0 \to \tau_1$, le but en découlera, par transitivité du sous-typage. La variable α est donc remplacée par sa borne supérieure construite. (Il serait inutile de remplacer α par une variable β telle que $\alpha \leq_C \beta$, puisque β devrait alors à son tour être remplacée par une de ses bornes supérieures.) Dans le cas général, toutes les

$$\frac{(\tau \leq \tau') \in H}{C, H \vdash \tau \leq \tau'} \quad (\textsc{Hist})$$

$$\frac{\alpha \in S \qquad \alpha' \in S' \qquad \alpha \leq_C \alpha'}{C, H \vdash \sqcap S \leq \sqcup S'} \quad (\textsc{Tauto})$$

$$\frac{\begin{array}{c}(S \cup S') \cap \mathcal{V} \neq \varnothing \\ H' = H \cup \{\sqcap S \leq \sqcup S'\} \\ C, H' \vdash \sqcap\{C^{\uparrow}(\tau)\,;\, \tau \in S\} \leq \sqcup\{C^{\downarrow}(\tau')\,;\, \tau' \in S'\}\end{array}}{C, H \vdash \sqcap S \leq \sqcup S'} \quad (\mathcal{V}\textsc{-Elim})$$

$$C, H \vdash \bot \leq \tau' \quad (\bot\textsc{-Elim})$$

$$C, H \vdash \tau \leq \top \quad (\top\textsc{-Elim})$$

$$\frac{C, H \vdash \tau'_0 \leq \tau_0 \qquad C, H \vdash \tau_1 \leq \tau'_1}{C, H \vdash \tau_0 \to \tau_1 \leq \tau'_0 \to \tau'_1} \quad (\to\textsc{-Elim})$$

FIG. 8.1: Axiomatisation de l'implication de contraintes

$$\frac{(\tau \leq \tau') \in H}{C, H \vdash \tau \leq \tau'} \quad (\textsc{Hist})$$

$$\frac{\alpha \leq_C \alpha'}{C, H \vdash \alpha \leq \alpha'} \quad (\textsc{Tauto})$$

$$\frac{\begin{array}{c}\{\tau, \tau'\} \cap \mathcal{V} \neq \varnothing \\ H' = H \cup \{\tau \leq \tau'\} \qquad C, H' \vdash C^{\uparrow}(\tau) \leq C^{\downarrow}(\tau')\end{array}}{C, H \vdash \tau \leq \tau'} \quad (\mathcal{V}\textsc{-Elim})$$

$$C, H \vdash \bot \leq \tau' \quad (\bot\textsc{-Elim})$$

$$C, H \vdash \tau \leq \top \quad (\top\textsc{-Elim})$$

$$\frac{C, H \vdash \tau'_0 \leq \tau_0 \qquad C, H \vdash \tau_1 \leq \tau'_1}{C, H \vdash \tau_0 \to \tau_1 \leq \tau'_0 \to \tau'_1} \quad (\to\textsc{-Elim})$$

FIG. 8.2: Restriction aux types simples

variables apparaissant dans le membre gauche (resp. droit) sont remplacées par leur borne supérieure (resp. inférieure) construite.

On note que, par ailleurs, la règle ajoute à l'historique le but qu'on lui a soumis. Elle est la seule à effectuer cette opération. En effet, il n'est pas nécessaire que les règles de décomposition structurelle le fassent ; cela signifie simplement que quelques étapes de décomposition supplémentaires peuvent être nécessaires avant de reconnaître un but déjà rencontré.

– Les règles ($\bot$-Elim), ($\top$-Elim) et ($\rightarrow$-Elim) sont appelées règles de décomposition structurelle, c'est-à-dire qu'elles décomposent leur but en un certain nombre de sous-buts, lorsque ses deux membres sont des types construits. (En fait, les règles ($\bot$-Elim) et ($\top$-Elim) s'appliquent également lorsque τ ou τ' n'est pas construit, mais on pourrait les restreindre à ce cas sans perte de puissance.) Si on ajoute au langage des constructeurs autres que $\bot$, $\rightarrow$ et $\top$, il faudra ajouter les règles de décomposition structurelle correspondantes.

Exemple. Soit F un opérateur de types covariant et distinct de l'identité. (Par exemple, on peut choisir $F : \tau \mapsto \epsilon \rightarrow \tau$.) Soit C le graphe de contraintes de domaine $\{\alpha, \beta\}$ défini par $C^{\uparrow}(\alpha) = F\,\alpha$ et $C^{\downarrow}(\beta) = F\,\beta$. Alors, on peut utiliser notre axiomatisation pour montrer que $C \Vdash \alpha \leq \beta$.

En effet, étant donné le but $\alpha \leq \beta$, on peut utiliser deux fois la règle (V-Elim) pour remplacer chacune des variables par sa borne construite. Le but devient donc $F\,\alpha \leq F\,\beta$. De plus, puisque cette règle ajoute sa conclusion à l'historique, la contrainte $\alpha \leq \beta$ est ajoutée à l'historique.

F étant, par hypothèse, un opérateur covariant, on peut alors appliquer un certain nombre de règles de décomposition structurelle pour obtenir à nouveau le but $\alpha \leq \beta$. (Pour reprendre notre exemple, si on a choisi $F : \tau \mapsto \epsilon \rightarrow \tau$, on applique la règle ($\rightarrow$-Elim). On obtient le but $\epsilon \leq \epsilon$, qui se résout trivialement par (Tauto), et le but attendu, à savoir $\alpha \leq \beta$.)

C'est ici qu'intervient la règle (Hist) : puisque le but $\alpha \leq \beta$ apparaît dans l'historique, il est immédiatement accepté. Sans cette règle, nous serions capables de construire une dérivation infinie de notre but, mais aucune dérivation finie. Elle permet donc de coder de façon finie une preuve infinie régulière.

Cette axiomatisation de l'implication de contraintes est donc très simple, et proche du système d'Amadio et Cardelli. Malheureusement, nous perdons la propriété de complétude, c'est-à-dire que tous les énoncés vrais ne sont pas démontrables. En deux mots, cela est dû à la trop grande simplicité de la règle (V-Elim). En effet, chez Amadio et Cardelli, remplacer une variable par sa borne n'est pas source d'incomplétude, puisque variable et borne sont reliées par une égalité. Ici, nous n'avons qu'une inégalité ; le remplacement reste correct, mais la complétude est perdue. Aucune axiomatisation complète n'est actuellement connue. Ce problème sera discuté plus en détail à la section 8.5.

8.2 Correction

Nous allons à présent vérifier que cette axiomatisation est correcte vis-à-vis de l'implication de contraintes.

Il est évidemment impossible de prouver que $C, H \vdash \tau \leq \tau'$ entraîne $C \Vdash \tau \leq \tau'$ sans aucune hypothèse sur H, à cause de la règle (Hist). Celle-ci permet de considérer comme prouvée toute assertion présente dans H ; elle est donc incorrecte en l'absence de tout invariant sur H. Plutôt que d'essayer de formuler un tel invariant, nous allons construire une preuve dans laquelle (Hist) est effectivement considérée comme dangereuse ; mais nous montrerons que son utilisation peut être retardée autant qu'on le désire.

Commençons par une définition technique :

Définition 8.2 *La* profondeur *d'une dérivation d'implication est le nombre de règles de décomposition structurelle apparaissant dans la plus courte des branches couronnées par une règle* (Hist). *(Si aucune branche n'est couronnée par* (Hist)*, la profondeur de la dérivation est* ∞*.)*

Nous pouvons maintenant établir la proposition suivante, qui constitue le cœur de la preuve de correction :

Proposition 8.1 *Soit une dérivation de* $C, H \vdash \tau \leq \tau'$ *de profondeur* k*. Alors*

$$\forall j \leq k \quad C \Vdash_j \tau \leq \tau'$$

Démonstration. Par induction sur la structure de la dérivation d'implication fournie en hypothèse. Il y a donc un cas par règle de dérivation. Dans chaque cas, nous utiliserons exactement les notations de la figure 8.1, ce qui nous permet de ne pas écrire explicitement nos hypothèses.

- (Hist) La dérivation a une profondeur 0. Puisque la relation $\leq_0$ est uniformément vraie, la relation $\Vdash_0$ l'est également ; donc $C \Vdash_0 \tau \leq \tau'$ est vérifié.
- (Tauto) La dérivation a une profondeur ∞. Soient $j \in \mathbb{N}^+ \cup \{\infty\}$ et ρ une j-solution de C. En particulier, puisque $\alpha \leq_C \alpha'$, nous avons $\rho(\alpha) \leq_j \rho(\alpha')$. Ensuite, du fait que $\alpha \in S$ et $\alpha' \in S'$, nous avons $\rho(\sqcap S) \leq \rho(\alpha)$ et $\rho(\alpha') \leq \rho(\sqcup S')$. Enfin, $\leq$ étant plus fine que $\leq_j$, et $\leq_j$ étant transitive, tout ceci implique $\rho(\sqcap S) \leq_j \rho(\sqcup S')$. ρ est donc j-solution de $\sqcap S \leq \sqcup S'$.
- (V-Elim) La profondeur k de cette dérivation coïncide avec la profondeur de sa prémisse. En appliquant l'hypothèse d'induction, nous obtenons

  $$\forall j \leq k \quad C \Vdash_j \sqcap\{C^{\uparrow}(\tau)\,;\, \tau \in S\} \leq \sqcup\{C^{\downarrow}(\tau')\,;\, \tau' \in S'\}$$

  Soit ρ une j-solution de C. Alors, d'après l'assertion ci-dessus,

  $$\rho(\sqcap\{C^{\uparrow}(\tau)\,;\, \tau \in S\}) \leq_j \rho(\sqcup\{C^{\downarrow}(\tau')\,;\, \tau' \in S'\})$$

  ou, de façon équivalente,

  $$\sqcap\{\rho(C^{\uparrow}(\tau))\,;\, \tau \in S\} \leq_j \sqcup\{\rho(C^{\downarrow}(\tau'))\,;\, \tau' \in S'\}$$

  Or, pour toute variable α, nous avons

  $$\rho(C^{\downarrow}(\alpha)) \leq_j \rho(\alpha) \leq_j \rho(C^{\uparrow}(\alpha))$$

  parce que ρ est j-solution de C. De plus, pour tout type construit τ, l'assertion

  $$\rho(C^{\downarrow}(\tau)) \leq_j \rho(\tau) \leq_j \rho(C^{\uparrow}(\tau))$$

  est trivialement vérifiée, de par notre définition de $C^{\uparrow}$ et $C^{\downarrow}$ sur les types construits (cf. définition 8.1). Donc, par transitivité, notre assertion implique

  $$\sqcap\{\rho(\tau)\,;\, \tau \in S\} \leq_j \sqcup\{\rho(\tau')\,;\, \tau' \in S'\}$$

  que l'on peut réécrire

  $$\rho(\sqcap S) \leq_j \rho(\sqcup S')$$

  donc ρ est j-solution de $\sqcap S \leq \sqcup S'$.
- ($\bot$-Elim) Cette dérivation a une profondeur infinie. Pour tout $j \in \mathbb{N}^+ \cup \{\infty\}$, il est clair que $C \Vdash_j \bot \leq \tau'$.
- ($\top$-Elim) Similaire au cas précédent.

– ($\rightarrow$-ELIM) Soit k la profondeur de cette dérivation ; soit $j \leq k$. Le cas $j = 0$ est immédiat, puisque $\Vdash_0$ est uniformément vraie ; supposons donc $j \leq 1$. La profondeur de chaque prémisse est au moins $k - 1$. Par conséquent, en appliquant l'hypothèse d'induction à chaque prémisse et en spécialisant le résultat pour $j - 1$, nous obtenons $C \Vdash_{j-1} \tau'_0 \leq \tau_0$ et $C \Vdash_{j-1} \tau_1 \leq \tau'_1$. De là, on déduit facilement $C \Vdash_j \tau_0 \rightarrow \tau_1 \leq \tau'_0 \rightarrow \tau'_1$. □

Nous avons donc démontré que si la règle (HIST) n'est pas utilisée avant la profondeur k, alors le prédicat de k-implication est satisfait. En d'autres termes, plus la règle (HIST) apparaît tard dans la dérivation, plus nous sommes proches de satisfaire le prédicat d'implication. Nous allons à présent montrer que l'utilisation de (HIST) peut être repoussée autant qu'on le désire, pourvu que l'on commence avec un historique vide.

Proposition 8.2 *Soit une dérivation de $C, \varnothing \vdash \tau \leq \tau'$ de profondeur finie. Alors, il existe une dérivation de cette même assertion de profondeur strictement supérieure.*

Démonstration. Soit D la dérivation originale, et soit k sa profondeur. Puisque k est fini, D contient au moins une branche couronnée par une règle (HIST) à la profondeur k. Considérons une telle occurrence de (HIST). Sa conclusion $\tau \leq \tau'$ provient de l'historique. Or, cet historique est $\varnothing$ au bas de la dérivation D, et il ne peut être étendu que par des applications de la règle (V-ELIM). Par conséquent, $\tau \leq \tau'$ doit apparaître, à un certain point de notre branche, en conclusion d'une règle (V-ELIM).

De façon générale, la règle (V-ELIM) dispose de quelques propriétés intéressantes :

- Sa prémisse est toujours une contrainte entre deux types construits. En effet, $C^{\downarrow}(\tau)$ (resp. $C^{\uparrow}(\tau)$) est toujours un type construit ; et toute combinaison de types construits par l'intermédiaire de $\sqcup$ (resp. $\sqcap$) est également un type construit.
- Sa conclusion met en jeu au moins une variable, grâce à la condition $(S \cup S') \cap \mathcal{V} \neq \varnothing$.
- De ces deux remarques, on déduit que la prémisse est toujours différente de la conclusion.

Revenons au cas qui nous intéresse. Notre occurrence de (V-ELIM) ne peut pas être directement précédée par l'occurrence de (HIST), parce qu'alors sa conclusion et sa prémisse seraient identiques. Par conséquent, une autre règle est utilisée au-dessus de (V-ELIM). Cette autre règle a deux types construits en conclusion, et au moins une prémisse, puisque la règle (HIST) apparaît plus haut dans la branche. Par conséquent, il s'agit nécessairement d'une application de la règle ($\rightarrow$-ELIM) (ou, plus généralement parlant, d'une règle de décomposition structurelle).

Considérons à présent le sous-arbre situé à notre occurrence de (V-ELIM). Nous pouvons remplacer l'occurrence de (HIST) par une copie de ce sous-arbre (cela peut demander d'augmenter les historiques apparaissant dans la copie, ce qui ne pose pas problème), tout en conservant une dérivation valide. Puisque ce sous-arbre contient une règle de décomposition structurelle juste après sa racine, sa profondeur est au moins 1. Donc, nous avons remplacé une branche de profondeur k par une branche de profondeur strictement supérieure, tout en conservant une dérivation valide.

La règle (HIST) ne peut avoir qu'un nombre fini d'occurrences à la profondeur k. Par conséquent, en itérant ce procédé, on obtient finalement une dérivation de profondeur strictement supérieure à k. □

Nous sommes maintenant en mesure de combiner ces deux propositions :

Théorème 8.1 *L'axiomatisation est correcte vis-à-vis de l'implication de contraintes : si $C \vdash \tau \leq \tau'$, alors $C \Vdash \tau \leq \tau'$.*

Démonstration. Considérons une dérivation de $C, \varnothing \vdash \tau \leq \tau'$. Si sa profondeur est infinie, alors nous avons $\forall k \quad C \Vdash_k \tau \leq \tau'$ d'après la proposition 8.1. Sinon, d'après la proposition 8.2, pour tout k fini, il existe une dérivation de $C, \varnothing \vdash \tau \leq \tau'$ de profondeur strictement supérieure à k. En appliquant la proposition 8.1 à cette dérivation et à k, nous obtenons $C \Vdash_k \tau \leq \tau'$.

Dans les deux cas, nous avons prouvé $C \Vdash_k \tau \leq \tau'$ pour tout k fini, qui est en fait un résultat plus fort que $C \Vdash \tau \leq \tau'$, d'après la proposition 3.3. □

8.3 Une axiomatisation spécialisée

L'axiomatisation que nous avons développée jusqu'ici accepte un graphe de contraintes quelconque en tant qu'hypothèse et une paire (neg-type, pos-type) en tant que but. C'est-à-dire que l'hypothèse et le but peuvent contenir des occurrences des constructeurs $\sqcup$ et $\sqcap$ à certaines positions. Il est possible de restreindre cette axiomatisation au cas où ces constructeurs sont entièrement interdits ; nous obtenons alors les règles données par la figure 8.2 page 83. Celles-ci sont d'une compréhension légèrement plus aisée.

Notons que cette spécialisation n'occasionne pas de perte de puissance intrinsèque, parce qu'il est possible de traduire tout problème d'implication de contraintes en un problème équivalent mais dénué de toute occurrence de $\sqcup$ et $\sqcap$. La preuve de ce fait est simple, quoique nous n'ayons pas encore développé tous les outils nécessaires. L'assertion $C \Vdash \tau \leq \tau'$ est équivalente à $\alpha \to \alpha \leq^\forall \tau \to \tau' \mid C$, qui est une assertion de comparaison polymorphe entre schémas de types. Par ailleurs, nous verrons au chapitre 11 qu'il est possible d'éliminer toute occurrence de $\sqcup$ et $\sqcap$ d'un schéma de types, tout en obtenant un schéma de types équivalent. En appliquant ce procédé au schéma de droite et en revenant en arrière, nous obtenons une nouvelle assertion d'implication de contraintes, équivalente à la première, mais ne contenant que des types simples.

Le fait d'autoriser $\sqcup$ et $\sqcap$ dans notre axiomatisation de l'implication de contraintes n'augmente donc pas réellement sa puissance. Cependant, cela ne la rend pas plus complexe ; le jeu semble donc en valoir la chandelle. On constate, au contraire, qu'autoriser ces constructions dans le cadre de l'algorithme de comparaison polymorphe (que nous définirons au chapitre 9) ne serait pas naturel, parce que cela conduirait à des contraintes mal formées comme $\alpha \leq \beta \sqcup \gamma$.

8.4 Algorithme

Il est aisé de lire notre axiomatisation de l'implication de contraintes comme un algorithme de décision. Il faut cependant vérifier que l'algorithme termine. De plus, pour des raisons d'efficacité, il est souhaitable que l'algorithme n'aie à effectuer aucun choix pendant la construction de la dérivation. Cela lui permet d'abandonner dès qu'un échec est détecté ; dans le cas contraire, il faudrait effectuer un retour en arrière. Ces deux propriétés sont vérifiées ci-dessous.

Proposition 8.3 *Il existe un algorithme qui, étant donné un but de la forme $C, H \vdash \tau \leq \tau'$, donne une dérivation de ce but s'il en existe une, et signale un échec dans le cas contraire.*

Démonstration. L'algorithme fonctionne en construisant une dérivation pour le but fourni. On vérifie qu'aucun but n'a une forme telle que deux règles « récursives » lui soient applicables. En effet, (V-ELIM) demande qu'une variable apparaisse dans le but, tandis que ($\to$-ELIM) (et, plus généralement parlant, toute règle de décomposition structurelle) demande deux types construits. Par conséquent, aucun retour en arrière ne sera nécessaire.

Par ailleurs, il va de soi que lorsque plusieurs règles s'appliquent au but courant, l'algorithme choisira une règle non récursive – en particulier, la règle (HIST) – de préférence à la règle récursive. Cela est en effet nécessaire pour garantir la terminaison.

Vérifions à présent que le processus termine. S'il ne termine pas, l'algorithme exhibe donc une branche de dérivation infinie ; montrons que cela est impossible. Considérons donc une telle branche. Elle ne peut consister que d'occurrences de (V-ELIM) et de règles de décomposition structurelle, puisque ce sont les seules règles récursives. Elle ne peut pas consister uniquement de règles de décomposition, puisque celles-ci diminuent strictement la taille du but. Par conséquent, un nombre infini de règles (V-ELIM) apparaissent dans la branche.

Les règles de dérivation sont telles que dans toute dérivation de $C, H \vdash \tau \leq \tau'$, les opérandes de tout but sont des combinaisons par $\sqcup$ ou $\sqcap$ de termes apparaissant dans C, τ ou τ'. Ces termes sont en nombre fini, donc toute dérivation contient un nombre fini de buts.

En particulier, notre famille infinie d'occurrences de (V-ELIM) ne possède qu'un nombre fini de conclusions. Or la règle (V-ELIM) ajoute sa conclusion à l'historique ; donc, à un certain point de la branche, toutes apparaissent dans l'historique. A cet endroit, il est donc possible de couronner la branche par une règle (HIST). Puisque l'algorithme donne la priorité à (HIST) par rapport aux autres règles, il n'a pas pu exhiber cette branche infinie. □

Pour évaluer la complexité de l'algorithme, nous supposerons que le graphe de contraintes considéré est dénué d'occurrences de $\sqcup$ et $\sqcap$, et qu'il vérifie de surcroît l'invariant des petits termes – invariant que nous n'avons pas encore utilisé au cours de ce chapitre. Cette hypothèse, on l'a dit, ne restreint pas la puissance de l'algorithme, mais simplifie l'analyse.

Sous cette condition, les buts rencontrés par l'algorithme se composent alors soit de deux petits termes, soit de deux variables. Etant donné un but composé de deux variables, les opérations à effectuer sont toujours les mêmes. D'abord, tester si les règles (HIST) ou (TAUTO) s'appliquent. Sinon, appliquer (V-ELIM), qui fournit un but composé de deux petits termes. Si ce but est insoluble, alors l'algorithme échoue ; dans le cas contraire, l'une des règles de décomposition structurelle s'applique, conduisant à nouveau à un certain nombre de buts composés de deux variables.

Ainsi, l'historique contient uniquement des buts composés de deux variables. Si le graphe de contraintes contient n variables, la taille de l'historique est donc bornée par $O(n^2)$. Or, si tous les buts possibles figurent dans l'historique, alors la règle (HIST) s'applique nécessairement et permet de conclure la branche en cours avec succès. La profondeur des branches est donc également bornée par $O(n^2)$, ce qui donne à l'algorithme une borne de complexité exponentielle.

Le même raisonnement s'applique à l'algorithme d'Amadio et Cardelli [9]. Or, le problème qu'ils étudient peut être résolu en temps $O(n^2)$, comme le font remarquer Kozen, Palsberg et Schwartzbach [33], qui proposent un algorithme à base d'automates. La même amélioration est possible ici ; en voici le principe, exposé d'une façon qui nous semble particulièrement simple.

Lorsque nous avons défini notre axiomatisation, nous aurions parfaitement pu éviter l'utilisation d'un historique et l'introduction de la règle (HIST). Pour cela, il nous aurait suffi de manipuler des arbres de preuve *réguliers*, plutôt que d'exiger l'existence d'une dérivation *finie*. En fait, l'introduction d'un but b dans l'historique correspond à l'introduction d'un lieur μb, et l'utilisation de b par la règle (HIST) n'est autre qu'une référence au nœud où b a été lié. L'algorithme que nous avons décrit construit donc un arbre de preuve régulier, décrit de façon finie à l'aide de lieurs μ. Or, cette méthode de description n'est pas optimale ; mieux vaut construire un *automate de termes*, qui permettra une représentation plus compacte.

La construction d'un automate décrivant l'arbre de preuve est fort simple. Les états de l'automate sont au nombre de n^2 : il s'agit de tous les buts composés de deux variables, l'état initial étant le but initial. Pour construire les transitions, on considère d'abord l'état initial. On lui applique l'étape élémentaire décrite plus haut. Plus précisément, on teste d'abord si cet état a déjà été traité. Si ce n'est pas le cas, on lui applique soit (TAUTO), soit (V-ELIM) suivie d'une règle de décomposition structurelle. Si celle-ci produit des sous-buts, on crée une transition vers chacun des états correspondants, étiquetée 0 ou 1, comme il convient ; puis, on applique le même procédé à chacun de ces états. Si, au cours de ce procédé, on rencontre une contrainte insoluble, l'algorithme échoue ; sinon, on a construit une dérivation régulière, décrite par un automate. Chaque état, ou but, est considéré au plus une fois, donc la complexité de l'algorithme est $O(n^2)$.

Le principe de cette amélioration est le même que celui utilisé par Kozen, Palsberg et Schwartzbach pour améliorer l'algorithme d'Amadio et Cardelli. Cependant, il nous a semblé intéressant d'insister sur le fait que, dans chaque cas, la version naïve et la version efficace de l'algorithme calculent en fait le même objet, à savoir une dérivation régulière, mais en produisent des représentations différentes.

Enfin, notons que la version efficace peut être décrite, informellement, en deux mots. Il suffit d'implémenter la version naïve, donnée par les règles de la figure 8.2, puis de la raffiner en partageant l'historique entre les branches. En d'autres termes, il suffit de faire de l'historique une variable globale, dans laquelle on ne fait *qu'ajouter* des buts ; l'historique ne diminue plus lors du retour des appels récursifs. Cette remarque a d'ailleurs été formulée par Trifonov et Smith [49], mais sans cette justification.

8.5 Incomplétude

Pour montrer que la relation d'implication entre contraintes est décidable, il resterait à prouver que notre axiomatisation est complète. Malheureusement, tel n'est pas le cas.

Proposition 8.4 *L'axiomatisation de l'implication est incomplète : $C \Vdash \tau \leq \tau'$* n'entraîne pas *$C \vdash \tau \leq \tau'$.*

Démonstration. Nous donnons plusieurs contre-exemples ci-dessous. □

Exemple. Soit C le graphe de contraintes associé à $\{\alpha \to \bot \leq \alpha\}$. Soit τ le type $(\bot \to \top) \to \bot$.

Nous prétendons que $C \Vdash \tau \leq \alpha$. En effet, considérons une solution ρ de C. ρ satisfait $\alpha \to \bot \leq \alpha$, donc doit associer à α un type brut dont le constructeur de tête est $\top$ ou $\to$. Dans le premier cas, ρ satisfait également $\tau \leq \alpha$. Dans le second cas, $\rho(\alpha)$ est inférieur au plus grand type de fonction : $\rho(\alpha) \leq \bot \to \top$. Par ailleurs, nous avons $\rho(\alpha) \to \bot \leq \rho(\alpha)$. Par transitivité, il en découle que $(\bot \to \top) \to \bot \leq \rho(\alpha)$, donc ρ satisfait $\tau \leq \alpha$.

Cependant, on vérifie que l'assertion $C \Vdash \tau \leq \alpha \to \bot$ est fausse. Une substitution brute ρ qui à α associe $\top$ est solution de C ; pourtant elle n'est pas solution de $\tau \leq \alpha \to \bot$.

A présent, il est facile de vérifier que l'assertion $C \vdash \tau \leq \alpha$ est fausse. En tentant de construire une dérivation, l'algorithme applique la règle (V-ELIM), qui donne $\tau \leq \alpha \to \bot$ comme nouveau but. Mais nous avons justement montré que ce but est faux dans le modèle ; il n'admet donc pas de dérivation, sans quoi le théorème 8.1 serait contredit. Donc, l'algorithme échoue.

Nous avons donc montré que la règle (V-ELIM) est essentiellement incomplète. Il est facile de vérifier qu'elle est en fait la seule source d'incomplétude : dans toutes les autres règles, les prémisses sont équivalentes (au sens de l'implication de contraintes) à la conclusion.

Exemple. On pourrait penser que l'incomplétude, dans l'exemple ci-dessus, provient du fait qu'une variable de types apparaît en position contravariante dans sa propre borne. Cependant, voici une variante du même exemple où ce n'est plus le cas.

Soit C le graphe de contraintes vide de domaine $\{\alpha\}$. Soit τ défini comme précédemment. Alors, on vérifie que $C \Vdash \tau \leq \alpha \sqcup (\alpha \to \bot)$; la preuve est essentiellement identique à la précédente.

Cependant, l'application de la règle (V-ELIM) à ce but produit comme nouveau but $\tau \leq \alpha \to \bot$, parce que α a été remplacé par sa borne inférieure dans C, qui est $\bot$. Encore une fois, ce nouveau but n'est pas impliqué par C.

Ainsi, la règle (V-ELIM) reste incomplète même en l'absence de contraintes récursives, et même en l'absence de toute contrainte. Ici, l'incomplétude semble provenir du fait que l'algorithme n'est pas capable de prédire de façon suffisamment fine la valeur de $\alpha \sqcup (\alpha \to \bot)$ quand α varie. Un algorithme basé sur une analyse par cas (considérant que toute solution de C doit associer à α soit $\bot$, soit $\top$, soit un type de fonction, et analysant chaque cas séparément) ne souffrirait pas de ce problème. Cependant, il semble difficile de garantir sa terminaison. Dans le cas où α représente un type de fonction, l'algorithme introduit deux nouvelles variables de type pour représenter son domaine et son codomaine. La taille du problème peut donc augmenter au fur et à mesure que l'analyse avance, et il est difficile d'énoncer un critère de terminaison pertinent.

Exemple. Enfin, d'après les contre-exemples précédents, on pourrait croire que l'incomplétude provient de la présence d'un constructeur contravariant. Cette supposition est infondée, comme le montre l'exemple suivant, mis au point par Joachim Niehren.

Supposons que le langage des types contienne un constructeur de type unaire et covariant F (cf. section 14.3). Soit C le graphe de contraintes associé à

$$\{F(\alpha) \leq \beta,\, \alpha \leq F(\beta)\}$$

Alors, nous prétendons que $C \Vdash \alpha \leq \beta$. Pour le démontrer, nous allons vérifier, par induction sur k, que $C \Vdash_k \alpha \leq \beta$ est vrai pour tout $k \in \mathbb{N}^+$. Le cas $k = 0$ est immédiat. Supposons que le résultat soit acquis pour un k donné ; soit ρ une $(k+1)$-solution de C. Si ρ associe $\bot$ à α ou $\top$ à β, alors ρ vérifie $\alpha \leq \beta$ et nous avons le résultat désiré. Sinon, notons que ρ est une 1-solution de $\alpha \leq F(\beta)$; il en découle que $\rho(\alpha)$ est de la forme $F(\tau_\alpha)$, où τ_α est un certain type brut. De même, $\rho(\beta)$ est de la forme $F(\tau_\beta)$. Du fait que ρ est une $(k+1)$-solution de C, nous déduisons $F(\tau_\alpha) \leq_k \tau_\beta$ et $\tau_\alpha \leq_k F(\tau_\beta)$. En d'autres termes, toute substitution brute ρ' qui à α associe τ_α et à β associe τ_β est une k-solution de C. D'après notre hypothèse d'induction, ρ' k-satisfait $\alpha \leq \beta$, donc $\tau_\alpha \leq_k \tau_\beta$. Il en découle que $F(\tau_\alpha) \leq_{k+1} F(\tau_\beta)$, c'est-à-dire que ρ $(k+1)$-satisfait $\alpha \leq \beta$. L'induction est terminée ; nous avons démontré que $C \Vdash \alpha \leq \beta$.

Or, il est facile de vérifier que ce but n'admet aucune dérivation. La règle (V-Elim) transforme $\alpha \leq \beta$ en $F(\beta) \leq F(\alpha)$, qui est à son tour réécrit en $\beta \leq \alpha$ par la règle de décomposition structurelle associé au constructeur covariant F. Or, $C \Vdash \beta \leq \alpha$ est faux, puisqu'une substitution brute qui à α associe $\bot$ et à β associe $\top$ est solution de C. Donc, encore une fois, la règle (V-Elim) est incomplète.

L'essence de l'incomplétude est difficile à saisir ici. On note que l'analyse par cas mentionnée précédemment fonctionnerait parfaitement, puisque la décomposition du problème conduit à une nouvelle instance du même problème, à α-conversion près, ce qui est facile à détecter. Cependant, il existe des situations dans lesquelles une telle décomposition crée un « nuage » de variables de type superflues, et la détection d'un motif récurrent devient alors non triviale.

La décidabilité de l'implication de contraintes reste donc pour l'instant un problème ouvert. Seule une borne inférieure de complexité est connue : Henglein et Rehof [28] démontrent que le problème est PSPACE-difficile.

Chapitre 9

Décision de la comparaison de schémas

L'objectif de ce chapitre est de décrire un algorithme qui, étant donnés deux schémas de types, décide s'ils sont comparables au sens de la relation $\leq^{\forall}$; en d'autres termes, si le premier est plus général que le second. Cet algorithme est dû à Trifonov et Smith [49], qui n'en donnent toutefois pas de preuve détaillée. Il constitue une généralisation de l'algorithme d'implication de contraintes exposé au chapitre 8 ; en fait, on peut le voir comme une combinaison de celui-ci et de l'algorithme de clôture. Il présente les mêmes problèmes d'incomplétude que l'algorithme d'implication.

Cet algorithme est utilisé, en pratique, pour comparer le schéma de types inféré pour une certaine expression à celui déclaré par l'utilisateur dans la signature du module. Par ailleurs, l'algorithme sert également de base théorique à l'une de nos méthodes de simplification, le dépoussiérage.

Pour des raisons de simplicité, nous nous restreindrons au cas où le schéma de droite est *simple*, c'est-à-dire sans occurrences de $\sqcup$ ou $\sqcap$. L'algorithme ne s'étend pas de façon naturelle au cas général. Cette restriction n'entraîne pas de perte de puissance, puisque tout schéma de types est équivalent à un schéma simple (cf. chapitre 11).

Nous commençons par quelques préliminaires techniques de peu d'importance (section 9.1). Nous introduisons ensuite la notion d'*extension faiblement close*, qui formera la base théorique de notre algorithme (section 9.2). La section 9.3 explique, de façon principalement informelle, les idées qui mènent à la conception de celui-ci. Enfin, l'algorithme est défini puis prouvé dans la section 9.4.

9.1 Préliminaires

Cette section a pour objet d'étendre légèrement l'axiomatisation de l'implication de contraintes développée au chapitre 8, de façon à lui permettre de traiter une classe de buts plus étendue. Elle est peu complexe, et d'intérêt purement technique.

La section 9.2, qui introduit la notion de *clôture faible*, aura besoin pour cela de manipuler des contraintes de la forme $\tau \leq \tau'$, où τ et τ' sont des petits termes *arbitraires*. En particulier, τ comme τ' peut être un pos-type ou un neg-type. Définissons donc

Définition 9.1 *Une* contrainte feuille *est une paire de termes feuilles τ et τ', notée $\tau \leq \tau'$. Une* petite contrainte *est une paire de petits termes τ et τ', notée $\tau \leq \tau'$.*

Rappelons que jusqu'ici, l'axiomatisation de l'implication de contraintes traite des buts de la forme $\tau \leq \tau'$, où $(\tau, \tau') \in \mathcal{T}^- \times \mathcal{T}^+$. Ici, nous souhaitons pouvoir spécifier des contraintes feuilles et des petites contraintes en tant que buts. Cela est assez facile, puisque la forme de ces nouveaux buts est très restreinte.

$$\frac{\forall \alpha \in V \quad C, H \vdash \alpha \leq \tau'}{C, H \vdash \sqcup V \leq \tau'} \quad (\sqcup\text{-Elim})$$

$$\frac{\forall \alpha' \in V' \quad C, H \vdash \tau \leq \alpha'}{C, H \vdash \tau \leq \sqcap V'} \quad (\sqcap\text{-Elim})$$

Fig. 9.1: Axiomatisation de l'implication, étendue aux contraintes feuilles

Nous étendons donc l'axiomatisation aux contraintes feuilles en ajoutant les deux règles données par la figure 9.1. Notons que cette extension n'est pas « récursive » : étant donnée une contrainte feuille, les règles ($\sqcup$-Elim) and ($\sqcap$-Elim) peuvent être appliquées au plus une fois chacune au bas de la dérivation, et nulle part ailleurs. Ainsi, une dérivation dans ce système étendu est simplement une famille de dérivations dans l'ancien système, éventuellement rassemblées grâce aux nouvelles règles. Notre extension n'est donc qu'une commodité qui permet de coder une conjonction de buts en un seul.

Pour vérifier que l'algorithme est toujours correct, il suffit de vérifier que la proposition 8.1 est toujours vérifiée ; le reste suit. En voici donc une nouvelle version.

Proposition 9.1 *Soit une dérivation de $C, H \vdash \tau \leq \tau'$ de profondeur k, où $\tau \leq \tau'$ est une contrainte feuille. Alors*

$$\forall j \leq k \quad C \Vdash_j \tau \leq \tau'$$

Démonstration. Il suffit de considérer les deux nouvelles règles ; puisqu'elles sont symétriques, traitons ($\sqcup$-Elim). Supposons $\forall \alpha \in V \quad C \Vdash_j \alpha \leq \tau'$. Nous devons vérifier $C \Vdash_j \sqcup V \leq \tau'$. C'est là une conséquence immédiate de la proposition 1.6. □

Nous avons donc étendu l'axiomatisation de l'implication de contraintes aux contraintes feuilles. L'extension aux petites contraintes est à présent immédiate. En effet, étant donnée une petite contrainte, l'application de ($\bot$-Elim), ($\top$-Elim) ou ($\rightarrow$-Elim) la décomposera en contraintes feuilles. Enonçons à nouveau le théorème de correction :

Théorème 9.1 *Soit C un graphe de contraintes. Soit c une contrainte feuille ou une petite contrainte telle que*

$$C \vdash c$$

Alors

$$\forall k \in \mathbb{N}^+ \cup \{\infty\} \quad C \Vdash_k c$$

Démonstration. Identique à celle du théorème 8.1, en utilisant la proposition 9.1 au lieu de la proposition 8.1. □

9.2 Clôture faible

Définition 9.2 *Soit C' un graphe de contraintes de domaine V'. Une* extension *de C' est un graphe de contraintes C de domaine $V \cup V'$ tel que $\leq_C$, $C^\downarrow$ et $C^\uparrow$, une fois restreints à V', coïncident avec $\leq_{C'}$, $C'^\downarrow$, et $C'^\uparrow$, respectivement.*

Dans cette section, nous désirons donner une condition suffisante sur une telle extension pour que toute solution de C' s'étende à une solution de C. Cette condition formera ensuite le fondement théorique de l'algorithme de comparaison polymorphe entre schémas de types.

La condition que nous énonçons ci-dessous est une sorte de condition de clôture, similaire à celle de la définition 3.10, à deux différences près.

Premièrement, nous désirons émettre une condition aussi faible que possible. Par conséquent, nous remplaçons les notions de transitivité et d'inclusion entre types par des assertions prouvables d'implication de contraintes, c'est-à-dire des jugements dérivables dans notre axiomatisation de l'implication. Nous remplaçons ainsi des critères simples et purement syntaxiques par une notion plus fine, qui autorise plus de flexibilité.

Deuxièmement, lorsque nous avons défini la clôture d'un graphe de contraintes, il s'agissait d'exhiber une solution de ce graphe. Ici, nous traitons un problème plus général, qui est celui d'étendre une solution donnée à un graphe plus grand. A cause de cela, les variables de V et de V' jouent des rôles différents, et cette distinction est utilisée pour affaiblir encore la condition.

Définition 9.3 *Soit C' un graphe de contraintes de domaine V'. Soit C une extension de C' à $V \cup V'$, où V et V' sont disjoints. Dans ce qui suit, les variables de V seront notées $\alpha, \beta, \ldots$ tandis que celles de V' seront notées $\alpha', \beta', \ldots$*

C est une extension faiblement close *de C' ssi, pour tous α, β, α', β', γ' :*

- $\alpha' \leq_C \beta$ *et* $\beta \leq_C \gamma'$ *impliquent* $C \vdash \alpha' \leq \gamma'$ *;*
- $\alpha' \leq_C \alpha$ *et* $\alpha \leq_C \beta$ *impliquent* $\alpha' \leq_C \beta$ *;*
- $\alpha \leq_C \beta$ *et* $\beta \leq_C \beta'$ *impliquent* $\alpha \leq_C \beta'$ *;*
- $\alpha \leq_C \alpha'$ *implique* $\exists \beta \geq_C \alpha \quad C \vdash C^{\downarrow}(\beta) \leq C^{\downarrow}(\alpha')$ *;*
- $\alpha' \leq_C \alpha$ *implique* $\exists \beta \leq_C \alpha \quad C \vdash C^{\uparrow}(\alpha') \leq C^{\uparrow}(\beta)$ *;*
- $\alpha \leq_C \beta$ *implique* $C \vdash C^{\downarrow}(\alpha) \leq C^{\downarrow}(\beta)$ *et* $C \vdash C^{\uparrow}(\alpha) \leq C^{\uparrow}(\beta)$ *;*
- $C \vdash C^{\downarrow}(\alpha) \leq C^{\uparrow}(\alpha)$.

On notera que, dans les conditions ci-dessus, certaines assertions d'implication utilisent l'axiomatisation étendue introduite par la section 9.1. Par exemple, dans l'assertion $C \vdash C^{\downarrow}(\alpha) \leq C^{\downarrow}(\beta)$, les deux membres du but sont des pos-types, donc l'axiomatisation décrite au chapitre 8 n'est pas suffisante. Grâce à l'invariant des petits termes, $C^{\downarrow}(\alpha) \leq C^{\downarrow}(\beta)$ est une petite contrainte, donc cette assertion entre dans le cadre de notre extension.

Nous pouvons à présent passer à la démonstration de la propriété principale des extensions faiblement closes. Commençons par une variante de la proposition 3.4 :

Proposition 9.2 *Soit $k \in \mathbb{N}^+$. Si un graphe de contraintes C implique prouvablement une petite contrainte c (c'est-à-dire si $C \vdash c$), alors toute k-solution de C est $(k+1)$-solution de c.*

Démonstration. Soit $k \in \mathbb{N}^+$. Puisque les membres de c sont des petits termes, c est (par décomposition structurelle) équivalente à un ensemble S de contraintes feuilles. Considérons $c' \in S$. Puisque l'axiomatisation de l'implication procède également par décomposition structurelle, $C \vdash c$ implique $C \vdash c'$. D'après le théorème 9.1, ceci implique $C \Vdash_k c'$. Ainsi, nous avons prouvé que toute k-solution de C est k-solution de S. Or, toute k-solution de S est $(k+1)$-solution de c, par décomposition structurelle. □

Théorème 9.2 *Soit C une extension faiblement close de C'. Alors toute solution de C' s'étend en une solution de C.*

Démonstration. La preuve est similaire, en principe, à celle du théorème 3.1, moyennant toutefois quelques différences importantes. D'abord, puisque la définition de la clôture faible utilise l'implication prouvable au lieu de l'inclusion entre types, nous utiliserons la proposition 9.2 au lieu de la proposition 3.4. Ensuite et surtout, parce que nous prouvons un résultat plus général sous des hypothèses affaiblies, les détails de la preuve sont plus délicats. Néanmoins, le principe général reste le même : exhiber une substitution brute, et prouver qu'il s'agit d'une k-solution de C, pour tout $k \in \mathbb{N}^+$, par induction sur k.

Soient $V' = \mathrm{dom}(C')$ et $V \cup V' = \mathrm{dom}(C)$, où V et V' sont disjoints. Dans ce qui suit, les variables de V seront notées $\alpha, \beta, \ldots$ tandis que celles de V' seront notées $\alpha', \beta', \ldots$

Soit ρ' une solution de C'. D'après la proposition 2.9, il existe un système contractif S', dont le domaine contient V', et dont l'unique solution coïncide avec ρ' sur V'. Il est clair que l'on peut exiger $\mathrm{dom}(S') \cap V = \varnothing$.

Considérons alors le système contractif S, de domaine $\mathrm{dom}(S') \cup V$, défini comme suit :

$$\begin{aligned} \alpha' \in \mathrm{dom}(S') &\mapsto S'(\alpha') \\ \alpha \in V &\mapsto C^{\downarrow}(\alpha) \sqcup (\sqcup\{S'(\alpha') \, ; \, \alpha' \in V' \wedge \alpha' \leq_C \alpha\}) \end{aligned}$$

S est bien contractif, car son image ne contient que des types construits. Soit ρ l'unique solution de S. ρ est en particulier solution de S', donc coïncide avec ρ' sur V' ; c'est une extension de ρ'.

Remarquons que pour tout $\alpha \in V$, nous avons

$$\begin{aligned} \rho(\alpha) &= \rho(C^{\downarrow}(\alpha)) \sqcup (\sqcup\{\rho(S'(\alpha')) \, ; \, \alpha' \in V' \wedge \alpha' \leq_C \alpha\}) \\ &= \rho(C^{\downarrow}(\alpha)) \sqcup (\sqcup\{\rho(\alpha') \, ; \, \alpha' \in V' \wedge \alpha' \leq_C \alpha\}) \end{aligned}$$

Nous allons à présent vérifier, par induction sur $k \in \mathbb{N}^+$, que ρ est k-solution de C. Le résultat est immédiat pour $k = 0$; supposons à présent qu'il a été vérifié pour un certain $k \in \mathbb{N}^+$.

D'abord, considérons deux variables reliées par $\leq_C$. Quatre sous-cas se présentent, selon que chacune d'elles appartient à V ou V'.

1. $\alpha' \in V'$ et $\beta' \in V'$. Alors $\alpha' \leq_{C'} \beta'$ (car C est une extension de C'). ρ est une extension de ρ', donc satisfait $\alpha' \leq \beta'$.
2. $\alpha' \in V'$ et $\alpha \in V$. Alors, selon la remarque ci-dessus,

$$\begin{aligned} \rho(\alpha) &= \rho(C^{\downarrow}(\alpha)) \sqcup (\sqcup\{\rho(\alpha') \, ; \, \alpha' \in V' \wedge \alpha' \leq_C \alpha\}) \\ &\geq \rho(\alpha') \end{aligned}$$

donc ρ satisfait $\alpha' \leq \alpha$.

3. $\alpha \in V$ et $\alpha' \in V'$. Nous avons

$$\rho(\alpha) = \rho(C^{\downarrow}(\alpha)) \sqcup (\sqcup\{\rho(\gamma') \, ; \, \gamma' \in V' \wedge \gamma' \leq_C \alpha\})$$

donc il suffit de prouver que

$$\rho(C^{\downarrow}(\alpha)) \leq_{k+1} \rho(\alpha') \tag{1}$$

$$\forall \gamma' \in V' \quad \gamma' \leq_C \alpha \Rightarrow \rho(\gamma') \leq_{k+1} \rho(\alpha') \tag{2}$$

C est une extension faiblement close de C', et $\alpha \leq_C \alpha'$; par conséquent, il existe $\beta \in V$ tel que $\alpha \leq_C \beta$ et $C \vdash C^{\downarrow}(\beta) \leq C^{\downarrow}(\alpha')$. Puisque $\alpha \leq_C \beta$, en utilisant une nouvelle fois le fait que C est une extension faiblement close de C', nous obtenons $C \vdash C^{\downarrow}(\alpha) \leq C^{\downarrow}(\beta)$. ρ est k-solution de C ; donc, d'après la proposition 9.2, ρ est $(k+1)$-solution de ces deux contraintes. Donc,

$$\rho(C^{\downarrow}(\alpha)) \leq_{k+1} \rho(C^{\downarrow}(\alpha'))$$

De plus, sachant que $\alpha' \in V'$ et que ρ est solution de C',

$$\rho(C^{\downarrow}(\alpha')) \leq \rho(\alpha')$$

Par transitivité, nous obtenons (1). A présent, soit $\gamma' \in V'$ tel que $\gamma' \leq_C \alpha$. Puisque $\alpha \leq_C \alpha'$ et que C est une extension faiblement close de C', nous avons $C \vdash \gamma' \leq \alpha'$. Considérons la dérivation de cette assertion. Nous sommes nécessairement dans l'un des cas suivants :

– $\gamma' \leq_C \alpha'$. Alors, puisque $\gamma' \in V'$ et $\alpha' \in V'$, et puisque C est une extension de C', nous avons $\gamma' \leq_{C'} \alpha'$, et ρ satisfait $\gamma' \leq \alpha'$.
– $C \vdash C^{\uparrow}(\gamma') \leq C^{\downarrow}(\alpha')$. Alors, ρ étant k-solution de C, il est $(k+1)$-solution de cette contrainte entre termes construits. Par ailleurs, puisque ρ est solution de C', il satisfait $\gamma' \leq C'^{\uparrow}(\gamma') = C^{\uparrow}(\gamma')$ et $C^{\downarrow}(\alpha') = C'^{\downarrow}(\alpha') \leq \alpha'$. Il en découle que ρ $(k+1)$-satisfait $\gamma' \leq \alpha'$.

Dans les deux cas, ρ $(k+1)$-satisfait $\gamma' \leq \alpha'$, donc nous avons vérifié (2).

4. $\alpha \in V$ et $\beta \in V$. Nous avons

$$\rho(\alpha) = \rho(C^{\downarrow}(\alpha)) \sqcup (\sqcup\{\rho(\alpha')\,;\, \alpha' \in V' \wedge \alpha' \leq_C \alpha\})$$
$$\rho(\beta) = \rho(C^{\downarrow}(\beta)) \sqcup (\sqcup\{\rho(\beta')\,;\, \beta' \in V' \wedge \beta' \leq_C \beta\})$$

Nous allons montrer que chaque membre de l'expression $\sqcup$ de la première ligne ci-dessus est inférieur (au rang $k+1$) à un membre de l'expression $\sqcup$ de la seconde ligne.

Nous avons $\alpha \leq_C \beta$, et C est une extension faiblement close de C', donc $C \vdash C^{\downarrow}(\alpha) \leq C^{\downarrow}(\beta)$. Comme précédemment, puisque ρ k-satisfait C, il $(k+1)$-satisfait cette contrainte.

A présent, soit $\alpha' \in V'$ tel que $\alpha' \leq_C \alpha$. Puisque $\alpha \leq_C \beta$ et que C est une extension faiblement close de C', nous avons $\alpha' \leq_C \beta$. Donc α' fait partie des β' ci-dessus.

Nous avons vérifié que tout membre de la première ligne est inférieur, au rang $k+1$, à un élément de la seconde ; par conséquent, $\rho(\alpha) \leq_{k+1} \rho(\beta)$.

Nous avons donc démontré que ρ est $(k+1)$-solution de toute contrainte de $\leq_C$.

A présent, soit $\alpha \in V$. Nous allons prouver que ρ est $(k+1)$-solution de $C^{\downarrow}(\alpha) \leq \alpha \leq C^{\uparrow}(\alpha)$. (Cette assertion est vraie pour $\alpha' \in V'$, parce que ρ étend ρ' ; c'est pourquoi nous considérons uniquement le cas $\alpha \in V$.) La première inégalité est conséquence directe de la définition de $\rho(\alpha)$. Considérons donc la seconde. Nous avons

$$\rho(\alpha) = \rho(C^{\downarrow}(\alpha)) \sqcup (\sqcup\{\rho(\alpha')\,;\, \alpha' \in V' \wedge \alpha' \leq_C \alpha\})$$

Nous allons vérifier que chaque membre de cette expression $\sqcup$ est inférieur, au rang $k+1$, à $\rho(C^{\uparrow}(\alpha))$.

C étant une extension faiblement close de C', nous avons $C \vdash C^{\downarrow}(\alpha) \leq C^{\uparrow}(\alpha)$. Comme nous l'avons vu précédemment, il en découle que ρ est $(k+1)$-solution de cette contrainte.

Soit à présent $\alpha' \in V'$ tel que $\alpha' \leq_C \alpha$. Puisque C est une extension faiblement close de C', il existe $\beta \in V$ tel que $\beta \leq_C \alpha$ et $C \vdash C^{\uparrow}(\alpha') \leq C^{\uparrow}(\beta)$. Puisque $\beta \leq_C \alpha$, et puisque C est une extension faiblement close de C', nous avons également $C \vdash C^{\uparrow}(\beta) \leq C^{\uparrow}(\alpha)$. Encore une fois, ρ $(k+1)$-satisfait ces contraintes. Par ailleurs, puisque $\alpha' \in V'$, ρ satisfait $\alpha' \leq C'^{\uparrow}(\alpha') = C^{\uparrow}(\alpha')$. Par conséquent, ρ $(k+1)$-satisfait $\alpha' \leq C^{\uparrow}(\alpha)$.

Nous avons à présent prouvé que ρ est $(k+1)$-solution de C ; l'induction est terminée, et ρ est solution de C. □

Il est intéressant de noter que, en corollaire, nous obtenons une condition plus faible pour qu'un graphe de contraintes admette une solution :

Définition 9.4 *Un graphe de contraintes C de domaine V est dit* faiblement clos *ssi pour tous $\alpha, \beta \in V$:*

– *$\alpha \leq_C \beta$ implique $C \vdash C^{\downarrow}(\alpha) \leq C^{\downarrow}(\beta)$ et $C \vdash C^{\uparrow}(\alpha) \leq C^{\uparrow}(\beta)$;*
– *$C \vdash C^{\downarrow}(\alpha) \leq C^{\uparrow}(\alpha)$.*

Un schéma de types $A \Rightarrow \tau \mid C$ est faiblement clos ssi C l'est.

Proposition 9.3 *Tout graphe de contraintes faiblement clos admet une solution.*

Démonstration. Soit C un graphe de contraintes faiblement clos. C est une extension faiblement close du graphe vide, d'où le résultat. □

Nous n'utiliserons cette notion de clôture faible que dans la preuve de l'algorithme de comparaison polymorphe entre schémas de types. En effet, dans le reste de notre théorie, la notion de clôture, plus simple, se révèle parfaitement adaptée. (La preuve de l'algorithme de canonisation, au chapitre 11, constitue une exception ; elle introduira une troisième notion de clôture, intermédiaire entre clôture et clôture faible.)

Du point de vue de l'implémentation, l'idée d'utiliser la clôture faible pourrait venir à l'esprit. En effet, celle-ci pose moins de conditions sur le graphe de contraintes ; on pourrait donc imaginer qu'un calcul de clôture faible produise un graphe plus petit, participant ainsi au processus de simplification. En fait, il est beaucoup plus facile, et efficace, de travailler avec la notion originale de clôture, pour plusieurs raisons.

La première raison, qui n'est pas la meilleure, mais la première que nous ayons considérée historiquement, réside dans la difficulté de mettre au point un algorithme incrémental de calcul de clôture faible. En effet, il est facile de *vérifier* qu'un graphe de contraintes donné est faiblement clos, grâce à l'algorithme d'implication de contraintes ; mais il est difficile, étant donné un graphe de contraintes, de calculer le plus petit graphe faiblement clos qui lui soit équivalent (si tant est qu'un tel graphe existe, ce qui est probablement faux en général). Dans le cas de la clôture, nous disposons d'un algorithme glouton simple (cf. définition 7.1). Il ajoute les contraintes qui doivent *nécessairement* l'être, jusqu'à atteindre un point fixe. Dans le cas de la clôture faible, l'utilisation d'assertions d'implication complique la situation. Il peut apparaître nécessaire d'ajouter une contrainte, parce qu'elle n'est pas impliquée par le graphe dans son état actuel ; mais celle-ci peut s'avérer plus tard inutile, parce qu'elle peut être déduite d'autres contraintes qui auront été ajoutées après elle. En d'autres termes, la fonction dont nous voulons calculer un point fixe n'est pas monotone. Il est peut-être possible de contourner ce problème, mais le jeu n'en vaut certainement pas la chandelle.

La deuxième raison, en effet, est simple et sans appel. Nous verrons au chapitre 12 que lorsque l'*invariant de mono-polarité* est en vigueur, alors tout graphe de contraintes *dépoussiéré* (cf. chapitre 10) est automatiquement clos et faiblement clos. Il n'y a donc plus de différence pratique entre les deux notions, et il n'est pas intéressant de pousser plus loin notre théorie de la clôture faible.

9.3 Principe

Munis des outils théoriques développés dans la section précédente, nous sommes à présent en mesure d'énoncer une condition suffisante pour que deux schémas de types soient comparables au sens de la relation $\leq^\forall$. Cette condition servira ensuite de base à l'algorithme. Commençons par une explication informelle de l'intuition sous-jacente.

Rappelons que l'assertion

$$A \Rightarrow \tau \mid C \leq^\forall A' \Rightarrow \tau' \mid C'$$

est équivalente à

$$\forall \rho' \vdash C' \quad \exists \rho \vdash C \quad \rho(A \Rightarrow \tau) \leq \rho'(A' \Rightarrow \tau')$$

Ainsi, pour montrer qu'une assertion de comparaison entre schémas est vraie, nous devons être capables, étant donnée une instance quelconque du schéma de droite, de produire une instance plus petite du schéma de gauche.

Considérons le cas où le schéma de droite ne contient aucune variable. Alors, le problème se réduit à un simple problème de solubilité, puisqu'il s'agit alors de déterminer si $C + (A \Rightarrow \tau \leq A' \Rightarrow \tau')$ admet une solution. Ce problème, nous l'avons vu, est décidable : il suffit de calculer la clôture de cet ensemble de contraintes. Si le calcul échoue, il n'existe aucune solution ; s'il réussit, nous obtenons un graphe clos, à partir duquel il est facile d'exhiber une solution.

Revenons maintenant au cas où le domaine V' du schéma de droite est non vide. (Nous supposerons que les deux schémas sont de domaines disjoints, ce qui n'est pas restrictif, puisque la relation $\leq^\forall$ contient l'α-conversion.) A nouveau, le problème revient à déterminer si le graphe ci-dessus admet une solution, mais cette fois, nous ne contrôlons plus la valeur des variables de V'. Il semble alors naturel d'effectuer à nouveau un calcul de clôture, mais en considérant cette fois les variables de V' comme des constantes inconnues. On devra alors échouer si de nouvelles contraintes apparaissent sur elles, puisque nous n'avons pas le choix de leur valeur. Si le calcul réussit, alors nous avons résolu le problème. De façon informelle, le graphe clos décrit alors les valeurs qu'il faut attribuer aux variables de gauche, en termes des valeurs des variables de droite. Trifonov et Smith [49] nomment ces dernières *rigides*, tandis que les premières sont appelées *flexibles*, en accord avec l'idée qu'elles sont seules à pouvoir varier.

Il faut apporter une correction au discours ci-dessus : au lieu d'utiliser la définition habituelle de la clôture, nous la remplacerons par la notion de clôture faible introduite à la section 9.2. La première, moins précise, demanderait l'addition de plus de contraintes. En particulier, nous risquerions d'être amenés à ajouter de nouvelles contraintes sur les variables rigides, ce qui provoquerait un échec de l'algorithme. La clôture faible est plus fine et diminue l'acuité du problème : dans de nombreux cas, ces nouvelles contraintes seront prouvablement impliquées par des contraintes existantes, et il sera donc inutile de les ajouter au graphe, évitant ainsi l'échec. (Néanmoins, le problème existe toujours, puisque la notion de clôture faible est basée sur une axiomatisation incomplète de l'implication de contraintes, et notre nouvel algorithme sera lui aussi incomplet.) Ainsi, les variables rigides sont considérées comme des constantes inconnues, mais nous savons qu'elles vérifient les relations exprimées par le graphe C', et nous faisons usage de cette information à travers la relation d'implication de contraintes.

Exemple. Considérons l'assertion

$$\alpha \to \alpha \leq^\forall \beta' \to \gamma' \mid C'$$

où $C' = \varnothing + (\beta' \leq \gamma')$. Essayons de calculer une extension faiblement close de C' qui implique

$$\alpha \to \alpha \leq \beta' \to \gamma'$$

Nous ajoutons à C' les contraintes $\beta' \leq \alpha$ et $\alpha \leq \gamma'$. Par transitivité, cela demande que $\beta' \leq \gamma'$ qui est une contrainte sur les variables « rigides » de V'. Heureusement, cette contrainte est déjà impliquée par C', donc nous avons construit une extension faiblement close, et nous avons terminé avec succès.

Supposons donnée une solution ρ' de C'. Soit ρ une substitution brute qui à α associe n'importe quel type brut compris entre $\rho'(\beta')$ et $\rho'(\gamma')$. Il est facile de vérifier que ρ est un témoin de l'assertion de comparaison de schémas. Ainsi, l'extension faiblement close calculée ci-dessus est bien une forme résolue du problème ; elle indique, étant donnée une instance quelconque du schéma de droite, comment construire une instance plus petite du schéma de gauche.

Passons à présent à la formalisation de cette discussion.

Proposition 9.4 *Soient $A \Rightarrow \tau \mid C$ et $A' \Rightarrow \tau' \mid C'$ deux schémas de types de domaines disjoints. On suppose qu'il existe un graphe de contraintes D tel que*

- *D est une extension faiblement close de C' à $\mathrm{dom}(C) \cup \mathrm{dom}(C')$;*
- *$D \Vdash C + (A \Rightarrow \tau \leq A' \Rightarrow \tau')$.*

Alors

$$A \Rightarrow \tau \mid C \leq^\forall A' \Rightarrow \tau' \mid C'$$

Démonstration. Soit ρ' une solution de C'. D est une extension faiblement close de C' ; d'après le théorème 9.2, il existe une solution ρ de D qui étend ρ'. Puisque $D \Vdash C + (A \Rightarrow \tau \leq A' \Rightarrow \tau')$, ρ est solution de C, et satisfait

$$\rho(A \Rightarrow \tau) \leq \rho(A' \Rightarrow \tau')$$

que l'on peut également écrire

$$\rho(A \Rightarrow \tau) \leq \rho'(A' \Rightarrow \tau')$$

puisque ρ étend ρ'. Ceci termine la preuve. □

Comme nous l'avons mentionné plus haut, la réciproque de cette proposition est fausse. Cela est dû au fait que la notion d'extension faiblement close fait appel à notre axiomatisation incomplète de l'implication de contraintes. Ainsi, tout exemple démontrant l'incomplétude de celle-ci peut être aisément codé pour produire un exemple démontrant l'incomplétude de la proposition ci-dessus. En d'autres termes, l'algorithme de comparaison polymorphe que nous allons développer dans la prochaine section possède (au moins) les mêmes exemples d'incomplétude que l'algorithme d'implication de contraintes.

Proposition 9.5 *La réciproque de la proposition 9.4 est fausse.*

Démonstration. Supposons donnée une assertion d'implication de contraintes, de la forme $C \Vdash \alpha \leq \beta$, qui soit vraie mais non prouvable, c'est-à-dire que $C \vdash \alpha \leq \beta$ est faux. (De telles assertions existent et ont été exposées dans la preuve de la proposition 8.4.)

Cette assertion d'implication de contraintes peut se coder en termes de comparaisons de schémas de types, comme ceci :

$$\gamma \rightarrow \gamma \leq^{\forall} \alpha \rightarrow \beta \mid C$$

où l'on choisit $\gamma \notin \mathrm{dom}(C)$. Il est clair qu'il n'existe pas d'extension faiblement close de C à $\mathrm{dom}(C) \cup \{\gamma\}$ qui entraîne $\gamma \rightarrow \gamma \leq \alpha \rightarrow \beta$; en effet, d'après la définition 9.3, cela demanderait précisément $C \vdash \alpha \leq \beta$. □

9.4 Algorithme

Grâce aux idées présentées dans la section précédentes, nous pouvons maintenant donner une définition formelle de l'algorithme de comparaison polymorphe entre schémas de types.

Définition 9.5 *Soient $\sigma = A \Rightarrow \tau \mid C$ et $\sigma' = A' \Rightarrow \tau' \mid C'$ deux schémas de types de domaines disjoints. On suppose que C est faiblement clos et que la relation $\leq_C$ est transitive. (On ne pose aucune condition de clôture sur C' ; cependant, l'algorithme obtiendra de meilleurs résultats si C' est clos.) De plus, σ' est supposé simple.*

Par convention, les variables de σ seront notées α, β, ... tandis que celles de σ' seront notées α', β', ...

Un état *de l'algorithme est un couple (D, Q), où D est un graphe de contraintes de domaine $\mathrm{dom}(C) \cup \mathrm{dom}(C')$ et Q est un ensemble de contraintes de la forme $\alpha \leq \alpha'$ ou $\alpha' \leq \alpha$.*

Si $\mathrm{dom}(A) \neq \mathrm{dom}(A')$, l'algorithme échoue immédiatement. Dans le cas contraire, son état initial *est $(C \cup C', A \Rightarrow \tau \leq A' \Rightarrow \tau')$.*

L'algorithme passe *d'un état (D, Q) à un nouvel état comme suit. Si la liste d'attente Q est vide, on signale le succès et on arrête l'algorithme. Sinon, on choisit une contrainte c dans Q et on pose $Q' = Q \setminus \{c\}$. Deux cas se présentent alors, selon la forme de c.*

- *c est de la forme $\alpha \leq \alpha'$. Si $\alpha \leq_D \alpha'$, on passe à l'état (D, Q'). Sinon, on effectue les étapes suivantes :*

- *Soit E défini par* $\leq_E = \leq_D \cup \{(\beta, \alpha') \,;\, \beta \leq_D \alpha\}$, $E^\uparrow = D^\uparrow$ *et* $E^\downarrow = D^\downarrow$.
- *Soit R égal à* $Q' \cup \mathrm{subc}(D^\downarrow(\alpha) \leq D^\downarrow(\alpha'))$.
- *On vérifie que* $\forall \beta' \leq_D \alpha \quad D \vdash \beta' \leq \alpha'$. *En cas de succès, on passe à l'état (E, R) ; dans le cas contraire, on signale l'échec et on arrête l'algorithme.*

- *c est de la forme $\alpha' \leq \alpha$. Si $\alpha' \leq_D \alpha$, on passe à l'état (D, Q'). Sinon, on effectue les étapes suivantes :*
  - *Soit E défini par* $\leq_E = \leq_D \cup \{(\alpha', \beta) \,;\, \alpha \leq_D \beta\}$, $E^\uparrow = D^\uparrow$ *et* $E^\downarrow = D^\downarrow$.
  - *Soit R égal à* $Q' \cup \mathrm{subc}(D^\uparrow(\alpha') \leq D^\uparrow(\alpha))$.
  - *On vérifie que* $\forall \beta' \geq_D \alpha \quad D \vdash \alpha' \leq \beta'$. *En cas de succès, on passe à l'état (E, R) ; dans le cas contraire, on signale l'échec et on arrête l'algorithme.*

A partir de l'état initial, l'algorithme passe d'état en état jusqu'à ce que le succès ou l'échec soit signalé.

Exemple. Illustrons le fonctionnement de l'algorithme. On considère le schéma de types $\sigma = \alpha \to \beta \to \gamma \times \delta \mid C$, où C est le graphe de contraintes défini par $\alpha \leq_C \gamma$, $\beta \leq_C \gamma$, $\alpha \leq_C \delta$ et $\beta \leq_C \delta$. On souhaite vérifier que ce schéma est équivalent au schéma $\sigma' = \epsilon \to \epsilon \to \epsilon \times \epsilon$. (On ignorera le fait que ces schémas ne vérifient pas l'invariant des petits termes ; cela n'affecte pas le principe de l'algorithme.)

L'assertion $\sigma \leq^\forall \sigma'$ est immédiate, puisque σ' s'obtient à partir de σ par une simple substitution. Voyons comment se comporte l'algorithme. Les variables $\alpha \ldots \gamma$ sont considérées ici comme flexibles, tandis qu'ϵ est rigide. L'état initial de l'algorithme est constitué du graphe C (étendu, pour être précis, au domaine $\{\alpha \ldots \epsilon\}$), et de la liste d'attente $\mathrm{subc}(\alpha \to \beta \to \gamma \times \delta \leq \epsilon \to \epsilon \to \epsilon \times \epsilon)$, qui est égale à $\{\gamma \leq \epsilon,\ \delta \leq \epsilon,\ \epsilon \leq \alpha,\ \epsilon \leq \beta\}$. Nous ne détaillerons pas les quatre étapes nécessaires pour retirer ces contraintes de la file d'attente. En résumé, chacune de ces contraintes vient s'ajouter au graphe de contraintes calculé par l'algorithme. Par transitivité, il en résulte des contraintes sur les variables rigides. Par exemple, nous avions au départ $\alpha \leq \gamma$, donc de l'ajout de $\epsilon \leq \alpha$ et $\gamma \leq \epsilon$, on déduit $\epsilon \leq \epsilon$. Cette contrainte entre variables rigides doit être vérifiée à l'aide de l'algorithme d'implication de contraintes, ce qui est immédiat. Aucune erreur n'est donc détectée, et l'algorithme signale un succès. De façon générale, on pourrait prouver que l'algorithme réussit toujours lorsque le schéma de droite s'obtient par substitution à partir de celui de gauche.

Réciproquement, il n'est pas forcément clair, au premier abord, que $\sigma' \leq^\forall \sigma$. Voyons ce que donne l'algorithme. Cette fois, ϵ est la seule variable flexible – les autres sont rigides. Le graphe de contraintes initial est le même, mais la liste d'attente est maintenant $\{\epsilon \leq \gamma,\ \epsilon \leq \delta,\ \alpha \leq \epsilon,\ \beta \leq \epsilon\}$. Encore une fois, ces quatre contraintes viennent s'ajouter au graphe de contraintes. Par transitivité sur ϵ, il en découle des contraintes sur les variables rigides, qui doivent être vérifiées à l'aide de l'algorithme d'implication de contraintes. Par exemple, de $\alpha \leq \epsilon$ et $\epsilon \leq \gamma$, on déduit $\alpha \leq \gamma$. Or, cette contrainte est déjà présente dans C, donc prouvablement impliquée par C. Il en va de même pour les autres contraintes entre variables rigides ainsi obtenues, et l'algorithme réussit.

Commentons ce dernier résultat. L'algorithme a démontré l'assertion de comparaison entre schémas, c'est-à-dire que pour toutes valeurs de $\alpha \ldots \delta$ vérifiant C, il existe une valeur de ϵ vérifiant $\epsilon \to \epsilon \to \epsilon \times \epsilon \leq \alpha \to \beta \to \gamma \times \delta$. Le calcul qu'il a effectué consiste, comme nous l'avons expliqué lorsque nous avons motivé la définition de l'algorithme, à donner une définition explicite d'ϵ en fonction de $\alpha \ldots \delta$. Ici, cette définition est $\alpha \sqcup \beta \leq \epsilon \leq \gamma \sqcap \delta$. Autrement dit, n'importe quelle valeur d'ϵ vérifiant cet encadrement convient. Il en existe au moins une, d'après nos hypothèses sur $\alpha \ldots \delta$; c'est pour le vérifier que nous avons utilisé l'algorithme d'implication de contraintes. Si le calcul avait révélé une contrainte sur $\alpha \ldots \delta$ non prouvable à partir de nos hypothèses, alors l'existence d'ϵ n'aurait pas été garantie, et l'algorithme aurait échoué.

L'équivalence entre σ et σ' autorise une simplification : si on remplace σ par σ' dans une dérivation d'inférence, celle-ci conduira toujours à un schéma de types principal. Reste à savoir si nous sommes capables de détecter cette opportunité en pratique. La réponse, affirmative, sera donnée au chapitre 13 par l'algorithme de minimisation. (Cet exemple y sera repris sous le nom *d'élimination des* 2*-couronnes*.)

L'algorithme est à présent défini. Il reste à démontrer qu'il termine et qu'il est correct.

Proposition 9.6 *L'algorithme de comparaison de schémas termine.*

Démonstration. Si l'algorithme ne termine pas, alors il existe une suite infinie de transitions. Notons que lorsqu'une contrainte est choisie dans la liste d'attente, si elle est déjà dans la relation $\leq_D$, alors la taille de la liste diminue strictement. Si, au contraire, elle n'est pas dans $\leq_D$, alors elle lui est immédiatement ajoutée (d'après la définition de $\leq_E$). Par conséquent, si $(D_n, Q_n)_{n\in\mathbb{N}^+}$ est une suite infinie de transitions, alors la suite $(\leq_{D_n})_{n\in\mathbb{N}^+}$ est croissante et croît strictement un nombre infini de fois. Or, $(\leq_{D_n})_{n\in\mathbb{N}^+}$ est bornée, car tous les D_n ont le même domaine, à savoir $\mathrm{dom}(C) \cup \mathrm{dom}(C')$. Par conséquent, il n'existe pas de suite infinie de transitions, et l'algorithme termine. □

Voici maintenant deux lemmes qui constituent le gros de la preuve de correction.

Lemme 9.7 *On suppose que l'algorithme atteint l'état* (D, Q)*. Alors* D *est une extension de* C' *à* $\mathrm{dom}(C) \cup \mathrm{dom}(C')$*, et une extension de* C *à* $\mathrm{dom}(C) \cup \mathrm{dom}(C')$*.*

Démonstration. Cela est vrai en ce qui concerne l'état initial : puisque σ et σ' sont de domaines disjoints, $C \cup C'$ est une extension de C' (resp. C) à $\mathrm{dom}(C) \cup \mathrm{dom}(C')$.

Supposons à présent que l'algorithme passe de l'état (D, Q) à l'état (E, R), et que D soit une extension de C' (resp. C) à $\mathrm{dom}(C) \cup \mathrm{dom}(C')$. D'après la définition de l'algorithme, la seule différence entre D et E est l'ajout éventuel dans $\leq_E$ de contraintes de la forme $\alpha \leq_E \alpha'$ ou $\alpha' \leq_E \alpha$. Donc, la restriction de E à $\mathrm{dom}(C')$ (resp. $\mathrm{dom}(C)$) coïncide avec la restriction de D à $\mathrm{dom}(C')$ (resp. $\mathrm{dom}(C)$). Il en découle que E est également une extension de C' (resp. C) à $\mathrm{dom}(C) \cup \mathrm{dom}(C')$. □

Lemme 9.8 *Supposons que l'algorithme atteigne un état* (D, Q)*. Alors*

1. $\alpha' \leq_D \beta$ *et* $\beta \leq_D \gamma'$ *impliquent* $D \vdash \alpha' \leq \gamma'$ *;*
2. $\alpha' \leq_D \alpha$ *et* $\alpha \leq_D \beta$ *impliquent* $\alpha' \leq_D \beta$ *;*
3. $\alpha \leq_D \beta$ *et* $\beta \leq_D \beta'$ *impliquent* $\alpha \leq_D \beta'$ *;*
4. $\alpha \leq_D \alpha'$ *implique* $\exists \beta \geq_D \alpha \quad D, Q \vdash D^{\downarrow}(\beta) \leq D^{\downarrow}(\alpha')$ *;*
5. $\alpha' \leq_D \alpha$ *implique* $\exists \beta \leq_D \alpha \quad D, Q \vdash D^{\uparrow}(\alpha') \leq D^{\uparrow}(\beta)$ *;*
6. $D, Q \vdash A \Rightarrow \tau \leq A' \Rightarrow \tau'$*.*

(Notez que dans certaines des assertions d'implication de contraintes ci-dessus, le contenu de Q *est utilisé en tant qu'axiomes.)*

Démonstration. Considérons d'abord l'état initial (D, Q), où $D = C \cup C'$. Puisque C et C' sont de domaines disjoints, $\leq_D$ ne contient aucune contrainte de la forme $\alpha \leq \alpha'$ ou $\alpha' \leq \alpha$, donc les cinq premières conditions sont trivialement vérifiées. Par ailleurs, la liste d'attente est précisément $Q = A \Rightarrow \tau \leq A' \Rightarrow \tau'$; donc l'énoncé est vrai en ce qui concerne l'état initial.

A présent, supposons que le résultat soit vrai pour un état (D, Q) (c'est là l'hypothèse d'induction) et que l'algorithme passe de cet état à un nouvel état (E, R). Supposons que la contrainte c qui a été choisie dans Q est $\alpha \leq \alpha'$; l'autre cas est symétrique. Posons $Q' = Q \setminus \{c\}$.

Si $\alpha \leq_D \alpha'$, alors $(E, R) = (D, Q')$. Il suffit de vérifier que si une contrainte est prouvablement impliquée par (D, Q), alors elle l'est également par (D, Q'). La seule différence est que l'axiome $\alpha \leq \alpha'$ a disparu dans le second cas. Mais cet axiome est dérivable puisque $\alpha \leq_D \alpha'$. Donc le résultat est vrai.

Supposons donc $\alpha \not\leq_D \alpha'$. Nous avons

$$\begin{aligned} \leq_E &= \leq_D \cup \{(\beta, \alpha') \,;\, \beta \leq_D \alpha\} \\ R &= Q' \cup \mathrm{subc}(D^\downarrow(\alpha) \leq D^\downarrow(\alpha')) \end{aligned}$$

Considérons la première condition. Supposons $\gamma' \leq_E \beta$ et $\beta \leq_E \delta'$. Nécessairement, $\gamma' \leq_D \beta$. Si $\beta \leq_D \delta'$, alors $D \vdash \gamma' \leq \delta'$ par hypothèse d'induction, et $E \vdash \gamma' \leq \delta'$ car E contient D. Sinon, nous avons $\delta' = \alpha'$ et $\beta \leq_D \alpha$ par définition de $\leq_E$. Par conséquent, $\gamma' \leq_D \beta \leq_D \alpha$. En appliquant l'hypothèse d'induction, nous obtenons $\gamma' \leq_D \alpha$. A présent, d'après la définition de l'algorithme, l'assertion

$$\forall \beta' \leq_D \alpha \quad D \vdash \beta' \leq \alpha'$$

a été vérifiée avant de passer à l'état (E, R). Il en découle que $D \vdash \gamma' \leq \alpha'$, et $E \vdash \gamma' \leq \alpha'$ car E contient D. La première condition est vérifiée.

La seconde condition est trivialement satisfaite, car aucune des contraintes nouvellement ajoutées n'y entre en jeu.

Considérons la troisième condition. Supposons $\gamma \leq_E \beta$ et $\beta \leq_E \delta'$. Nécessairement, $\gamma \leq_D \beta$. Si $\beta \leq_D \delta'$, nous concluons aisément, comme ci-dessus. Sinon, nous avons $\delta' = \alpha'$ et $\beta \leq_D \alpha$ par définition de $\leq_E$. Cependant, puisque chaque transition ajoute des contraintes « hétérogènes » de la forme $\delta \leq \delta'$ ou $\delta' \leq \delta$, $\gamma \leq_D \beta$ implique $\gamma \leq_C \beta$. De même, $\beta \leq_D \alpha$ implique $\beta \leq_C \alpha$. La relation $\leq_C$ étant transitive, $\gamma \leq_C \alpha$, ce qui implique $\gamma \leq_D \alpha$. Par définition de E, il s'ensuit que $\gamma \leq_E \alpha'$. Nous avons démontré que $\leq_E$ vérifie la troisième condition.

Considérons la quatrième condition. Supposons $\beta \leq_E \delta'$. Le cas $\beta \leq_D \delta'$ est immédiat ; supposons donc $\beta \not\leq_D \delta'$. Alors, par définition de $\leq_E$, $\delta' = \alpha'$ et $\beta \leq_D \alpha$. Par ailleurs, d'après la définition de R, il est immédiat que

$$E, R \vdash E^\downarrow(\alpha) \leq E^\downarrow(\alpha')$$

Enfin, si l'on note que $\beta \leq_E \alpha$, on constate que la quatrième condition est satisfaite.

La cinquième condition est trivialement satisfaite, parce qu'aucune des contraintes nouvellement ajoutées n'y entre en jeu.

La sixième condition est toujours satisfaite, parce que l'« axiome manquant » $\alpha \leq \alpha'$ est dérivable dans E. □

Nous pouvons finalement affirmer que

Théorème 9.3 *L'algorithme de comparaison de schémas de types est correct vis-à-vis de la relation* $\leq^\forall$.

Démonstration. Supposons que l'algorithme signale le succès. Alors il a atteint un certain état $(D, \varnothing)$. D'après le lemme 9.7, D est une extension de C', et de C, à $\mathrm{dom}(C) \cup \mathrm{dom}(C')$. D'après le lemme 9.8, nous avons

1. $\alpha' \leq_D \beta$ et $\beta \leq_D \gamma'$ impliquent $D \vdash \alpha' \leq \gamma'$;
2. $\alpha' \leq_D \alpha$ et $\alpha \leq_D \beta$ impliquent $\alpha' \leq_D \beta$;
3. $\alpha \leq_D \beta$ et $\beta \leq_D \beta'$ impliquent $\alpha \leq_D \beta'$;
4. $\alpha \leq_D \alpha'$ implique $\exists \beta \geq_D \alpha \quad D \vdash D^\downarrow(\beta) \leq D^\downarrow(\alpha')$;
5. $\alpha' \leq_D \alpha$ implique $\exists \beta \leq_D \alpha \quad D \vdash D^\uparrow(\alpha') \leq D^\uparrow(\beta)$;
6. $D \vdash A \Rightarrow \tau \leq A' \Rightarrow \tau'$.

De plus, C étant un graphe de contraintes faiblement clos, nous avons, pour tous $\alpha, \beta \in \mathrm{dom}(C)$:

- $\alpha \leq_C \beta$ implique $C \vdash C^\downarrow(\alpha) \leq C^\downarrow(\beta)$ et $C \vdash C^\uparrow(\alpha) \leq C^\uparrow(\beta)$;

- $C \vdash C^{\downarrow}(\alpha) \leq C^{\uparrow}(\alpha)$.

D étant une extension de C, nous pouvons réécrire cela

- $\alpha \leq_D \beta$ implique $D \vdash D^{\downarrow}(\alpha) \leq D^{\downarrow}(\beta)$ et $D \vdash D^{\uparrow}(\alpha) \leq D^{\uparrow}(\beta)$;
- $D \vdash D^{\downarrow}(\alpha) \leq D^{\uparrow}(\alpha)$.

Les cinq premières assertions, en conjonction avec les deux dernières assertions ci-dessus, signifient exactement que D est une extension faiblement close de C' à $\mathrm{dom}(C) \cup \mathrm{dom}(C')$. Par ailleurs, nous avons $D \vdash A \Rightarrow \tau \leq A' \Rightarrow \tau'$, et $D \vdash C$ puisque D est une extension de C. D'après la proposition 9.4, nous avons donc $\sigma \leq^{\forall} \sigma'$. $\square$

Chapitre 10

Polarités et dépoussiérage

Un schéma de types $A \Rightarrow \tau \mid C$ n'est autre qu'une description (certes approximative) du flot de données à travers une expression. Le contexte A décrit un ensemble de données nécessaire au bon fonctionnement de l'expression, donc représente en quelque sorte un ensemble de points d'entrée. Le corps τ, au contraire, décrit la valeur calculée par l'expression, donc représente un point de sortie. Ils sont reliés entre eux par les contraintes du graphe C. En effet, chaque application de fonction, c'est-à-dire chaque envoi de données d'un créateur à un consommateur, engendre une contrainte. Cela correspond à l'idée que le typage n'est autre qu'une analyse de flot.

Cette vision des choses est intéressante. En particulier, elle suggère de classifier les variables de types d'un schéma selon leur fonctionnalité. Plus précisément, si σ est le schéma de types associé à l'expression e, il serait intéressant de distinguer les variables qui représentent une entrée (c'est-à-dire une donnée attendue par l'expression e) de celles qui représentent une sortie (c'est-à-dire un résultat fourni par e). Nous annoterons chaque variable à l'aide du signe $-$ dans le premier cas, et du signe $+$ dans le second. Bien sûr, il sera possible pour une variable de jouer les deux rôles à la fois, et donc de porter les deux signes. D'autres, au contraire, se révéleront dénuées de tout signe. Ainsi, nous associerons une paire de booléens, nommée *polarité*, à chaque variable.

Quel est le but de ce calcul ? C'est sur lui que repose le processus de *dépoussiérage*, que nous introduirons également dans ce chapitre. Ce processus consiste à identifier, puis éliminer, certaines contraintes superflues au sein du graphe de contraintes. Comme nous allons le voir, le calcul des polarités permet de détecter une large classe de contraintes superflues. Par ailleurs, les polarités jouent un rôle important dans la définition de l'invariant de mono-polarité (cf. chapitre 12) et de l'algorithme de minimisation (cf. chapitre 13). Enfin, elles sont également utilisées pendant la phase de simplification « externe », qui permet d'améliorer la lisibilité d'un schéma de types avant son affichage (cf. section 15.2).

Plutôt que de définir directement la notion de polarité et son application directe, le dépoussiérage, nous procédons en plusieurs étapes, en commençant par exposer une notion plus grossière (section 10.1), que nous raffinerons ensuite à deux reprises (sections 10.2 et 10.3) pour obtenir la version définitive. Enfin, la section 10.4 fournit un point de vue intéressant sur le processus de dépoussiérage, et en prouve ensuite la correction.

10.1 Une définition grossière

Plutôt que de donner d'entrée de jeu la définition de la polarité, nous commençons par introduire une notion d'*accessibilité*. Celle-ci se calcule en annotant chaque variable par un unique booléen, qui indique si elle joue un rôle dans le flot de données, sans distinguer entrées et sorties. Cette notion, plus grossière, a été proposée dans [43]. Actuellement, elle n'est plus nécessaire à notre théorie ; cependant, nous l'utilisons comme point de départ, et montrerons ensuite comment elle peut être améliorée, de deux façons indépendantes, pour obtenir la notion de polarité.

Définition 10.1 *Soit $\sigma = A \Rightarrow \tau \mid C$ un schéma de types clos. L'ensemble des variables* accessibles *de σ, noté $\mathrm{R}(\sigma)$, est le plus petit sous-ensemble R de $\mathrm{dom}(\sigma)$ tel que*

- $\mathrm{fv}(\tau) \subset R$
- $\forall (x : \tau_x) \in A \quad \mathrm{fv}(\tau_x) \subset R$
- $\forall \alpha \in R \quad (\alpha' \leq_C \alpha) \vee (\alpha \leq_C \alpha') \Rightarrow \alpha' \in R$
- $\forall \alpha \in R \quad \mathrm{fv}(C^{\downarrow}(\alpha)) \cup \mathrm{fv}(C^{\uparrow}(\alpha)) \subset R$

L'intuition qui sous-tend cette définition est fort simple. Notre but est de repérer toutes les variables susceptibles de jouer un rôle dans le comportement de σ. Par comportement de σ, nous entendons la façon dont il se combinera, au cours de la dérivation de typage, avec d'autres schémas, pour finalement mener à un succès ou à un échec de la dérivation. En observant les règles de typage, on remarque que les contraintes explicitement créées par celles-ci ne pourront mettre en jeu que « l'interface » de σ, c'est-à-dire son contexte A et son corps τ. Donc, pour commencer, nous marquons toutes les variables de A et de τ comme étant accessibles.

Cependant, rappelons que les règles de typage demandent que tous les schémas mis en jeu aient une dénotation non vide, c'est-à-dire que leur graphe de contraintes admette une solution. Pour vérifier cette condition, on effectuera un calcul de clôture, qui combinera les nouvelles contraintes avec celles déjà présentes dans C, par transitivité et par décomposition structurelle. Toute variable de σ susceptible de jouer un rôle dans ce calcul doit donc également être considérée comme accessible. Cela se traduit par les deux dernières conditions ci-dessus, qui expriment que toute borne d'une variable accessible est elle-même accessible.

Pour conclure cette explication, on peut dire que le calcul d'accessibilité est essentiellement une simulation d'un futur calcul de clôture, qui marque toutes les variables susceptibles d'être mises en jeu.

A partir de cette définition, on peut aisément imaginer une méthode de simplification, connue sous le nom d'*élimination des contraintes inaccessibles* dans [43]. Si une variable est inaccessible, nous avons la certitude qu'elle n'interviendra dans aucun calcul de clôture à l'avenir. Par conséquent, toute information concernant cette variable est superflue, et peut être abandonnée sans modifier le comportement du schéma de types.

Nous ne chercherons pas à prouver ce résultat ici, puisqu'il sera généralisé plus loin. Donnons plutôt un exemple, que nous retrouverons au cours de ce qui suit.

Exemple. Considérons le schéma de types $\sigma = \alpha \rightarrow \gamma \mid C$, où C est le graphe de contraintes défini par

- $\alpha \leq_C \beta \leq_C \gamma$;
- $C^{\uparrow}(\alpha) = C^{\uparrow}(\beta) = C^{\uparrow}(\gamma) = \delta \rightarrow \epsilon$;
- $C^{\downarrow}(\lambda) = \alpha \rightarrow \gamma$;
- $C^{\downarrow}(\epsilon) = \top$.

Pour calculer $\mathrm{R}(\sigma)$, nous commençons par marquer α et γ comme accessibles. Ensuite, nous laissons les marques se propager (toute borne d'une variable accessible devenant accessible) jusqu'à ce que plus aucune variable ne puisse être marquée. Nous obtenons

$$\mathrm{R}(\sigma) = \{\alpha, \beta, \gamma, \delta, \epsilon\}$$

Ainsi, λ est la seule variable inaccessible. Si nous retirons les contraintes inaccessibles, nous obtenons un nouveau schéma $\alpha \rightarrow \gamma \mid D$, où D est défini par

- $\alpha \leq_D \beta \leq_D \gamma$;
- $D^{\uparrow}(\alpha) = D^{\uparrow}(\beta) = D^{\uparrow}(\gamma) = \delta \rightarrow \epsilon$;

– $D^{\downarrow}(\epsilon) = \top$.

La contrainte concernant λ a été abandonnée. Il est facile de comprendre pourquoi cela est correct. Cette contrainte ne restreignait pas les valeurs possibles de α ou γ, donc n'avait aucun effet sur la dénotation du schéma de types σ.

Pour résumer, l'analyse d'accessibilité est une simulation d'une hypothétique phase de clôture à venir, et l'élimination des contraintes inaccessibles en est l'application naturelle. Dans [49], Trifonov et Smith raffinent la notion d'accessibilité, pour obtenir la notion de polarité. L'application naturelle en est une méthode de simplification plus puissante, nommée *dépoussiérage* (*garbage collection* en anglais). Nous allons à présent expliquer les améliorations qu'ils ont apportées. Elles sont au nombre de deux, et sont indépendantes l'une de l'autre. Nous les considérons donc isolément.

10.2 Première amélioration

La première amélioration consiste à rendre l'analyse « directionnelle ». Nous l'avons vu, l'analyse d'accessibilité simule un calcul de clôture, et marque les variables qui recevront éventuellement de nouvelles contraintes. Mais la direction de ces contraintes n'est pas prise en compte, c'est-à-dire que l'analyse ne distingue pas les nouvelles bornes supérieures des nouvelles bornes inférieures. Cependant, cette distinction est fondamentale.

Nous raffinons donc l'analyse en annotant chaque variable par deux booléens. Le premier indique si la variable est susceptible de recevoir de nouvelles bornes inférieures à l'avenir, et le second si elle risque de recevoir de nouvelles bornes supérieures. Par convention, la variable sera dite *négative* (resp. *positive*), et annotée par le signe $-$ (resp. $+$), si le premier (resp. second) booléen a la valeur « vrai ».

Le calcul de ces marques se fait, comme précédemment, par point fixe. Nous avons mentionné précédemment que lorsque de nouvelles contraintes viennent se greffer sur un schéma σ, elles concernent nécessairement son contexte A et son corps τ. Effectuons un examen plus précis des règles de typage. Pour amener de nouvelles contraintes sur le corps τ, il faut passer l'expression e correspondant à σ à une fonction. Mais la contrainte ainsi créée constitue toujours une borne *supérieure* pour τ. Symétriquement, pour amener de nouvelles contraintes sur un élément x du contexte A, il faut λ-abstraire l'expression e par rapport à x et appliquer la fonction ainsi obtenue à un argument. Mais la contrainte ainsi obtenue constitue toujours une borne *inférieure* pour $A(x)$. Ainsi, au début du calcul de point fixe, il suffit de marquer τ positif et A négatif.

Pendant le calcul de point fixe, les marques se propagent, comme précédemment, le long des contraintes. Cependant, les règles de propagation sont plus fines. Si une variable est positive, alors elle risque de recevoir une nouvelle borne supérieure ; donc, par transitivité, toutes ses bornes inférieures sont dans le même cas, et doivent également être marquées positives. Ainsi, les signes $+$ se propagent vers le bas, et, symétriquement, les signes $-$ vers le haut.

Enfin, notons que si τ est un terme construit, lui ajouter une borne supérieure revient en fait à ajouter une borne supérieure aux variables de $\mathrm{fv}^+(\tau)$ et une borne inférieure aux variables de $\mathrm{fv}^-(\tau)$. (C'est là une conséquence des règles de décomposition structurelle.) Par conséquent, lorsque nous parlons de marquer τ positif, il faut en fait marquer $\mathrm{fv}^+(\tau)$ positif et $\mathrm{fv}^-(\tau)$ négatif.

Donnons à présent la définition de cette analyse. (Rappelons qu'il ne s'agit pas là de la définition définitive de la polarité ; il nous reste à exposer une seconde amélioration dans la section 10.3.)

Définition 10.2 *Soit $\sigma = A \Rightarrow \tau \mid C$ un schéma de types clos. L'ensemble des variables* positives *de σ, et l'ensemble des variables* négatives *de σ, notés respectivement* $\mathrm{dom}^+(\sigma)$ *et* $\mathrm{dom}^-(\sigma)$*, sont les plus petits sous-ensembles P et N de* $\mathrm{dom}(\sigma)$ *tels que*

– $\mathrm{fv}^+(\tau) \subset P \wedge \mathrm{fv}^-(\tau) \subset N$

- $\forall (x : \tau_x) \in A \quad \mathrm{fv}^+(\tau_x) \subset N \wedge \mathrm{fv}^-(\tau_x) \subset P$
- $\forall \alpha \in P \quad \alpha' \leq_C \alpha \Rightarrow \alpha' \in P$
- $\forall \alpha \in N \quad \alpha \leq_C \alpha' \Rightarrow \alpha' \in N$
- $\forall \alpha \in P \quad \mathrm{fv}^+(C^\downarrow(\alpha)) \subset P \wedge \mathrm{fv}^-(C^\downarrow(\alpha)) \subset N$
- $\forall \alpha \in N \quad \mathrm{fv}^+(C^\uparrow(\alpha)) \subset N \wedge \mathrm{fv}^-(C^\uparrow(\alpha)) \subset P$

Cette analyse est plus fine que la précédente, puisque les marques d'accessibilité se propagent dans les deux directions, tandis que les signes $+$ ne se propagent que vers le bas et les signes $-$ vers le haut. Par ailleurs, conformément à ce que nous avons annoncé, les marques portées par une variable indiquent son rôle en tant qu'entrée ou sortie. En effet, si une variable α représente le type d'un résultat fourni par l'expression e, alors c'est qu'il est possible de construire un programme P, contenant e, dans lequel ce résultat est utilisé. Or, dans un système basé sur le sous-typage, toute utilisation d'une donnée engendre une contrainte $\tau_1 \leq \tau_2$, où τ_1 est le type de la donnée fournie, et τ_2 est le type attendu par le consommateur. Donc, le typage de l'expression P engendre une nouvelle borne supérieure pour α. La variable α est donc nécessairement positive. Nous avons montré, informellement, que les polarités fournissent une approximation conservative du rôle des variables : si une variable ne porte pas le signe $+$ (resp. $-$), alors elle ne joue aucun rôle en sortie (resp. en entrée).

Nous pouvons à présent appliquer cette nouvelle analyse pour en tirer une méthode de simplification. Le principe est le même que précédemment : une contrainte ne doit être gardée que si elle a une chance de jouer un rôle à l'avenir. Considérons par exemple une variable non négative α. Elle ne recevra aucune nouvelle borne inférieure à l'avenir. Par conséquent, il n'y aucune raison de conserver la borne supérieure de α. En effet, la seule façon, pour la contrainte $\alpha \leq C^\uparrow(\alpha)$, d'intervenir dans un futur calcul de clôture est d'être combinée, par transitivité, avec une nouvelle borne inférieure de α. Symétriquement, nous pouvons oublier la borne inférieure de toute variable non positive. Enfin, une contrainte reliant deux variables $\alpha \leq \beta$ représente une borne supérieure pour α et une borne inférieure pour β, donc ne peut être jetée que si α est non négative et β non positive.

Nous obtenons donc une méthode de simplification plus fine que l'élimination des contraintes inaccessibles. Reprenons l'exemple introduit dans la section précédente.

Exemple. Pour calculer $\mathrm{dom}^+(\sigma)$ et $\mathrm{dom}^-(\sigma)$, nous commençons par marquer $\alpha \to \gamma$ positif, c'est-à-dire α négative et γ positive. Ensuite, nous laissons les marques se propager (les signes $+$ vers le bas, les signes $-$ vers le haut). Nous obtenons

$$\mathrm{dom}^+(\sigma) = \{\alpha, \beta, \gamma, \delta\}$$
$$\mathrm{dom}^-(\sigma) = \{\alpha, \beta, \gamma, \epsilon\}$$

Ainsi, λ est la seule variable dénuée de toute marque. Une fois encore, nous oublierons donc toute information à son propos. Mais l'analyse remarque également que ϵ est négative, donc que sa borne inférieure n'a pas de signification, et peut être oubliée. Ainsi, la simplification donne un nouveau schéma de types $\alpha \to \gamma \mid E$, où E est défini par

- $\alpha \leq_E \beta \leq_E \gamma$;
- $E^\uparrow(\alpha) = E^\uparrow(\beta) = E^\uparrow(\gamma) = \delta \to \epsilon$.

10.3 Seconde amélioration et version définitive

Nous allons à présent exposer une deuxième amélioration de la notion d'accessibilité. Elle est entièrement indépendante de la précédente, et un peu plus délicate, mais d'importance capitale. Elle consiste à ignorer totalement les contraintes entre variables durant le calcul des polarités.

Rappelons que l'idée fondamentale qui sous-tend la notion d'accessibilité est une simulation d'un futur calcul de clôture ; calcul qui aura effectivement lieu lorsque l'expression actuelle e sera utilisée dans un programme P. Jusqu'ici, nous avons cherché à déterminer quelles variables seraient mises en jeu dans ce calcul. Cependant, au final, notre unique intérêt est de déterminer si P est typable, c'est-à-dire de savoir si le calcul de clôture peut échouer. Donc, nous pouvons restreindre notre attention aux seules variables susceptibles de provoquer un échec. (De façon informelle, on dira qu'une variable α provoque un échec quand la contrainte $C^{\downarrow}(\alpha) \leq C^{\uparrow}(\alpha)$ ne peut être décomposée parce qu'elle est insoluble.)

Considérons la variable β de l'exemple ci-dessus. Elle est à la fois positive et négative. Si elle est positive, c'est uniquement parce que γ est positive, et que le signe $+$ s'est propagé le long de la contrainte $\beta \leq \gamma$. Ainsi, β peut recevoir de nouvelles bornes supérieures à l'avenir ; mais à chaque fois que β reçoit une nouvelle borne, nous savons que γ a également reçu cette même borne. Supposons que β cause un échec après avoir reçu une nouvelle borne τ. Alors $C^{\downarrow}(\beta) \leq \tau$ est insoluble. Mais $C^{\downarrow}(\gamma) \leq \tau$ l'est alors également, puisque $C^{\downarrow}(\gamma)$ contient $C^{\downarrow}(\beta)$. γ cause donc également un échec. Nous pouvons donc ne pas tenir compte des échecs provoqués par β, puisqu'ils seront également provoqués par γ.

On peut dire que le signe $+$ porté par β joue deux rôles. Le premier est de signaler que β risque de provoquer un échec. Nous avons vu que ce fait peut être ignoré. Le second est de se propager à toutes les bornes inférieures de β. Or, puisque $\beta \leq \gamma$, celles-ci sont également bornes inférieures de γ, qui est positif, et auront donc déjà reçu le signe $+$ pour cette raison. Ainsi, cette deuxième fonction est également inutile.

Concluons cette discussion informelle. Nous avons vérifié que le signe $+$ porté par β n'apporte en fait aucune information. Par conséquent, il est possible de ne pas marquer β positif. Nous avons donc découvert qu'il n'est pas nécessaire que les polarités se propagent d'une variable à l'autre, mais seulement d'une variable vers sa borne construite.

Comme nous allons le voir, cette idée est fondamentale, car la méthode de simplification qui en découle sera très puissante. Puisque β ne porte plus aucun signe, toutes les références à cette variable vont être éliminées. C'est excellent – β était en effet superflue, puisqu'elle n'était qu'une variable intermédiaire entre α et γ, et ne portait aucune information propre.

Nous pouvons à présent intégrer cette seconde amélioration et donner la définition définitive de la polarité. Cette définition, ainsi que la méthode de *dépoussiérage* qui en résultera, ont été proposées par Trifonov et Smith [49]. Aiken et Fähndrich [4] présentent également cette notion, avec et sans la seconde amélioration (à propos de laquelle ils parlent de « détection des variables réellement intermédiaires »).

Par ailleurs, nous affaiblissons nos hypothèses sur le schéma de types ; la clôture au sens fort n'est pas nécessaire.

Définition 10.3 *Soit $\sigma = A \Rightarrow \tau \mid C$ un schéma de types faiblement clos. L'ensemble des variables* positives *de σ, et l'ensemble des variables* négatives *de σ, notés respectivement $\mathrm{dom}^+(\sigma)$ et $\mathrm{dom}^-(\sigma)$, sont les plus petits sous-ensembles P et N de $\mathrm{dom}(\sigma)$ tels que*

- $\mathrm{fv}^+(\tau) \subset P \wedge \mathrm{fv}^-(\tau) \subset N$
- $\forall (x : \tau_x) \in A \quad \mathrm{fv}^+(\tau_x) \subset N \wedge \mathrm{fv}^-(\tau_x) \subset P$
- $\forall \alpha \in P \quad \mathrm{fv}^+(C^{\downarrow}(\alpha)) \subset P \wedge \mathrm{fv}^-(C^{\downarrow}(\alpha)) \subset N$
- $\forall \alpha \in N \quad \mathrm{fv}^+(C^{\uparrow}(\alpha)) \subset N \wedge \mathrm{fv}^-(C^{\uparrow}(\alpha)) \subset P$

Une variable α est dite positive *si $\alpha \in \mathrm{dom}^+(\sigma)$, et* négative *si $\alpha \in \mathrm{dom}^-(\sigma)$. Elle est dite* bipolaire *si elle est positive et négative, et* neutre *si elle n'est ni positive ni négative.*

De plus, on définit la fonction polarité_σ*, qui à toute variable $\alpha \in \mathrm{dom}(\sigma)$ associe un sous-ensemble de $\{+, -\}$, par*

$$\epsilon \in \text{polarité}_\sigma(\alpha) \iff \alpha \in \mathrm{dom}^\epsilon(\sigma)$$

Exemple. Continuons à étudier l'exemple introduit dans les sections précédentes. Pour calculer $\mathrm{dom}^+(\sigma)$ et $\mathrm{dom}^-(\sigma)$, nous marquons d'abord α négative et γ positive. Ensuite, nous effectuons le calcul de point fixe. Cette fois, β ne reçoit aucune marque, puisque les marques ne suivent plus les liens entre variables. Pour la même raison, γ ne devient pas négative, et α ne devient pas positive. Nous obtenons

$$\mathrm{dom}^+(\sigma) = \{\gamma, \delta\}$$
$$\mathrm{dom}^-(\sigma) = \{\alpha, \epsilon\}$$

Ce résultat est nettement plus fin que les précédents. Nous allons à présent expliquer comment effectuer le dépoussiérage à partir de ces données.

Dans la section 10.2, nous avons expliqué que la borne supérieure (resp. inférieure) construite d'une variable non négative (resp. non positive) peut être jetée, car sa présence ne peut pas provoquer d'échec pendant les calculs de clôture à venir. Nous avons également indiqué dans quelles conditions une contrainte de la forme $\alpha \leq \beta$ pouvait être éliminée ; cependant, notre deuxième amélioration nous permet de raffiner ce critère.

Supposons, par exemple, β non positive. Alors, si β reçoit une nouvelle borne supérieure à l'avenir, c'est qu'il existe un certain γ tel que $\beta \leq \gamma$ et γ reçoit la même borne supérieure. Dans ce cas, nous avons, par transitivité, $\alpha \leq \gamma$; nous pouvons donc déduire qu'α reçoit cette nouvelle borne, sans utiliser la contrainte $\alpha \leq \beta$. Ainsi, cette contrainte ne joue aucun rôle essentiel dans le calcul de clôture, et elle peut être éliminée. Dans le cas symétrique où α est non négative, nous obtenons la même conclusion. Par conséquent, une contrainte $\alpha \leq \beta$ ne doit être conservée que si α est négative et β positive – un résultat assez puissant.

Exemple. Concluons à présent l'analyse de l'exemple utilisé au cours de cette discussion. Comme expliqué plus haut, β ne porte plus aucun signe, donc les contraintes $\alpha^- \leq \beta$ et $\beta \leq \gamma^+$ peuvent être éliminées. En fait, toute référence à β disparaît. Le lien entre α et γ n'est pas perdu, cependant ; puisque $\leq_C$ est fermé par transitivité, la contrainte $\alpha^- \leq \gamma^+$ était présente et demeure.

Comme précédemment, λ disparaît entièrement, et la borne inférieure de ϵ est jetée. De plus, cette fois, γ est non négative, donc sa borne supérieure est également éliminée. Reste un schéma de types $\alpha \rightarrow \gamma \mid F$, où F est défini par

- $\alpha \leq_F \gamma$;
- $F^\uparrow(\alpha) = \delta \rightarrow \epsilon$.

Ce schéma de types est nettement plus simple que σ. Le dépoussiérage a donc permis d'éliminer une grande quantité d'informations superflues à un faible coût. En particulier, il détruit les « variables réellement intermédiaires », c'est-à-dire les variables comme β, qui ne servent qu'à former un lien entre d'autres variables. Cela est fondamental, car de tels liens sont engendrés en grande quantité par les règles d'inférence et par l'algorithme de canonisation (introduit au chapitre 11).

Passons à présent à la définition formelle du dépoussiérage. Pour plus de clarté, nous commençons par définir les conditions que doit vérifier un schéma de types pour que ce procédé lui soit applicable.

Définition 10.4 *Un schéma de types $\sigma = A \Rightarrow \tau \mid C$ est* dépoussiérable *ssi*

- *C est faiblement clos, et $\leq_C$ est transitive ;*
- *σ est simple.*

En fait, la seconde condition ci-dessus apparaît pour une raison purement technique : la preuve de l'algorithme de dépoussiérage repose sur l'algorithme de comparaison polymorphe entre schémas, lequel n'a été mis au point que dans le cas des schémas simples. Il serait possible de retirer cette condition, mais cela demanderait des preuves plus lourdes. En pratique, cette restriction impose d'effectuer la canonisation avant le dépoussiérage.

Ceci posé, passons à la définition même :

Définition 10.5 *Soit σ un schéma de types dépoussiérable. L'image de σ par le* dépoussiérage*, notée* $\mathrm{GC}(\sigma)$*, est le schéma de types $A \Rightarrow \tau \mid D$, de domaine* $\mathrm{dom}^+(\sigma) \cup \mathrm{dom}^-(\sigma)$*, où le graphe de contraintes D est défini comme suit :*

- $\alpha \leq_D \beta$ *ssi* $\alpha \leq_C \beta$, $\alpha \in \mathrm{dom}^-(\sigma)$ *et* $\beta \in \mathrm{dom}^+(\sigma)$ *;*
- $D^{\downarrow}(\alpha)$ *est égal à* $C^{\downarrow}(\alpha)$ *si* $\alpha \in \mathrm{dom}^+(\sigma)$*, et à* $\perp$ *sinon ;*
- $D^{\uparrow}(\alpha)$ *est égal à* $C^{\uparrow}(\alpha)$ *si* $\alpha \in \mathrm{dom}^-(\sigma)$*, et à* $\top$ *sinon.*

10.4 Correction

Il reste à prouver la correction de cette transformation. Notre but est de montrer que les schémas σ et $\mathrm{GC}(\sigma)$ ont la même dénotation, c'est-à-dire que $\sigma =^{\forall} \mathrm{GC}(\sigma)$. Pour cela, nous allons utiliser l'algorithme développé au chapitre 9.

On peut formuler une remarque très intéressante à propos de cette preuve, qui permet à la fois de mieux comprendre la preuve et de donner une définition alternative, courte et élégante, du dépoussiérage. Imaginons que nous utilisions l'algorithme de comparaison de schémas pour déterminer si $\sigma \leq^{\forall} \sigma$. (*A priori*, il n'est pas certain que l'algorithme réponde positivement, puisqu'il est incomplet.) Si on examine le comportement de l'algorithme dans une telle situation, on s'aperçoit que la réponse est toujours positive ; mais la remarque la plus intéressante, et la plus inattendue, est qu'il est capable de répondre positivement sans utiliser toutes les contraintes présentes dans le membre droit. On en déduit immédiatement un procédé de simplification : soit σ' un schéma identique à σ, excepté que toutes les contraintes inutilisées par l'algorithme dans l'expérience précédente ont été retirées. Alors, nous avons toujours $\sigma \leq^{\forall} \sigma'$. Par ailleurs, $\sigma' \leq^{\forall} \sigma$ est immédiat, puisque σ' contient moins de contraintes. Les deux schémas sont donc équivalents, et nous avons là un procédé de simplification correct, qui n'est autre que le dépoussiérage.

Cette remarque est intéressante, car elle fournit une définition fort succincte du dépoussiérage, tout en le reliant à l'algorithme de comparaison de schémas. Elle justifie le dépoussiérage de manière « sémantique », en prouvant qu'il ne modifie pas la dénotation du schéma de types concerné. Les commentaires que nous avons fait dans les sections précédentes, au contraire, le justifient de façon plus « opérationnelle », en prouvant que le fait de jeter ces contraintes ne modifiera pas le comportement observable du schéma.

Théorème 10.1 *Soit σ un schéma de types dépoussiérable. Alors*

$$\sigma =^{\forall} \mathrm{GC}(\sigma)$$

Démonstration. Posons $\sigma' = \mathrm{GC}(\sigma) = A \Rightarrow \tau \mid D$ comme dans la définition 10.5. Alors, puisque D contient moins de contraintes que C, il est clair que $C \Vdash D$. Il en découle que $\sigma' \leq^{\forall} \sigma$.

Réciproquement, nous souhaitons prouver $\sigma \leq^{\forall} \sigma'$. Pour cela, nous allons simuler une exécution de l'algorithme défini au chapitre 9 et vérifier qu'elle se termine par un succès.

Puisque l'algorithme travaille avec deux schémas de types de domaines disjoints, effectuons d'abord une étape d'α-conversion sur σ'. Pour chaque $\alpha \in \mathrm{dom}(\sigma)$, on choisit une variable fraîche $\phi(\alpha)$.

Nous devons vérifier que σ et $\phi(\sigma')$ sont des arguments acceptables pour l'algorithme de comparaison. Ils sont de domaines disjoints ; C est faiblement clos, et $\leq_C$ est transitive ; $\phi(\sigma')$ est simple.

Nous allons à présent montrer que si l'algorithme atteint un état (D_n, Q_n), alors

$$\begin{aligned}\leq_{D_n} \subset \leq_C \cup \leq_{\phi(D)} & \cup \{\beta \leq \phi(\alpha) ; \alpha \in \mathrm{dom}^+(\sigma) \wedge \beta \leq_C \alpha\} \\ & \cup \{\phi(\alpha) \leq \beta ; \alpha \in \mathrm{dom}^-(\sigma) \wedge \alpha \leq_C \beta\}\end{aligned}$$

et

$$\begin{aligned} Q_n \subset &\{\alpha \leq \phi(\alpha)\,;\, \alpha \in \mathrm{dom}^+(\sigma)\} \\ \cup &\{\phi(\alpha) \leq \alpha\,;\, \alpha \in \mathrm{dom}^-(\sigma)\} \end{aligned}$$

Cela est vrai de l'état initial (D_0, Q_0), parce que $\leq_{D_0} = \leq_C \cup \leq_{\phi(D)}$ et $Q_0 = (A \Rightarrow \tau) \leq (\phi(A) \Rightarrow \phi(\tau))$. Supposons que cela soit vrai pour un état (D_n, Q_n). Supposons que l'algorithme atteigne l'état suivant (D_{n+1}, Q_{n+1}) en sélectionnant une contrainte c. D'après l'hypothèse d'induction sur Q_n, c est de la forme $\alpha \leq \phi(\alpha)$, où $\alpha \in \mathrm{dom}^+(\sigma)$. (L'autre cas est symétrique ; nous ne le traitons donc pas explicitement.) Si cette contrainte est déjà dans $\leq_{D_n}$, alors $(D_{n+1}, Q_{n+1}) = (D_n, Q_n \setminus \{c\})$ et le résultat est vérifié. Sinon, nous avons $\leq_{D_{n+1}} = \leq_{D_n} \cup \{(\beta, \phi(\alpha))\,;\, \beta \leq_{D_n} \alpha\}$. $\beta \leq_{D_n} \alpha$ implique $\beta \leq_C \alpha$; puisque $\alpha \in \mathrm{dom}^+(\sigma)$, $\leq_{D_{n+1}}$ est de la forme attendue. De plus, nous avons $Q_{n+1} \subset Q_n \cup \mathrm{subc}(C^{\downarrow}(\alpha) \leq (\phi(D))^{\downarrow}(\phi(\alpha)))$. Parce que $\alpha \in \mathrm{dom}^+(\sigma)$, α a la même borne inférieure dans D et dans C, donc $(\phi(D))^{\downarrow}(\phi(\alpha)) = \phi(C^{\downarrow}(\alpha))$. Ainsi, les contraintes ajoutées à la queue sont les sous-contraintes de $C^{\downarrow}(\alpha) \leq \phi(C^{\downarrow}(\alpha))$. D'après la définition de dom^+ et dom^-, elles sont de la forme $\beta \leq \phi(\beta)$ (resp. $\phi(\beta) \leq \beta$) où $\beta \in \mathrm{dom}^+(\sigma)$ (resp. $\beta \in \mathrm{dom}^-(\sigma)$). Ainsi, le résultat annoncé au début de ce paragraphe est vérifié.

Nous pouvons à présent vérifier que l'algorithme n'échoue pas. Il ne peut échouer que si l'une des assertions d'implication de contraintes qui doivent être vérifiées lors des changements d'états n'est pas dérivable. Considérons une tentative de passer de l'état (D_n, Q_n) à un nouvel état. Supposons que la contrainte c choisie dans Q_n est de la forme $\alpha \leq \phi(\alpha)$, où $\alpha \in \mathrm{dom}^+(\sigma)$. (L'autre cas est symétrique, donc nous ne le traiterons pas explicitement.) Les assertions à vérifier sont de la forme $D_n \vdash \phi(\beta) \leq \phi(\alpha)$, où $\phi(\beta) \leq_{D_n} \alpha$. D'après le résultat prouvé au paragraphe précédent, $\phi(\beta) \leq_{D_n} \alpha$ implique $\beta \in \mathrm{dom}^-(\sigma)$ et $\beta \leq_C \alpha$. Par définition du dépoussiérage, $\beta \in \mathrm{dom}^-(\sigma) \wedge \alpha \in \mathrm{dom}^+(\sigma) \wedge \beta \leq_C \alpha$ implique $\beta \leq_D \alpha$. Il en découle que $\phi(\beta) \leq_{\phi(D)} \phi(\alpha)$. Finalement, puisque $\leq_{D_n}$ contient $\leq_{\phi(D)}$, l'assertion $D_n \vdash \phi(\beta) \leq \phi(\alpha)$ est prouvable par simple utilisation de (TAUTO).

Nous avons vérifié que l'algorithme ne peut échouer. Donc, il signale un succès, et l'assertion de comparaison entre schémas est vérifiée. □

Puisque le dépoussiérage ne fait rien d'autre que jeter des contraintes, le schéma de types qu'il produit est simple, et vérifie toujours l'invariant des petits termes. De plus, le dépoussiérage conserve les polarités, comme le montre la proposition ci-dessous.

Proposition 10.1 *Soit σ un schéma de types dépoussiérable. Alors*

$$\begin{aligned} \mathrm{dom}^+(\mathrm{GC}(\sigma)) &= \mathrm{dom}^+(\sigma) \\ \mathrm{dom}^-(\mathrm{GC}(\sigma)) &= \mathrm{dom}^-(\sigma) \end{aligned}$$

Démonstration. Rappelons que la définition 10.3 définit les ensembles des variables positives et négatives d'un schéma de types comme les plus petits ensembles vérifiant une certain conjonction de conditions. Si nous comparons la définition de $\mathrm{dom}^+(\sigma)$ et $\mathrm{dom}^-(\sigma)$, d'un côté, avec celle de $\mathrm{dom}^+(\mathrm{GC}(\sigma))$ et $\mathrm{dom}^-(\mathrm{GC}(\sigma))$, de l'autre, nous constatons qu'elles sont identiques. Le résultat en découle. □

Enfin, il serait intéressant de savoir si le schéma $\mathrm{GC}(\sigma)$ vérifie une propriété de clôture. Ce problème sera abordé au chapitre 12.

Chapitre 11

Canonisation

Jusqu'ici, nous avons travaillé avec des schémas de types dans lesquels pouvaient apparaître les constructeurs $\sqcup$ et $\sqcap$. Comme l'explique la section 2.1, ceux-ci constituent une façon de coder des conjonctions de contraintes. Or, plusieurs algorithmes avancés, en particulier l'algorithme de minimisation (cf. chapitre 13), s'accommodent mal de leur présence. Il est donc souhaitable de les éliminer. Par ailleurs, cette élimination offre de nouvelles opportunités au processus de dépoussiérage, et constitue donc une simplification en soi.

Le but de ce chapitre est donc de montrer que cette élimination est possible sans perte d'expressivité, et de définir l'algorithme qui l'effectue. Ce procédé est nommé *canonisation*, d'après Trifonov et Smith [49]. Leur description de l'algorithme n'est pas entièrement satisfaisante, parce qu'elle ne préserve pas la propriété de clôture. D'un point de vue pratique, il est alors nécessaire de faire suivre la canonisation d'une nouvelle phase de clôture. Pour éviter cette perte d'efficacité, nous donnons ici une description plus précise de l'algorithme, et montrons qu'il préserve la propriété de *clôture simple*. Ensuite, nous montrons que la plupart des contraintes produites pour obtenir la clôture seront en fait éliminées par le dépoussiérage. Nous pouvons alors définir un algorithme combinant canonisation et dépoussiérage, qui évite d'engendrer ces contraintes superflues.

Par ailleurs, nous verrons que l'algorithme de canonisation peut également être utilisé pour éliminer les variables bipolaires d'un schéma de types, de façon à ce qu'il respecte *l'invariant de mono-polarité* décrit au chapitre 12. Les deux tâches peuvent ainsi être effectuées en une seule passe par l'algorithme.

Ce chapitre commence par un court exposé du principe de la canonisation (section 11.1). Vient ensuite une section de préliminaires techniques (section 11.2), au cours de laquelle nous exposons une nouvelle notion de clôture, adaptée à la preuve du processus de canonisation. Cela nous permet de donner la définition d'une première notion, appelée *canonisation brute* (section 11.3). Nous simulons ensuite une exécution de l'algorithme de dépoussiérage sur le schéma de types produit par la canonisation brute, et définissons la *canonisation* comme la combinaison de ces deux phases (section 11.4). Enfin, la section 11.5 évoque un hypothétique algorithme de canonisation incrémental.

11.1 Principe

Le principe de la canonisation est très simple ; aussi ne le décrirons-nous que brièvement. L'idée est de remplacer l'expression $\alpha \sqcap \beta$ par une variable fraîche γ, accompagnée des contraintes $\gamma \leq \alpha$ et $\gamma \leq \beta$. Ainsi, toutes les bornes supérieures de α et de β deviendront, par transitivité, bornes supérieures de γ. γ aura donc le même comportement que $\alpha \sqcap \beta$, au sens où les contraintes $\tau \leq \gamma$ et $\tau \leq \alpha \sqcap \beta$ auront les mêmes conséquences vis-à-vis de la clôture. (Les contraintes de la forme $\alpha \sqcap \beta \leq \tau$ ne nous préoccupent pas, puisque $\sqcap$ ne peut pas apparaître dans cette position.) Symétriquement, $\alpha \sqcup \beta$ sera remplacée par une variable fraîche γ accompagnée des contraintes $\alpha \leq \gamma$ et $\alpha \leq \beta$.

L'algorithme de canonisation doit préserver une propriété de clôture, de façon à pouvoir être composé avec la phase de simplification suivante, le dépoussiérage. Il faut donc ajouter non seulement les contraintes mentionnées ci-dessus, mais leurs conséquences par transitivité et par décomposition structurelle.

Si on se contentait d'introduire les contraintes ci-dessus, puis de calculer leurs conséquences à l'aide des règles de clôture habituelles, celles-ci risqueraient de réintroduire des constructions $\sqcup$ ou $\sqcap$. Par exemple, supposons que F soit un constructeur de types unaire covariant. Supposons que la borne supérieure de α (resp. β) soit $F\,\alpha$ (resp. $F\,\beta$). Alors, lorsqu'on élimine $\alpha \sqcap \beta$, on introduit les contraintes $\gamma \leq \alpha$ et $\gamma \leq \beta$. Par transitivité, la borne supérieure construite de γ est $F\,(\alpha \sqcap \beta)$. L'élimination du constructeur $\sqcap$ n'est pas complète. On ne peut donc pas confier ce calcul de clôture à l'algorithme habituel ; il faut spécifier, dans la définition de la canonisation, que la borne supérieure construite de γ sera $F\,\gamma$.

Parce qu'elle contient ce calcul de clôture, la spécification initiale de l'algorithme, donnée par la définition 11.4, semble complexe. Cependant, nous verrons ensuite qu'une partie des contraintes qu'elle introduit peut être immédiatement éliminée par le dépoussiérage, ce qui nous conduit à une spécification plus simple (définition 11.5).

11.2 Clôture simple

D'un point de vue pratique, la phase de canonisation viendra s'insérer entre les phases de clôture et de dépoussiérage. L'algorithme de canonisation peut donc supposer que le schéma de types sur lequel il travaille est clos, tandis que celui qu'il construit doit être faiblement clos.

Comme il est plus agréable de travailler en termes d'invariant, nous choisissons de démontrer que l'algorithme préserve la *clôture simple*, une notion intermédiaire entre clôture et clôture faible. (Ni la clôture, ni la clôture faible ne sont préservées, la première étant un but trop fort et la seconde une hypothèse trop faible.) L'objet de cette section préliminaire est de définir cette notion et d'établir quelques-unes de ses propriétés.

Expliquons, en deux mots, de quoi il s'agit. Toutes les notions de clôture sont basées sur la même idée : elles exigent que les conséquences (par transitivité et par décomposition structurelle) des contraintes présentes dans le graphe soient elles-mêmes impliquées, d'une certaine façon, par le graphe. Cependant, on peut choisir une notion d'implication plus ou moins puissante. La clôture forte est basée sur l'inclusion entre types (cf. définition 2.11), tandis que la clôture faible utilise notre axiomatisation de l'implication de contraintes, ce qui autorise beaucoup plus de souplesse. Nous adoptons ici une position intermédiaire, en utilisant une notion d'implication assez naïve, basée sur une simple extension de la relation $\leq_C$ aux petits termes. La notion de clôture obtenue reste très simple, tout en étant suffisamment flexible pour convenir à nos intentions.

Définition 11.1 *Soit C un graphe de contraintes. La relation $\leq_C$ est étendue aux termes feuilles en posant*

$$\begin{array}{rcl} \alpha \leq_C \sqcup V & \iff & \exists \beta \in V \quad \alpha \leq_C \beta \\ \sqcap V \leq_C \alpha & \iff & \exists \beta \in V \quad \beta \leq_C \alpha \\ \sqcup V \leq_C \tau & \iff & \forall \beta \in V \quad \beta \leq_C \tau \\ \tau \leq_C \sqcap V & \iff & \forall \beta \in V \quad \tau \leq_C \beta \end{array}$$

Ensuite, $\leq_C$ est étendue aux petits termes, par simple décomposition structurelle.

On notera qu'ainsi, l'assertion $\tau \leq_C \tau'$ est définie quand les deux conditions suivantes sont remplies :

– τ et τ' sont soit deux termes feuilles, soit deux petits termes ;
– $(\tau, \tau') \in (\mathcal{T}^+ \times \mathcal{T}^-) \cup (\mathcal{T}^+ \times \mathcal{T}^+) \cup (\mathcal{T}^- \times \mathcal{T}^-)$.

En particulier, elle reste indéfinie quand $(\tau, \tau') \in \mathcal{T}^- \times \mathcal{T}^+$.

Proposition 11.1 *Si $\leq_C$ est transitive sur les variables, alors son extension aux termes feuilles et aux petits termes est transitive.*

Démonstration. Dans le cas des termes feuilles, quatre cas se présentent, selon la forme des trois types mis en jeu ; tous sont immédiats. La propriété se transmet ensuite aux petits termes par décomposition structurelle. □

Proposition 11.2 *Les propriétés suivantes, qui, d'après la définition 11.1, sont vérifiées lorsque S et T sont des ensembles de variables de types, s'étendent aux termes feuilles et aux petits termes.*

$$\begin{array}{lcl} \sqcup S \leq_C \sqcap T & \iff & \forall \tau \in S \quad \forall \tau' \in T \quad \tau \leq_C \tau' \\ \sqcup S \leq_C \sqcup T & \iff & \forall \tau \in S \quad \exists \tau' \in T \quad \tau \leq_C \tau' \\ \sqcap S \leq_C \sqcap T & \iff & \forall \tau' \in T \quad \exists \tau \in S \quad \tau \leq_C \tau' \end{array}$$

Démonstration. On notera que pour que ces assertions aient un sens, S et T doivent être des ensembles de pos-types (resp. neg-types) quand ils sont arguments de $\sqcup$ (resp. $\sqcap$). La vérification est laissée au lecteur. □

Définition 11.2 *Un graphe de contraintes C, de domaine V, est* simplement clos *ssi*

– *$\leq_C$ est transitive ;*
– *pour tous $\alpha, \beta \in V$ tels que $\alpha \leq_C \beta$, $C^\downarrow(\alpha) \leq_C C^\downarrow(\beta)$ et $C^\uparrow(\alpha) \leq_C C^\uparrow(\beta)$;*
– *pour tout $\alpha \in V$, $C^\downarrow(\alpha) \leq_C C^\uparrow(\alpha)$.*

Un schéma de types $\sigma = A \Rightarrow \tau \mid C$ est simplement clos ssi C l'est.

On notera que les deuxième et troisième conditions ci-dessus peuvent être vérifiées avant de clore $\leq_C$ par transitivité. C'est-à-dire, en termes formels :

Proposition 11.3 *Soit C un graphe de contraintes, de domaine V, tel que*

– *pour tous $\alpha, \beta \in V$ tels que $\alpha \leq_C \beta$, $C^\downarrow(\alpha) \leq_C C^\downarrow(\beta)$ et $C^\uparrow(\alpha) \leq_C C^\uparrow(\beta)$;*
– *pour tout $\alpha \in V$, $C^\downarrow(\alpha) \leq_C C^\uparrow(\alpha)$.*

Alors, le graphe de contraintes D défini par $\leq_D = \leq_C^*$, $D^\uparrow = C^\uparrow$ et $D^\downarrow = C^\downarrow$ est simplement clos.

Démonstration. $\leq_D$ est transitive par construction, et la troisième condition est vérifiée par hypothèse, donc il suffit de vérifier la deuxième condition. Supposons donc $\alpha \leq_D \beta$. Puisque $\leq_D$ est la clôture transitive de $\leq_C$, il existe une chaîne $\alpha = \gamma_1 \leq_C \ldots \leq_C \gamma_n = \beta$. D'après notre hypothèse, ceci implique $C^\downarrow(\gamma_1) \leq_C \ldots \leq_C C^\downarrow(\gamma_n)$. Puisque $\leq_D$ contient $\leq_C$, on a donc $C^\downarrow(\gamma_1) \leq_D \ldots \leq_D C^\downarrow(\gamma_n)$. Enfin, puisque $\leq_D$ est transitive, la proposition 11.1 implique $C^\downarrow(\gamma_1) \leq_D C^\downarrow(\gamma_n)$, que l'on peut également écrire $D^\downarrow(\alpha) \leq_D D^\downarrow(\beta)$. De même, $D^\uparrow(\alpha) \leq_D D^\uparrow(\beta)$. □

La proposition suivante formalise le fait que la clôture simple est intermédiaire entre clôture et clôture faible. Elle indique également que l'algorithme de dépoussiérage peut être appliqué au schéma de types produit par la phase de canonisation brute.

Proposition 11.4 *Tout graphe de contraintes clos est simplement clos. Tout graphe de contraintes simplement clos est faiblement clos. Tout schéma de types simple et simplement clos est dépoussiérable.*

Démonstration. Soit C un graphe clos. Alors $\leq_C$ est transitive. De plus, on vérifie aisément que si un pos-terme feuille τ' contient un pos-terme feuille τ, alors $\tau \leq_C \tau'$. Un résultat symétrique s'applique aux neg-termes ; ces résultats s'étendent ensuite aux petits termes. La deuxième condition requise pour la clôture simple en découle. En ce qui concerne la troisième, elle découle d'une propriété similaire, reliant cette fois un pos-terme à un neg-terme.

Soit C un graphe simplement clos. On vérifie aisément que $\tau \leq_C \tau'$ implique $C \vdash \tau \leq \tau'$. (Encore une fois, on le vérifie d'abord dans le cas des termes feuilles, puis on passe aux petits termes.) Par conséquent, C est également faiblement clos.

Enfin, soit $\sigma = A \Rightarrow \tau \mid C$ un schéma de types simple et simplement clos. Alors C est faiblement clos, et $\leq_C$ est transitive ; donc, d'après la définition 10.4, σ est dépoussiérable. □

11.3 Canonisation brute

Nous allons maintenant décrire une première version de l'algorithme de canonisation. Celui-ci pourra être utilisé pour éliminer les constructeurs $\sqcup$ et $\sqcap$, mais aussi, optionnellement, pour éliminer les variables bipolaires. De façon à obtenir une présentation unifiée, nous le paramétrons par un *filtre.*

Définition 11.3 *Soit V un ensemble de variables de types. Un* filtre *de domaine V est un sous-ensemble de 2^V (c'est-à-dire un ensemble de parties de V) contenant toutes les parties de cardinal strictement supérieur à 1, et ne contenant pas l'ensemble vide.*

Un filtre $\mathcal{F}$ indique à l'algorithme quelles transformations il doit effectuer. Tout terme feuille peut s'écrire sous la forme $\sqcup S$ ou $\sqcap S$, où S est un ensemble (éventuellement réduit à un élément) de variables. Si $S \in \mathcal{F}$, un tel terme sera éliminé, et remplacé par une variable fraîche ; il restera inchangé dans le cas contraire.

Tout ensemble S de cardinal strictement supérieur à 1 est obligatoirement compris dans $\mathcal{F}$, ce qui signifie que toutes les occurrences de $\sqcup$ et $\sqcap$ seront éliminées. Une variable α peut s'écrire, selon sa position dans le terme, $\sqcup\{\alpha\}$ ou $\sqcap\{\alpha\}$; si $\{\alpha\} \in \mathcal{F}$, chacun de ces deux termes sera remplacé par une variable fraîche différente. Nous montrerons que la transformation n'introduit aucune variable bipolaire ; ainsi, pour éliminer toutes les variables bipolaires, il suffira d'inclure tous les singletons correspondants à des variables bipolaires dans le filtre. Si, au contraire, on désire simplement effectuer une canonisation, il ne sera pas nécessaire que le filtre contienne les singletons.

Définissons à présent le processus de canonisation brute. Rappelons que celui-ci sera ensuite combiné avec une phase de dépoussiérage partiel (section 11.4), ce qui conduira à une définition nettement plus simple.

$$
\begin{array}{rlcrl}
r^+(\alpha) = \alpha & \text{quand } \{\alpha\} \notin \mathcal{F} & \qquad & r^-(\alpha) = \alpha & \text{quand } \{\alpha\} \notin \mathcal{F} \\
r^+(\sqcup S) = \lambda_S & \text{quand } S \in \mathcal{F} & & r^-(\sqcap S) = \gamma_S & \text{quand } S \in \mathcal{F} \\
r^+(\bot) = \bot & & & r^-(\bot) = \bot & \\
r^+(\top) = \top & & & r^-(\top) = \top & \\
r^+(\tau_0 \to \tau_1) = r^-(\tau_0) \to r^+(\tau_1) & & & r^-(\tau_0 \to \tau_1) = r^+(\tau_0) \to r^-(\tau_1) &
\end{array}
$$

FIG. 11.1: Définition des fonctions de réécriture

Définition 11.4 *Soit $\sigma = A \Rightarrow \tau \mid C$ un schéma de types simplement clos, de domaine V. Soit $\mathcal{F}$ un filtre de domaine V.*

Pour chaque $S \in \mathcal{F}$, on choisit deux variables fraîches λ_S et γ_S. (Par variables fraîches, on entend que ces variables sont distinctes deux à deux et distinctes de celles de σ.) On

$$D^{\downarrow}(\alpha) = r^{+}(C^{\downarrow}(\alpha)) \qquad D^{\uparrow}(\alpha) = r^{-}(C^{\uparrow}(\alpha))$$
$$D^{\downarrow}(\gamma_S) = r^{+}(\bigsqcup_{\alpha \leq_C \sqcap S} C^{\downarrow}(\alpha)) \qquad D^{\uparrow}(\gamma_S) = r^{-}(\bigsqcap_{\alpha \in S} C^{\uparrow}(\alpha))$$
$$D^{\downarrow}(\lambda_S) = r^{+}(\bigsqcup_{\alpha \in S} C^{\downarrow}(\alpha)) \qquad D^{\uparrow}(\lambda_S) = r^{-}(\bigsqcap_{\sqcup S \leq_C \alpha} C^{\uparrow}(\alpha))$$

FIG. 11.2: Définition de $D^{\downarrow}$ et $D^{\uparrow}$

$$\begin{array}{rcll}
\alpha & \leq_D & \beta & \text{quand } \alpha \leq_C \beta \\
\gamma_S & \leq_D & \alpha & \text{quand } \alpha \in S \\
\alpha & \leq_D & \lambda_S & \text{quand } \alpha \in S \\
r^{-}(\sqcap S) & \leq_D & \gamma_T & \text{quand } |\,S\,| \geq 1 \text{ et } \sqcap S \leq_C \sqcap T \\
\lambda_S & \leq_D & r^{+}(\sqcup T) & \text{quand } |\,T\,| \geq 1 \text{ et } \sqcup S \leq_C \sqcup T \\
r^{+}(\sqcup S) & \leq_D & r^{-}(\sqcap T) & \text{quand } |\,S\,| \geq 1, |\,T\,| \geq 1 \text{ et } \sqcup S \leq_C \sqcap T
\end{array}$$

FIG. 11.3: Définition de $\leq_D$, modulo clôture transitive

définit les fonctions de réécriture r^{+} *et* r^{-} *selon la figure 11.1 page précédente. La première (resp. seconde) est définie sur les pos-termes (resp. neg-termes) feuilles et les petits pos-termes (resp. neg-termes).*

On définit le graphe résultat D *par ses composants* $D^{\downarrow}$ *et* $D^{\uparrow}$*, donnés par la figure 11.2, et par la relation* $\leq_D$*, qui est la* clôture transitive *de la relation donnée par la figure 11.3.*

Le schéma de types produit par le processus de canonisation brute*, noté* $\mathrm{Can}^0_{\mathcal{F}}(\sigma)$*, est* $r^{-}(A) \Rightarrow r^{+}(\tau) \mid D$*.*

Le reste de cette section est consacré à la preuve de deux résultats. Premièrement, cette transformation est bien une canonisation, c'est-à-dire qu'elle fournit un schéma de types équivalent au schéma original, mais dénué de toute occurrence de $\sqcup$ ou $\sqcap$. Deuxièmement, elle préserve la clôture simple, comme nous l'avons annoncé.

Lemme 11.5 *Si* τ *est un pos-terme feuille ou un petit pos-terme, alors* $\tau \leq_D r^{+}(\tau)$*. Si* τ *est un neg-terme feuille ou un petit neg-terme, alors* $r^{-}(\tau) \leq_D \tau$*.*

Démonstration. Considérons un pos-terme feuille $\sqcup S$. Si $S \notin \mathcal{F}$, alors $S = \{\alpha\}$, donc $\tau = \alpha = r^{+}(\tau)$. Sinon, $r^{+}(\tau) = \lambda_S$. Nous devons alors vérifier $\sqcup S \leq_D \lambda_S$. Soit $\alpha \in S$. Alors, par définition de $\leq_D$, $\alpha \leq_D \lambda_S$. Le résultat est donc vérifié pour les pos-termes feuilles. Le cas des neg-termes feuilles est symétrique, et l'extension aux petits termes ne pose pas de difficulté. □

Voici maintenant le premier des résultats attendus.

Théorème 11.1 *Soit* σ *un schéma de types simplement clos. Soit* $\mathcal{F}$ *un filtre de domaine* $\mathrm{dom}(\sigma)$*. On pose* $\sigma' = \mathrm{Can}^0_{\mathcal{F}}(\sigma)$*. Alors* σ' *est simple, et* $\sigma =^{\forall} \sigma'$*.*

Démonstration. Les fonctions de réécriture r^{+} et r^{-} ne produisent aucune construction $\sqcup$ ou $\sqcap$. Aussi, il est clair que σ' est simple.

Etant donné que $\leq_D$ contient $\leq_C$, et étant donné le lemme 11.5, on constate aisément que $D \Vdash C$, et que $D \Vdash (A \Rightarrow \tau) \leq (r^{-}(A) \Rightarrow r^{+}(\tau))$. Il en découle que $\sigma \leq^{\forall} \sigma'$.

Réciproquement, soit ρ une solution de C. Définissons ρ' par

$$\rho'(\alpha) = \rho(\alpha) \qquad \rho'(\gamma_S) = \bigsqcap_{\alpha \in S} \rho(\alpha) \qquad \rho'(\lambda_S) = \bigsqcup_{\alpha \in S} \rho(\alpha)$$

Alors, on vérifie aisément que pour tout pos-terme feuille τ, $\rho'(r^+(\tau)) = \rho(\tau)$. De même, pour tout neg-terme feuille τ, $\rho'(r^-(\tau)) = \rho(\tau)$. Ce résultat s'étend aisément aux petits termes. Grâce à lui, il est facile de vérifier que ρ' est solution de D, et que

$$\rho'(r^-(A) \Rightarrow r^+(\tau)) \leq \rho(A \Rightarrow \tau)$$

Donc, $\sigma' \leq^{\forall} \sigma$. □

Lemme 11.6 *Si τ et τ' sont deux pos-termes feuilles, ou deux petits pos-termes, tels que $\tau \leq_C \tau'$, alors $r^+(\tau) \leq_D r^+(\tau')$. Si τ et τ' sont deux neg-termes feuilles, ou deux petits neg-termes, tels que $\tau \leq_C \tau'$, alors $r^-(\tau) \leq_D r^-(\tau')$.*

Démonstration. Soient $\sqcup S$ et $\sqcup T$ deux pos-termes feuilles tels que $\sqcup S \leq_C \sqcup T$. Distinguons deux cas, selon que $S \in \mathcal{F}$ ou non.

Si $S \notin \mathcal{F}$, alors S est nécessairement de la forme $\{\alpha\}$, et $r^+(\sqcup S) = \alpha$. L'hypothèse $\sqcup S \leq_C \sqcup T$ se traduit en $\alpha \leq_C \sqcup T$. Par ailleurs, d'après le lemme 11.5, $\sqcup T \leq_D r^+(\sqcup T)$. Puisque $\leq_D$ contient $\leq_C$, et puisque $\leq_D$ est transitive, il en découle que $\alpha \leq_D r^+(\sqcup T)$, ce qui était notre but.

Si $S \in \mathcal{F}$, alors $r^+(\sqcup S) = \lambda_S$, donc le but devient $\lambda_S \leq_D r^+(\sqcup T)$. Rappelons que $\sqcup S \leq_C \sqcup T$; le résultat en découle par définition de $\leq_D$.

Nous avons traité le cas des pos-termes feuilles ; le cas des neg-termes feuilles est symétrique, et l'extension aux petits termes suit. □

Lemme 11.7 *Soient τ un pos-terme feuille et τ' un neg-terme feuille tels que $\tau \leq_C \tau'$. Alors $r^+(\tau) \leq_D r^-(\tau')$. Ce résultat s'étend aux petits termes.*

Démonstration. Posons $\tau = \sqcup S$ et $\tau' = \sqcap T$. L'assertion $r^+(\sqcup S) \leq_D r^-(\sqcap T)$ en découle par définition de $\leq_D$. L'extension aux petits termes est aisée. □

Voici à présent le second résultat promis.

Théorème 11.2 *Soit σ un schéma de types simplement clos. Soit $\mathcal{F}$ un filtre de domaine* $\mathrm{dom}(\sigma)$. *On pose $\sigma' = \mathrm{Can}^0_{\mathcal{F}}(\sigma)$. Alors σ' est simplement clos.*

Démonstration. $\leq_D$ est clos par construction, donc il reste à vérifier les deux autres conditions. Commençons par la dernière, qui exige que borne inférieure et borne supérieure de toute variable soient dans la relation $\leq_D$.

- Soit $\alpha \in V$. Puisque C est simplement clos, nous avons $C^{\downarrow}(\alpha) \leq_C C^{\uparrow}(\alpha)$. D'après le lemme 11.7, nous avons alors $r^+(C^{\downarrow}(\alpha)) \leq_D r^-(C^{\uparrow}(\alpha))$.
- Soit $S \in \mathcal{F}$. Nous désirons montrer

  $$r^+(\bigsqcup_{\alpha \leq_C \sqcap S} C^{\downarrow}(\alpha)) \leq_D r^-(\bigsqcap_{\alpha \in S} C^{\uparrow}(\alpha))$$

  Grâce au lemme 11.7, il suffit de montrer

  $$\bigsqcup_{\alpha \leq_C \sqcap S} C^{\downarrow}(\alpha) \leq_C \bigsqcap_{\alpha \in S} C^{\uparrow}(\alpha)$$

  qui, d'après la proposition 11.2, est équivalent à

  $$\forall \alpha \leq_C \sqcap S \quad \forall \beta \in S \quad C^{\downarrow}(\alpha) \leq_C C^{\uparrow}(\beta)$$

  Choisissons $\alpha \leq_C \sqcap S$ et $\beta \in S$. Cela implique $\alpha \leq_C \beta$. C étant simplement clos, cela implique $C^{\downarrow}(\alpha) \leq_C C^{\downarrow}(\beta)$. Par ailleurs, nous avons $C^{\downarrow}(\beta) \leq_C C^{\uparrow}(\beta)$. Par transitivité de $\leq_C$, il en découle que $C^{\downarrow}(\alpha) \leq_C C^{\uparrow}(\beta)$.
- Le cas de λ_S est symétrique.

Traitons maintenant la deuxième condition, qui demande que lorsque deux variables sont dans la relation $\leq_D$, leurs bornes inférieures (resp. supérieures) le soient également. Grâce à la proposition 11.3, il suffit de vérifier que cette condition est vérifiée pour les contraintes explicitement données par la figure 11.3.

- Considérons $\alpha \leq_D \beta$ quand $\alpha \leq_C \beta$. Puisque C est simplement clos, nous avons $C^\downarrow(\alpha) \leq_C C^\downarrow(\beta)$ et $C^\uparrow(\alpha) \leq_C C^\uparrow(\beta)$. Le résultat découle alors du lemme 11.6.
- Considérons $\gamma_S \leq_D \alpha$ quand $\alpha \in S$. Nous devons montrer

  $$r^+(\bigsqcup_{\beta \leq_C \sqcap S} C^\downarrow(\beta)) \leq_D r^+(C^\downarrow(\alpha)) \quad \text{et} \quad r^-(\bigsqcap_{\beta \in S} C^\uparrow(\beta)) \leq_D r^-(C^\uparrow(\alpha))$$

  D'après le lemme 11.6, il suffit de vérifier

  $$\bigsqcup_{\beta \leq_C \sqcap S} C^\downarrow(\beta) \leq_C C^\downarrow(\alpha) \quad \text{et} \quad \bigsqcap_{\beta \in S} C^\uparrow(\beta) \leq_C C^\uparrow(\alpha)$$

  La seconde de ces assertions est conséquence directe de la proposition 11.2. Toujours d'après la proposition 11.2, la première est équivalente à

  $$\forall \beta \leq_C \sqcap S \quad C^\downarrow(\beta) \leq_C C^\downarrow(\alpha)$$

  Soit $\beta \leq_C \sqcap S$. Alors $\beta \leq_C \alpha$, puisque $\alpha \in S$. Il s'ensuit que $C^\downarrow(\beta) \leq_C C^\downarrow(\alpha)$.
- Considérons $\alpha \leq_D \lambda_S$ quand $\alpha \in S$. Ce cas est symétrique au précédent.
- Considérons $r^-(\sqcap S) \leq_D \gamma_T$ quand $|\, S\, | \geq 1$ et $\sqcap S \leq_C \sqcap T$. Supposons d'abord $S \in \mathcal{F}$. Alors, nous devons démontrer (modulo application du lemme 11.6, comme toujours)

  $$\bigsqcup_{\alpha \leq_C \sqcap S} C^\downarrow(\alpha) \leq_C \bigsqcup_{\beta \leq_C \sqcap T} C^\downarrow(\beta) \quad \text{et} \quad \bigsqcap_{\alpha \in S} C^\uparrow(\alpha) \leq_C \bigsqcap_{\beta \in T} C^\uparrow(\beta)$$

  Considérons la première de ces assertions. Par transitivité de $\leq_C$, $\alpha \leq_C \sqcap S$ implique $\alpha \leq_C \sqcap T$, donc l'expression $\sqcup$ de gauche a moins d'opérandes que celle de droite. L'assertion se déduit alors de la proposition 11.2. Considérons la seconde assertion. Nous avons $\sqcap S \leq_C \sqcap T$, c'est-à-dire

  $$\forall \beta \in T \quad \exists \alpha \in S \quad \alpha \leq_C \beta$$

  C étant simplement clos, ceci implique

  $$\forall \beta \in T \quad \exists \alpha \in S \quad C^\downarrow(\alpha) \leq_C C^\downarrow(\beta)$$

  qui à son tour, d'après la proposition 11.2, fournit exactement la seconde assertion.

  Il reste à traiter le cas où $S \notin \mathcal{F}$. Alors, S est de la forme $\{\alpha\}$ et nous devons montrer

  $$C^\downarrow(\alpha) \leq_C \bigsqcup_{\beta \leq_C \sqcap T} C^\downarrow(\beta) \quad \text{et} \quad C^\uparrow(\alpha) \leq_C \bigsqcap_{\beta \in T} C^\uparrow(\beta)$$

  On remarque que la seconde de ces assertions est identique à la seconde rencontrée plus haut. Quant à la première, c'est une conséquence de la première assertion rencontrée plus haut, car

  $$C^\downarrow(\alpha) \leq_C \bigsqcup_{\alpha \leq_C \sqcap S} C^\downarrow(\alpha)$$
- Considérons $\lambda_S \leq_D r^+(\sqcup T)$ quand $|\, T\, | \geq 1$ et $\sqcup S \leq_C \sqcup T$. Ce cas est symétrique au précédent.

- Considérons $r^+(\sqcup S) \leq_D r^-(\sqcap T)$ quand $|S| \geq 1$, $|T| \geq 1$ et $\sqcup S \leq_C \sqcap T$. Supposons d'abord $S \in \mathcal{F}$ et $T \in \mathcal{F}$. Nous devons montrer (modulo une application du lemme 11.6)

$$\bigsqcup_{\alpha \in S} C^{\downarrow}(\alpha) \leq_C \bigsqcup_{\beta \leq_C \sqcap T} C^{\downarrow}(\beta)$$

(Nous omettons l'autre but, qui est symétrique.) Ceci est équivalent à

$$\forall \alpha \in S \quad C^{\downarrow}(\alpha) \leq_C \bigsqcup_{\beta \leq_C \sqcap T} C^{\downarrow}(\beta)$$

Soit $\alpha \in S$. Puisque $\sqcup S \leq_C \sqcap T$, nous avons $\alpha \leq_C \sqcap T$. Il en découle que α est l'un des β ci-dessus, et l'inégalité est vérifiée.
Quand $S \notin \mathcal{F}$, S est de la forme $\{\alpha\}$ et le terme de gauche est $C^{\downarrow}(\alpha)$; l'assertion est donc inchangée.
Quand $T \notin \mathcal{F}$, T est de la forme $\{\alpha\}$ et le terme de droite est $C^{\downarrow}(\alpha)$. Il suffit de vérifier

$$\bigsqcup_{\beta \leq_C \sqcap T} C^{\downarrow}(\beta) \leq_C C^{\downarrow}(\alpha)$$

Soit $\beta \leq_C \sqcap T$. Alors $\beta \leq_C \alpha$, donc $C^{\downarrow}(\beta) \leq_C C^{\downarrow}(\alpha)$. Tous les opérandes de l'expression de gauche sont alors inférieurs à l'expression de droite, d'où le résultat. □

11.4 Canonisation

A partir d'un schéma de types σ simplement clos, le processus de canonisation brute exposé dans la section précédente produit un schéma de types σ' dépoussiérable. Par conséquent, on peut effectuer sur σ' le calcul des polarités puis le dépoussiérage, c'est-à-dire l'élimination des contraintes superflues. Nous allons à présent simuler ce calcul.

De façon générale, nous ne pouvons pas prédire exactement la polarité de chaque variable dans σ', parce qu'elle dépend en partie de σ, qui n'est pas connu. Cependant, nous pouvons en calculer une approximation. Celle-ci se révélera suffisamment précise pour permettre d'éliminer avec certitude une large classe de contraintes. Nous pourrons alors combiner le processus de canonisation brute exposé plus haut avec ce dépoussiérage approché, ce qui revient, en pratique, à éviter de créer ces contraintes superflues, plutôt que de les construire pour les jeter aussitôt après. Nous obtiendrons ainsi une spécification de la canonisation nettement moins lourde que la précédente.

Ce calcul approché des polarités dans σ' révèle que les variables λ_S fraîchement introduites sont au plus positives, tandis que les γ_S sont au plus négatives. Les variables existantes α voient leur polarité décroître. En particulier, si $\{\alpha\} \in \mathcal{F}$, alors α devient neutre. Ces résultats sont formalisés ci-dessous.

Lemme 11.8 *Soient σ un schéma de types simplement clos, $\mathcal{F}$ un filtre de domaine* $\mathrm{dom}(\sigma)$ *et $\sigma' = \mathrm{Can}^0_{\mathcal{F}}(\sigma)$. Alors*

$$\mathrm{dom}^+(\sigma') \subset \{\lambda_S\,;\, S \in \mathcal{F} \wedge S \subset \mathrm{dom}^+(\sigma)\} \cup \{\alpha\,;\, \{\alpha\} \notin \mathcal{F} \wedge \alpha \in \mathrm{dom}^+(\sigma)\}$$
$$\mathrm{dom}^-(\sigma') \subset \{\gamma_S\,;\, S \in \mathcal{F} \wedge S \subset \mathrm{dom}^-(\sigma)\} \cup \{\alpha\,;\, \{\alpha\} \notin \mathcal{F} \wedge \alpha \in \mathrm{dom}^-(\sigma)\}$$

Démonstration. Soit P_0 (resp. N_0) le membre droit de la première (resp. seconde) assertion ci-dessus.

Commençons par formuler une remarque à propos des fonctions de réécriture. Si $S \subset \mathrm{dom}^+(\sigma)$, alors $r^+(\sqcup S) \in P_0$. En d'autres termes, à tout terme feuille positif, r^+ associe un élément de P_0. La fonction r^- jouit d'une propriété symétrique.

D'après la définition 10.3, $\mathrm{dom}^+(\sigma')$ et $\mathrm{dom}^-(\sigma')$ sont les plus petits ensembles P et N tels que

- $r^+(\tau) \in P$
- $\forall (x : \tau_x) \in A \quad r^-(\tau_x) \in N$
- $\forall \delta \in P \quad (\mathrm{fv}^+(D^\downarrow(\delta)) \subset P) \wedge (\mathrm{fv}^-(D^\downarrow(\delta)) \subset N)$
- $\forall \delta \in N \quad (\mathrm{fv}^+(D^\uparrow(\delta)) \subset N) \wedge (\mathrm{fv}^-(D^\uparrow(\delta)) \subset P)$

Nous prétendons que P_0 et N_0 vérifient également ces quatre conditions. Le résultat en découlera alors immédiatement.

τ est un terme feuille, donc s'écrit $\sqcup S$, où $S \subset \mathrm{dom}^+(\sigma)$ par définition des polarités dans σ. D'après la remarque ci-dessus, cela implique $r^+(\tau) \in P_0$. De même, pour tout $(x : \tau_x) \in A$, nous avons $r^-(\tau_x) \in N_0$.

Considérons à présent la troisième condition. Soit $\delta \in P_0$. Que δ soit de la forme α, γ_S ou λ_S, on a

$$D^\downarrow(\delta) = r^+(\bigsqcup_{\alpha \in V} C^\downarrow(\alpha))$$

pour un certain $V \subset \mathrm{dom}(\sigma)$ (cf. figure 11.2). La valeur de V dépend de la forme de δ :

- Si δ est de la forme $\alpha \in \mathrm{dom}(\sigma)$, alors $V = \{\alpha\}$. Puisque $\delta \in P_0$, on a $\alpha \in \mathrm{dom}^+(\sigma)$.
- δ ne peut pas être de la forme γ_S, puisque $\delta \in P_0$.
- Si δ est de la forme λ_S, alors $V = S$. Puisque $\delta \in P_0$, on a $S \subset \mathrm{dom}^+(\sigma)$.

Ainsi, quel que soit δ, on a $V \subset \mathrm{dom}^+(\sigma)$. En d'autres termes, tout $\alpha \in V$ est positif dans σ. Par définition des polarités dans σ, cela implique que chaque $C^\downarrow(\alpha)$ est également un terme positif. Il en va donc de même de l'union de tous ces termes, $\bigsqcup_{\alpha \in V} C^\downarrow(\alpha)$, que nous noterons τ_δ. Plus précisément, $\mathrm{fv}^+(\tau_\delta) \subset \mathrm{dom}^+(\sigma)$ et $\mathrm{fv}^-(\tau_\delta) \subset \mathrm{dom}^-(\sigma)$. D'après notre remarque initiale, cela implique $\mathrm{fv}^+(r^+(\tau_\delta)) \subset P_0$ et $\mathrm{fv}^-(r^+(\tau_\delta)) \subset N_0$. La troisième condition est donc satisfaite.

La quatrième condition est symétrique à la précédente. □

De ce lemme découlent plusieurs propriétés intéressantes. La première indique que les polarités décroissent pendant la canonisation.

Proposition 11.9 *Soient σ un schéma de types simplement clos, $\mathcal{F}$ un filtre de domaine* $\mathrm{dom}(\sigma)$ *et* $\sigma' = \mathrm{Can}^0_{\mathcal{F}}(\sigma)$. *Alors*

$$\mathrm{dom}^+(\sigma') \cap \mathrm{dom}(\sigma) \subset \mathrm{dom}^+(\sigma)$$
$$\mathrm{dom}^-(\sigma') \cap \mathrm{dom}(\sigma) \subset \mathrm{dom}^-(\sigma)$$

Démonstration. Soit $\alpha \in \mathrm{dom}^+(\sigma') \cap \mathrm{dom}(\sigma)$. D'après le lemme 11.8, $\alpha \in \mathrm{dom}^+(\sigma')$ implique $\alpha \in \{\lambda_S\,;\, S \in \mathcal{F} \wedge S \subset \mathrm{dom}^+(\sigma)\} \cup \{\alpha\,;\, \{\alpha\} \notin \mathcal{F} \wedge \alpha \in \mathrm{dom}^+(\sigma)\}$. Si on prend en compte le fait que $\alpha \in \mathrm{dom}(\sigma)$, cette assertion devient $\alpha \in \{\alpha\,;\, \{\alpha\} \notin \mathcal{F} \wedge \alpha \in \mathrm{dom}^+(\sigma)\}$, ce qui implique $\alpha \in \mathrm{dom}^+(\sigma)$, d'où le premier résultat attendu. Le second est symétrique. □

La propriété suivante montre que l'algorithme peut être utilisé pour éliminer les variables bipolaires, donc pour faire respecter *l'invariant de mono-polarité* introduit au chapitre 12. En même temps, elle indique que cet invariant est conservé par la transformation.

Proposition 11.10 *Soient σ un schéma de types simplement clos et $\mathcal{F}$ un filtre de domaine* $\mathrm{dom}(\sigma)$. *On suppose que toutes les variables bipolaires de σ sont « filtrées », c'est-à-dire que*

$$\forall \alpha \in \mathrm{dom}^+(\sigma) \cap \mathrm{dom}^-(\sigma) \quad \{\alpha\} \in \mathcal{F}$$

Alors, $\sigma' = \mathrm{Can}^0_{\mathcal{F}}(\sigma)$ *n'a aucune variable bipolaire, c'est-à-dire*

$$\mathrm{dom}^+(\sigma') \cap \mathrm{dom}^-(\sigma') = \varnothing$$

Démonstration. D'après le lemme 11.8,

$$\mathrm{dom}^+(\sigma') \cap \mathrm{dom}^-(\sigma') \subset \{\alpha\,;\, \{\alpha\} \notin \mathcal{F} \wedge \alpha \in \mathrm{dom}^+(\sigma) \cap \mathrm{dom}^-(\sigma)\}$$

Or, le membre droit de cette assertion est précisément l'ensemble vide, grâce à notre hypothèse. □

Enfin, notre calcul approché des polarités nous permet de simuler une étape de dépoussiérage. On notera qu'il ne s'agit que d'un dépoussiérage partiel, puisque les polarités ne sont pas connues avec précision. En d'autres termes, le schéma de types σ'' que nous allons calculer n'est pas $\mathrm{GC}(\sigma')$; il contient plus de contraintes que $\mathrm{GC}(\sigma')$, mais moins que σ'. En pratique, cela ne pose pas de problèmes, puisque la phase de canonisation est suivie d'une véritable phase de dépoussiérage. La machine calculera donc $\mathrm{GC}(\sigma'')$, qui n'est autre que $\mathrm{GC}(\sigma')$, comme si nous avions directement fourni σ' à l'algorithme de dépoussiérage. Nous avons donc, du point de vue théorique, ajouté une étape, mais nous obtenons, en pratique, un gain de performance. En effet, la machine ne calculera pas σ' ; elle produira directement σ'', évitant ainsi d'engendrer des contraintes inutiles.

Nous définissons à présent la canonisation comme la combinaison de la canonisation brute et de ce dépoussiérage partiel.

$$\begin{array}{ll}
E^{\downarrow}(\alpha) = r^+(C^{\downarrow}(\alpha)) & E^{\uparrow}(\alpha) = r^-(C^{\uparrow}(\alpha)) \\
E^{\downarrow}(\gamma_S) = \bot & E^{\uparrow}(\gamma_S) = r^-(\displaystyle\bigsqcap_{\alpha \in S} C^{\uparrow}(\alpha)) \\
E^{\downarrow}(\lambda_S) = r^+(\displaystyle\bigsqcup_{\alpha \in S} C^{\downarrow}(\alpha)) & E^{\uparrow}(\lambda_S) = \top
\end{array}$$

FIG. 11.4: Définition de $E^{\downarrow}$ et $E^{\uparrow}$

$$\begin{array}{rcll}
\alpha & \leq_E & \beta & \text{quand } \alpha \leq_C \beta \\
\gamma_S & \leq_E & \alpha & \text{quand } \sqcap S \leq_C \alpha \\
\alpha & \leq_E & \lambda_S & \text{quand } \alpha \leq_C \sqcup S \\
\gamma_S & \leq_E & \lambda_T & \text{quand } \exists \alpha \in S \quad \exists \beta \in T \quad \alpha \leq_C \beta
\end{array}$$

FIG. 11.5: Définition de $\leq_E$

Définition 11.5 *Soient $\sigma = A \Rightarrow \tau \mid C$ un schéma de types simplement clos et $\mathcal{F}$ un filtre de domaine* $\mathrm{dom}(\sigma)$. *Soit E le graphe de contraintes défini par les figures 11.4 et 11.5. Le schéma de types produit par le processus de* canonisation, *noté* $\mathrm{Can}_{\mathcal{F}}(\sigma)$, *est $r^-(A) \Rightarrow r^+(\tau) \mid E$.*

Il reste à prouver que cette définition est correcte :

Proposition 11.11 *Soient σ un schéma de types simplement clos et $\mathcal{F}$ un filtre de domaine* $\mathrm{dom}(\sigma)$. *On pose $\sigma' = \mathrm{Can}^0_{\mathcal{F}}(\sigma)$ et $\sigma'' = \mathrm{Can}_{\mathcal{F}}(\sigma)$. Alors*

$$\mathrm{GC}(\sigma'') = \mathrm{GC}(\sigma')$$

Démonstration. Avoir de pouvoir calculer les effets du dépoussiérage sur σ', nous devons calculer $\leq_D$ explicitement. Rappelons que $\leq_D$ a jusqu'ici été défini modulo clôture transitive ; mais nous avons maintenant besoin de connaître exactement toutes les contraintes de

$\leq_D$. Nous prétendons qu'elles sont de l'une des formes suivantes :

1.	$\alpha \leq \beta$	où $\alpha \leq_C \beta$
2.	$\gamma_S \leq \alpha$	où $\sqcap S \leq_C \alpha$
3.	$\alpha \leq \lambda_S$	où $\alpha \leq_C \sqcup S$
4.	$\alpha \leq \gamma_S$	où $\alpha \leq_C \sqcap S$
5.	$\gamma_S \leq \gamma_T$	où $\sqcap S \leq_C \sqcap T$
6.	$\lambda_S \leq \alpha$	où $\sqcup S \leq_C \alpha$
7.	$\lambda_S \leq \lambda_T$	où $\sqcup S \leq_C \sqcup T$
8.	$\lambda_S \leq \gamma_T$	où $\sqcup S \leq \sqcap T$
9.	$\gamma_S \leq \lambda_T$	où $\exists \alpha \in S \quad \exists \beta \in T \quad \alpha \leq_C \beta$

La preuve de cette assertion est simple. D'abord, on vérifie aisément que toutes les contraintes données par la figure 11.3 sont de l'une de ces formes. Ensuite, on vérifie que si deux contraintes quelconques, dans la liste ci-dessus, sont combinées par transitivité, alors la contrainte résultante est également de l'une de ces formes. Il y a donc 81 cas, tous triviaux, et laissés au lecteur.

Nous pouvons à présent utiliser le lemme 11.8, qui fournit une approximation des polarités dans σ'. Le dépoussiérage ne conserve que les contraintes reliant une variable négative à une variable positive. Il en découle que les contraintes des formes 4–8 disparaissent nécessairement. Les autres sont précisément celles données par la figure 11.5.

Enfin, le lemme 11.8 permet de jeter la borne inférieure de γ_S et la borne supérieure de λ_S, d'où la définition de la figure 11.4. □

Enfin, reste une optimisation simple dont notre exposé théorique ne rend pas compte. Notre définition de la canonisation crée des variables fraîches γ_S et λ_S pour tout $S \in \mathcal{F}$, ce qui représente un nombre exponentiel de variables fraîches. En pratique, on vérifie aisément que si σ ne contient aucune occurrence de $\sqcup S$ (resp. $\sqcap S$), alors λ_S (resp. γ_S) est neutre dans σ'. En pratique, on ne crée donc la variable λ_S (resp. γ_S) que lorsque l'on rencontre une occurrence de $\sqcup S$ (resp. $\sqcap S$) dans σ.

Pour conclure cette section, voici quelques exemples simples. (Pour plus de concision, les graphes de contraintes sont parfois représentés par des ensembles de contraintes.)

Exemple. Voici d'abord un court exemple destiné à illustrer l'élimination des variables bipolaires. On peut attribuer à la fonction identité $\lambda x.x$ le schéma de types

$$\delta^+ \mid \{\epsilon^\pm \to \epsilon^\pm \leq \delta^+\}$$

(Le schéma $\epsilon^\pm \to \epsilon^\pm$ est également correct, mais ne vérifie pas l'invariant des petits termes.) Ce schéma ne vérifie pas l'invariant de mono-polarité, que nous introduirons au chapitre 12, parce qu'il contient une variable bipolaire ϵ. Pour éliminer toute variable bipolaire, on applique l'algorithme de canonisation, avec un filtre $\mathcal{F}$ contenant le singleton $\{\epsilon\}$. L'algorithme introduit deux variables fraîches γ et λ, reliées par la contrainte $\gamma \leq \lambda$. Il remplace toutes les occurrences négatives (resp. positives) d'ϵ par la première (resp. seconde). On obtient ainsi

$$\delta^+ \mid \{\gamma^- \to \lambda^+ \leq \delta^+,\, \gamma^- \leq \lambda^+\}$$

Exemple. Comme exposé ci-dessus, en présence de l'invariant de mono-polarité, le schéma de types inféré pour la fonction identité $\lambda x.x$ sera le suivant :

$$\delta \mid \{\alpha \to \beta \leq \delta,\, \alpha \leq \beta\}$$

Intéressons-nous alors au schéma inféré pour l'expression

$$\texttt{if true then } \lambda x.x \texttt{ else } \lambda x.x$$

Nous obtenons deux exemplaires du schéma ci-dessus, reliés par un constructeur $\sqcup$:

$$\delta_1 \sqcup \delta_2 \mid \{\alpha_1 \to \beta_1 \leq \delta_1,\, \alpha_2 \to \beta_2 \leq \delta_2,\, \alpha_1 \leq \beta_1,\, \alpha_2 \leq \beta_2\}$$

Appliquons l'algorithme de canonisation à ce schéma. Une variable fraîche δ apparaît pour remplacer $\delta_1 \sqcup \delta_2$. De plus, en calculant la borne inférieure construite de δ, on voit apparaître les expressions $\alpha_1 \sqcap \alpha_2$ et $\beta_1 \sqcup \beta_2$; deux variables fraîches α et β sont créées pour les remplacer. On obtient $\delta \mid C$, où C est le graphe de contraintes défini par

- $\delta_1 \leq_C \delta$, $\delta_2 \leq_C \delta$;
- $\alpha \leq_C \alpha_1 \leq_C \beta_1 \leq_C \beta$, $\alpha \leq_C \alpha_2 \leq_C \beta_2 \leq_C \beta$ et $\alpha \leq_C \beta$;
- $C^{\downarrow}(\delta_1) = \alpha_1 \rightarrow \beta_1$, $C^{\downarrow}(\delta_2) = \alpha_2 \rightarrow \beta_2$ et $C^{\downarrow}(\delta) = \alpha \rightarrow \beta$.

Nous avons donc bien réalisé l'élimination des constructeurs $\sqcup$ et $\sqcap$. Le point le plus intéressant est que, ce faisant, nous avons ouvert de nouvelles possibilités à l'algorithme de dépoussiérage. En effet, avant la canonisation, aucune variable n'était neutre, et le dépoussiérage n'aurait eu aucun effet. A présent, au contraire, le calcul des polarités révèle que δ et β sont positives, α est négative, et toutes les autres variables sont devenues neutres. Le dépoussiérage produit donc le schéma de types

$$\delta \mid \{\alpha \rightarrow \beta \leq \delta,\, \alpha \leq \beta\}$$

qui est identique au schéma de types de l'identité. La canonisation, avec l'aide du dépoussiérage, a donc réussi, d'une certaine façon, à identifier les deux jeux de contraintes identiques produits par les deux branches de la conditionnelle. De façon générale, la canonisation est donc une simplification, non seulement parce qu'elle élimine les constructeurs $\sqcup$ et $\sqcap$, qui sont indésirables dans certaines phases ultérieures, mais aussi parce que ce faisant, elle ouvre des opportunités pour le processus de dépoussiérage.

11.5 Canonisation incrémentale

Nous avons présenté la canonisation comme un algorithme à part, ce qui demande de lui consacrer une phase spécifique au cours du processus de simplification des schémas de types. Ce choix peut être controversé. Plutôt que d'autoriser la phase de clôture à introduire des constructions $\sqcup$ et $\sqcap$ (ou des bornes multiples, ce qui revient au même ; cf. section 2.1), et de devoir ensuite effectuer une passe de canonisation pour les éliminer, ne pourrions-nous pas effectuer la canonisation de façon incrémentale ? C'est-à-dire, modifier l'algorithme de clôture pour que ces constructions soient immédiatement remplacées par des variables fraîches munies des contraintes appropriées ?

Cette idée semble, à première vue, intéressante, parce que si elle est correctement mise en œuvre, elle devrait permettre d'économiser une partie du coût de la canonisation. En effet, l'algorithme de canonisation doit parcourir tout le graphe de contraintes pour trouver les occurrences de $\sqcup$ et $\sqcap$, tandis qu'un algorithme incrémental n'effectuerait de calculs qu'en cas de besoin.

Récapitulons d'abord le principe de la canonisation. Il s'agit, grosso modo, de remplacer l'expression $\alpha \sqcap \beta$ par une variable fraîche γ, en ajoutant les contraintes $\gamma \leq \alpha$ et $\gamma \leq \beta$. L'idée serait donc d'effectuer ce remplacement pendant le calcul de clôture, à chaque fois que la combinaison de deux bornes provoque l'apparition d'une telle expression. Certains points méritent cependant d'être mentionnés.

Le premier point est le risque de non-terminaison. Par exemple, supposons que notre graphe contienne les contraintes $\alpha \leq F\,\alpha$ et $\beta \leq F\,\beta$, où F est un constructeur de types covariant. Supposons que l'expression $\alpha \sqcap \beta$ apparaisse pendant la clôture du graphe. Nous introduisons donc une variable γ, comme indiqué plus haut. Le calcul de clôture continue ; en particulier, il doit calculer la borne supérieure construite de γ. Puisque $\gamma \leq \alpha$ et $\gamma \leq \beta$, cette borne est $F(\alpha \sqcap \beta)$. Ainsi, en ajoutant ces nouvelles contraintes, nous avons provoqué l'apparition d'une nouvelle construction $\sqcap$. Si nous lui associons une nouvelle variable, le même problème se présentera à nouveau et le processus ne terminera pas. Nous devons donc être capables de mémoriser l'association entre $\alpha \sqcap \beta$ et γ. Cela demande de garder trace des variables qui ont été créées et des expressions auxquelles elles sont associées – c'est en fait exactement ce que doit faire l'algorithme non incrémental, puisque qu'il maintient l'association $S \mapsto (\gamma_S, \lambda_S)$. Le passage à une version incrémentale permettrait donc d'économiser le coût (linéaire) du parcours du graphe, mais les calculs à effectuer resteraient les mêmes.

Le second point est le fait que l'algorithme de clôture doit être reformulé. L'algorithme actuel est basé sur la notion de clôture forte. Or, en l'absence de toute construction $\sqcup$ ou $\sqcap$, celle-ci devient trop restrictive ; il faut adopter une notion plus souple, comme la clôture simple. En soi, cela ne devrait pas poser de problème ; l'algorithme reste fondamentalement identique, avec un critère de terminaison légèrement différent, basé sur la comparaison $\leq_C$ plutôt que sur l'inclusion entre types.

Un tel algorithme de clôture, contenant un mécanisme de canonisation incrémental, paraît concevable. Cependant, les détails de sa mise au point et de sa preuve, en particulier en ce qui concerne sa terminaison, nous ont paru relativement délicats. C'est pourquoi nous avons conservé l'algorithme non incrémental, lequel est clairement compris. La perte de performance devrait être minime, puisque la phase de canonisation n'a besoin d'être effectuée que rarement (c'est-à-dire aux nœuds `let`), et le temps perdu durant chaque phase correspond *grosso modo* à un parcours du graphe de contraintes. Ce problème reste cependant une piste de recherche, avec une éventuelle amélioration d'efficacité à la clef.

Chapitre 12

L'invariant de mono-polarité

Lors de la définition du dépoussiérage (cf. chapitre 10), une question a été laissée en suspens : le schéma de types produit par le dépoussiérage est-il clos, selon l'une au moins des notions de clôture développées jusqu'ici ? Cette propriété est souhaitable, puisque tous les algorithmes développés demandent des schémas de types (faiblement, simplement ou fortement) clos. Les différentes phases de simplification se combineront donc plus aisément, et plus efficacement, si elle est satisfaite. Cependant, dans le cas général, la réponse est négative.

L'objet de ce chapitre est de montrer que, à condition d'adopter un invariant – non restrictif – sur la forme des schémas de types, on peut apporter une réponse positive à cette question. Cet invariant sera nommé *invariant de mono-polarité*.

Nous étudions d'abord la forme des schémas de types produits par le dépoussiérage, de façon à comprendre la cause du problème (section 12.1) ; cela nous amène naturellement à la définition de l'invariant et à son effet sur le dépoussiérage (section 12.2). Nous nous intéressons ensuite aux moyens de faire respecter cet invariant (section 12.3), et concluons par diverses remarques sur ses avantages secondaires (section 12.4).

12.1 Motivation

Pour analyser le problème, intéressons-nous au résultat produit par le dépoussiérage dans un cas particulier. Soit σ le schéma de types

$$\alpha \to \alpha \mid \{F\,\beta \leq \alpha \leq F\,\gamma,\, \beta \leq \gamma\}$$

où F est un constructeur de types covariant. (Afin de simplifier la notation, nous utilisons ici un ensemble de contraintes, plutôt qu'un graphe ; de plus, l'invariant des petits termes n'est pas respecté. Cela n'a pas d'influence sur le problème qui nous intéresse.)

σ est clos, puisqu'en reliant, par transitivité sur α, la borne inférieure d'α à sa borne supérieure, on obtient la contrainte $F\,\beta \leq F\,\gamma$, dont la conséquence par décomposition structurelle, à savoir $\beta \leq \gamma$, figure bien dans le graphe de contraintes. Il est donc correct de lui appliquer l'algorithme de dépoussiérage. Commençons par le calcul des polarités :

$$\alpha^{\pm} \to \alpha^{\pm} \mid \{F\,\beta^{+} \leq \alpha^{\pm} \leq F\,\gamma^{-},\, \beta^{+} \leq \gamma^{-}\}$$

Nous pouvons ensuite calculer $\mathrm{GC}(\sigma)$. Celui-ci est identique à σ, mis à part le fait que la contrainte $\beta^{+} \leq \gamma^{-}$ est éliminée. En effet, le dépoussiérage ne conserve une contrainte entre deux variables que si la première est négative et la seconde positive. Donc, $\mathrm{GC}(\sigma)$ s'écrit

$$\alpha^{\pm} \to \alpha^{\pm} \mid \{F\,\beta^{+} \leq \alpha^{\pm} \leq F\,\gamma^{-}\}$$

Or, ce schéma n'est pas clos, quelle que soit la notion de clôture utilisée. En effet, pour qu'il soit faiblement clos, il faudrait qu'on ait $\{F\,\beta^{+} \leq \alpha^{\pm} \leq F\,\gamma^{-}\} \vdash \beta \leq \gamma$, ce qui n'est pas le cas.

Cette situation est embarrassante. D'un point de vue pratique, elle signifie que la phase de dépoussiérage ne s'intègre pas naturellement au processus d'inférence et de simplification de types. Une propriété de clôture est en effet nécessaire, d'abord pour vérifier que le programme étudié est bien typé, ensuite comme prérequis pour les algorithmes de simplification, y compris le dépoussiérage lui-même.

En l'absence de cette propriété, il faudrait explicitement recalculer la clôture du graphe produit par le dépoussiérage. On perdrait donc la propriété d'incrémentalité du calcul de clôture. De plus, cela semble particulièrement peu naturel, puisque ce calcul ajouterait alors à nouveau au graphe des contraintes explicitement éliminées auparavant par le dépoussiérage !

Heureusement, il existe une solution simple à ce problème. Dans l'exemple ci-dessus, la propriété de clôture est violée parce que la variable α conserve une borne inférieure et une borne supérieure non triviales après le dépoussiérage. Or, cela n'est possible que parce que α est bipolaire. Si elle était uniquement positive (resp. négative), sa borne supérieure (resp. inférieure) après le dépoussiérage serait $\top$ (resp. $\perp$) et le problème ne se poserait pas. D'où l'idée d'interdire, purement et simplement, toute variable bipolaire.

12.2 Définition

Définition 12.1 *Soit σ un schéma de types faiblement clos. σ vérifie* l'invariant de mono-polarité *ssi*

$$\mathrm{dom}^+(\sigma) \cap \mathrm{dom}^-(\sigma) = \varnothing$$

c'est-à-dire ssi σ n'a aucune variable bipolaire.

(Dans la définition ci-dessus, la propriété de clôture faible est requise uniquement pour que le calcul des polarités ait un sens.) Il est facile de vérifier que $\mathrm{GC}(\sigma)$ est (fortement) clos lorsque σ vérifie l'invariant de mono-polarité ; en fait, les conditions de clôture sont trivialement remplies, comme le montre la proposition suivante.

Proposition 12.1 *Soit σ un schéma de types dépoussiérable et vérifiant l'invariant de mono-polarité. Alors $\mathrm{GC}(\sigma)$ est clos.*

Démonstration. Posons $\sigma = A \Rightarrow \tau \mid C$ et $\mathrm{GC}(\sigma) = A \Rightarrow \tau \mid D$, comme dans la définition 10.5. Nous devons d'abord vérifier que $\leq_D$ est transitive. Supposons $\alpha \leq_D \beta$ et $\beta \leq_D \gamma$. Alors, par définition de $\leq_D$, $\beta \in \mathrm{dom}^+(\sigma)$ et $\beta \in \mathrm{dom}^-(\sigma)$. Cela est impossible, puisque σ vérifie l'invariant de mono-polarité. Par conséquent, la transitivité de $\leq_D$ est trivialement vérifiée.

Ensuite, supposons $\alpha \leq_D \beta$. Nous désirons prouver que $D^\uparrow(\alpha)$ contient $D^\uparrow(\beta)$. Par définition de $\leq_D$, $\alpha \leq_D \beta$ implique $\beta \in \mathrm{dom}^+(\sigma)$. Puisque σ vérifie l'invariant de mono-polarité, il en découle que $\beta \notin \mathrm{dom}^-(\sigma)$. D'après la définition 10.5, ceci implique $D^\uparrow(\beta) = \top$. Par conséquent, l'assertion d'inclusion entre types est trivialement vérifiée. Symétriquement, on vérifie que $D^\downarrow(\beta)$ contient $D^\downarrow(\alpha)$.

Enfin, soit $\alpha \in \mathrm{dom}(\mathrm{GC}(\sigma))$. Nous souhaitons prouver que D contient la contrainte $D^\downarrow(\alpha) \leq D^\uparrow(\alpha)$. Puisque α ne peut être en même temps positive et négative dans σ, soit $D^\downarrow(\alpha)$ est égal à $\perp$, soit $D^\uparrow(\alpha)$ est égal à $\top$. Dans les deux cas, la contrainte est trivialement contenue dans D.

Nous avons vérifié que les trois conditions de la définition 3.10 sont trivialement satisfaites. D est donc clos, et $\mathrm{GC}(\sigma)$ également. □

Grâce à l'invariant de mono-polarité, le dépoussiérage produit des schémas de types sous une forme particulièrement simple, que nous qualifierons de *parfaite*. Nous allons la décrire brièvement. Commençons par la définir :

Définition 12.2 *Un schéma de types σ est* parfait *ssi il est égal à $\mathrm{GC}(\sigma_0)$, pour un certain σ_0 dépoussiérable et vérifiant l'invariant de mono-polarité.*

Il est clair que tout schéma de types est équivalent à un schéma parfait. Il suffit, pour exhiber celui-ci, d'appliquer successivement les algorithmes de clôture, de canonisation et de dépoussiérage. Les schémas parfaits sont extrêmement simples, comme le montre la proposition suivante :

Proposition 12.2 *Soit $\sigma = A \Rightarrow \tau \mid C$ un schéma de types parfait. Alors*

- *toute variable de σ est soit négative, soit positive, mais ni neutre ni bipolaire ;*
- *pour tout $\alpha \in \mathrm{dom}^+(\sigma)$, $C^\downarrow(\alpha)$ est simple et $C^\uparrow(\alpha)$ est égal à $\top$;*
- *pour tout $\alpha \in \mathrm{dom}^-(\sigma)$, $C^\downarrow(\alpha)$ est égal à $\perp$ et $C^\uparrow(\alpha)$ est simple ;*
- *si $\alpha \leq_C \beta$, alors $\alpha \in \mathrm{dom}^-(\sigma)$ et $\beta \in \mathrm{dom}^+(\sigma)$.*

Démonstration. Immédiat, en tenant compte de l'invariant de mono-polarité. □

L'algorithme de minimisation, que nous décrirons au chapitre 13, demande un schéma de types parfait et préserve cette propriété. Les schémas de types entièrement simplifiés produit par notre système seront donc sous cette forme.

Cette représentation est particulièrement simple. On peut d'ailleurs formuler une remarque assez intéressante concernant la forme des schémas de types : le « polymorphisme » d'un schéma provient uniquement des contraintes entre variables. Cette affirmation est formalisée et expliquée ci-dessous.

Définition 12.3 *Soit $\sigma = A \Rightarrow \tau \mid C$ un schéma de types parfait. σ est* trivial *ssi il ne contient aucune contrainte entre variables, c'est-à-dire ssi la relation $\leq_C$ est réduite à la diagonale.*

Proposition 12.3 *Soit σ un schéma de types trivial. Alors sa dénotation est un cône.*

Démonstration. Soit σ' le schéma obtenu à partir de σ en ajoutant une contrainte d'égalité entre chaque variable positive (resp. négative) et sa borne inférieure (resp. supérieure) construite. Du fait que σ' ne contient aucune contrainte entre variables, il est trivialement clos. Par conséquent, on peut lui appliquer l'algorithme de dépoussiérage. On constate immédiatement que $\mathrm{GC}(\sigma') = \sigma$. Il en découle que σ et σ' ont la même dénotation.

Or, le graphe de contraintes de σ' ne contient que des équations reliant chaque variable à sa borne construite. En d'autres termes, il s'agit d'un système d'équations contractives. Par conséquent, il admet une unique solution. Le schéma σ' admet donc une unique instance brute ; sa dénotation est le cône engendré par cette instance, c'est-à-dire l'ensemble des types bruts qui sont au-dessus de cette instance pour la relation de sous-typage brut. □

En termes moins techniques, la proposition ci-dessus indique que si σ est un schéma trivial, alors il admet une plus petite instance brute, qui le caractérise entièrement. Le schéma σ n'offre donc essentiellement aucun polymorphisme, même s'il contient des variables universellement quantifiées, puisqu'il admet une instance plus précise que toutes les autres. Cette propriété sera utilisée dans la section 16.1 lors du typage des constructions `let` expansives à *toplevel*.

L'analogue en ML de la notion de schéma de types trivial est la notion de schéma de types sans variables. Celle-ci ne serait pas satisfaisante ici ; dans notre système, certains schémas ne peuvent s'exprimer sans variables, d'une part parce que notre modèle contient des types bruts récursifs, d'autre part parce que nous avons choisi d'utiliser l'invariant des petits termes. La proposition ci-dessus exprime donc le fait que, dans un schéma de types trivial, les variables ne servent pas à offrir du polymorphisme, mais seulement à étiqueter les nœuds de la structure.

En l'absence de contraintes entre variables, un schéma de types se résume donc à un simple point dans l'espace des typages bruts. (Si cet énoncé surprend, rappelons qu'il suppose l'invariant de mono-polarité en vigueur. Par exemple, le schéma $\alpha \to \alpha$ ne contient aucune contrainte entre variables, mais α y est bipolaire. Si on élimine cette variable bipolaire, on obtient $\alpha \to \beta \mid \{\alpha \leq \beta\}$, lequel n'est pas trivial.) Toute la complexité de la relation de comparaison de schémas (dont la décidabilité, rappelons-le, est un problème ouvert), provient donc de la présence de contraintes entre variables.

$$\frac{\alpha, \beta, \gamma \notin F \qquad A = \mathrm{single}_{(x,\alpha,\beta)}(\Gamma)}{[F]\,\Gamma \vdash_{\mathrm{I}} x : [F \cup \{\alpha, \beta, \gamma\}]\ A \Rightarrow \gamma \mid \varnothing + (\alpha \leq \gamma)} \quad (\textsc{Var}_{\mathrm{I}})$$

$$\frac{[F]\ \mathrm{lift}_x(\Gamma); x \vdash_{\mathrm{I}} e : [F']\ (A; x : \tau) \Rightarrow \tau' \mid C \qquad \alpha \notin F'}{[F]\,\Gamma \vdash_{\mathrm{I}} \lambda x.e : [F' \cup \{\alpha\}]\ A \Rightarrow \alpha \mid C + (\tau \rightarrow \tau' \leq \alpha)} \quad (\textsc{Abs}_{\mathrm{I}})$$

$$\frac{\begin{array}{c}[F]\,\Gamma \vdash_{\mathrm{I}} e_1 : [F']\ A_1 \Rightarrow \tau_1 \mid C_1 \\ [F']\,\Gamma \vdash_{\mathrm{I}} e_2 : [F'']\ A_2 \Rightarrow \tau_2 \mid C_2 \qquad \alpha, \gamma \notin F''\end{array}}{\begin{array}{c}[F]\,\Gamma \vdash_{\mathrm{I}} e_1\, e_2 : [F'' \cup \{\alpha, \gamma\}]\ (A_1 \sqcap A_2) \Rightarrow \gamma \mid C \\ \text{où } C = (C_1 \cup C_2) + (\alpha \leq \gamma) + (\tau_1 \leq \tau_2 \rightarrow \alpha)\end{array}} \quad (\textsc{App}_{\mathrm{I}})$$

$$\frac{\Gamma(X) = \sigma \qquad \rho \text{ renommage de } \sigma \qquad \mathrm{im}(\rho) \cap F = \varnothing}{[F]\,\Gamma \vdash_{\mathrm{I}} X : [F \cup \mathrm{im}(\rho)]\ \rho(\sigma)} \quad (\textsc{LetVar}_{\mathrm{I}})$$

$$\frac{\begin{array}{c}[F]\,\Gamma \vdash_{\mathrm{I}} e_1 : [F']\ A_1 \Rightarrow \tau_1 \mid C_1 \\ [F']\,\Gamma; X : A_1 \Rightarrow \tau_1 \mid C_1 \vdash_{\mathrm{I}} e_2 : [F'']\ A_2 \Rightarrow \tau_2 \mid C_2\end{array}}{[F]\,\Gamma \vdash_{\mathrm{I}} \texttt{let}\, X = e_1\ \texttt{in}\ e_2 : [F'']\ (A_1 \sqcap A_2) \Rightarrow \tau_2 \mid C_1 \cup C_2} \quad (\textsc{Let}_{\mathrm{I}})$$

FIG. 12.1: Règles d'inférence, respectant l'invariant de mono-polarité

12.3 Mise en œuvre

Est-il possible de faire respecter l'invariant de mono-polarité à chaque étape du processus d'inférence de types ? Oui, et de façon extrêmement simple. Quelques légères modifications des règles d'inférence de types suffisent à garantir que tout schéma apparaissant dans un jugement d'inférence de types est conforme à l'invariant. Par ailleurs, chacune des phases de simplification (canonisation, dépoussiérage, minimisation) préserve l'invariant. Enfin, nous avons indiqué au chapitre 11 que l'algorithme de canonisation est capable d'éliminer les variables bipolaires d'un schéma donné ; cela permettra de rendre conformes à l'invariant les schémas de types fournis par l'utilisateur, comme ceux que l'on trouve dans les signatures de modules.

Les règles d'inférence modifiées sont données par la figure 12.1. Les seules règles effectivement modifiées ici sont ($\textsc{Var}_{\mathrm{I}}$) et ($\textsc{App}_{\mathrm{I}}$). Les autres sont inchangées, mais nous les rappelons néanmoins, car il s'agit là de la formulation définitive de notre système d'inférence.

Les deux changements effectués ici ont la même explication. Dans les versions précédentes de ($\textsc{Var}_{\mathrm{I}}$) et de ($\textsc{App}_{\mathrm{I}}$), une variable α était créée et utilisée d'une façon qui pouvait éventuellement la rendre bipolaire. En effet, dans le schéma de types $\mathrm{single}_{(x,\alpha,\beta)}(\Gamma) \Rightarrow \alpha \mid \varnothing$, créé par l'ancienne règle ($\textsc{Var}_{\mathrm{I}}$), α est bipolaire, puisqu'elle apparaît en position négative (dans le contexte) et en position positive (dans le corps du schéma). Dans le schéma $(A_1 \sqcap A_2) \Rightarrow \alpha \mid C$, où $C = (C_1 \cup C_2) + (\tau_1 \leq \tau_2 \rightarrow \alpha)$, créé par l'ancienne règle ($\textsc{App}_{\mathrm{I}}$), α est positive (puisqu'elle apparaît dans le corps), et elle risque d'être négative, parce qu'elle apparaît en position positive dans la borne supérieure de τ_1.

Les nouvelles règles évitent ce problème en créant deux variables fraîches, α et γ, au lieu d'une seule, et en les reliant par une contrainte supplémentaire $\alpha \leq \gamma$. Elles seront utilisées de façon à ce que α soit (au plus) négative et γ positive. Aucune ne sera bipolaire. (Rappelons que les signes ne se propagent pas d'une variable à l'autre, donc la contrainte $\alpha \leq \gamma$ ne rend pas nos variables bipolaires.) Il est facile de vérifier que chacune des nouvelles règles produit un schéma de types équivalent à celui produit par sa version précédente. Le nouveau système d'inférence de types est donc équivalent au précédent.

Nous avons donc éliminé les deux points où de nouvelles variables bipolaires pouvaient apparaître. Il reste à vérifier que cela suffit à garantir l'absence de toute variable bipolaire. L'explication essentielle de cette propriété est le fait que la polarité d'une variable décroît au cours de sa vie. En d'autres termes, si une variable n'est pas positive (resp. négative) lors de sa création, alors elle ne le deviendra jamais au cours de la dérivation d'inférence. Ce fait est formalisé par le lemme ci-dessous.

Lemme 12.4 *Soit* $[F]\,\Gamma \vdash_{\mathrm{I}} e : [F']\,\sigma$ *un jugement d'inférence (dérivé à l'aide des règles de la figure 12.1). Ses prémisses contiennent zéro, un ou plusieurs jugements d'inférence ; on note* $(\sigma_i)_{i\in I}$ *les schémas de types qui y apparaissent. Soit* $\alpha \in \mathrm{dom}(\sigma)$*. Si* $\alpha \in \mathrm{dom}(\sigma_i)$ *pour un certain* $i \in I$*, alors* $\alpha \in \mathrm{dom}^+(\sigma)$ *implique* $\alpha \in \mathrm{dom}^+(\sigma_i)$*, et* $\alpha \in \mathrm{dom}^-(\sigma)$ *implique* $\alpha \in \mathrm{dom}^-(\sigma_i)$*.*

Démonstration. Il serait élégant de formuler une preuve basée sur l'intuition énoncée plus haut. Voici, de façon fort grossière, quelles seraient ses grandes lignes. On commence par démontrer qu'une variable est positive si et seulement si elle est susceptible de recevoir une nouvelle borne supérieure à l'avenir, ce qui correspond à l'intuition donnée au chapitre 10. On considère ensuite une variable positive α. Il est donc possible d'exhiber un contexte dans lequel α reçoit une nouvelle borne. Mais alors, la composition de deux contextes étant un contexte, on pouvait en dire autant d'α au moment au de sa création. α était donc positive dès sa création.

Malheureusement, bien que très intuitif, le discours ci-dessus contient des inexactitudes, et nous ne sommes pas certains qu'il soit possible d'en donner une version rigoureuse. Nous préférons donc nous contenter de démontrer l'énoncé ci-dessus, en raisonnant par cas sur la forme de la dernière règle utilisée dans la dérivation de $[F]\,\Gamma \vdash_{\mathrm{I}} e : [F']\,\sigma$. Il y a donc un cas par règle d'inférence. Tous sont aisés, à l'exception de (App$_{\mathrm{I}}$) et (Let$_{\mathrm{I}}$). De plus, comme on l'a expliqué dans les sections 5.3 et 5.5, la règle (Let$_{\mathrm{I}}$) peut être considérée comme la composition d'une règle simplifiée et d'une règle d'application. Nous ne détaillerons donc que le cas de la règle (App$_{\mathrm{I}}$). Nous reprenons exactement les notations de la figure 12.1, ce qui nous permet de ne pas écrire explicitement nos hypothèses.

Le schéma de types produit par la règle est $\sigma = (A_1 \sqcap A_2) \Rightarrow \beta \mid C$, où C est égal à $(C_1 \cup C_2) + (\alpha \leq \gamma) + (\tau_1 \leq \tau_2 \to \alpha)$. Nous désirons à présent effectuer un calcul (approché) des polarités dans σ, de façon à pouvoir les relier aux polarités de σ_1 et σ_2.

Cependant, rappelons que les polarités d'un schéma ne sont définies que si celui-ci est faiblement clos (cf. définition 10.3). Or, tel qu'il est défini ci-dessus, C n'est pas clos. Pour pouvoir parler des polarités de σ, il faut donc d'abord effectuer un calcul de clôture. Le gros de la preuve consiste à simuler ce calcul.

Or, cette simulation est rendue sensiblement plus complexe par l'utilisation des graphes de contraintes, où les bornes multiples sont obligatoirement combinées à l'aide de $\sqcup$ ou $\sqcap$. Cela complique le raisonnement et gêne l'intuition. Aussi, exceptionnellement, nous exprimerons la preuve en termes d'ensembles de contraintes, plutôt que de graphes. Il s'agit là d'un manque de rigueur, mais qui, nous l'espérons, facilite la compréhension de la preuve. Nous supposons donc, dans ce qui suit, que C est un ensemble de contraintes, et nous désirons calculer sa clôture C^*, par transitivité sur les variables et par décomposition structurelle.

Posons $B = C_1 \cup C_2$ et $B' = B \cup \{\alpha \leq \gamma\}$. On suppose que C_1 et C_2 sont des ensembles de contraintes clos, puisqu'ils sont produits par des jugements d'inférence. De plus, $\mathrm{dom}(C_1)$, $\mathrm{dom}(C_2)$ et $\{\alpha, \gamma\}$ sont disjoints deux à deux, donc B' est clos. Reste à ajouter la contrainte $\tau_1 \leq \tau_2 \to \alpha$ et à effectuer le calcul de clôture.

Par convention, si β est une variable appartenant à $\mathrm{dom}(\sigma_i)$ pour un certain $i \in \{1, 2\}$, nous écrirons simplement β^+ (resp. β^-) pour indiquer que $\beta \in \mathrm{dom}^+(\sigma_i)$ (resp. $\mathrm{dom}^-(\sigma_i)$). Une notation similaire est adoptée en ce qui concerne les petits termes.

Essayons de donner l'intuition de la preuve. Une application de fonction engendre une contrainte qui relie un producteur à un consommateur, donc un terme positif à un terme négatif. Le phénomène de clôture provoque l'apparition de contraintes additionnelles, qui sont de la même forme. Il en découle que les nouvelles contraintes ne font pas croître les polarités. En effet, une contrainte peut éventuellement rendre son membre gauche positif, et son membre droit négatif ; or, ici, ils le sont déjà.

Pour calculer C^*, nous considérons toutes les contraintes de C, et nous les combinons par transitivité sur les variables et par décomposition structurelle. Nous prétendons que toutes les contraintes obtenues sont de l'une des formes ci-dessous :

1. $c \in B'$;
2. $\beta \leq \tau_2^+ \to \alpha$, où $\beta \leq_B \delta^+$;
3. $\tau^+ \leq \phi$, où τ est un petit terme, ou $\beta \leq \phi$, où $\beta \leq_B \delta^+$, et $\phi \in \{\alpha, \gamma\}$;
4. $\beta \leq \lambda$, où $\beta \leq_B \delta^+$ et $\epsilon^- \leq_B \lambda$;
5. $\tau^+ \leq \lambda$, où τ est un petit terme, et $\epsilon^- \leq_B \lambda$;
6. $\beta \leq \tau^-$, où τ est un petit terme, et $\beta \leq_B \delta^+$.

Pour prouver cette assertion, on vérifie d'abord que toutes les contraintes présentes au début du calcul sont de l'une de ces formes. C'est le cas de toutes les contraintes de B', qui sont de la forme 1 ; quant à la contrainte $\tau_1^+ \leq \tau_2^+ \to \alpha$, elle est de la forme 2.

Ensuite, il reste à vérifier que l'ensemble décrit ci-dessus est stable par transitivité et par décomposition structurelle. Par exemple, considérons la combinaison par transitivité d'une contrainte de la forme 1 avec une contrainte de la forme 2. La seconde s'écrit $\beta \leq \tau_2^+ \to \alpha$, où $\beta \leq_B \delta^+$. Quant à la première, elle peut être de la forme $\epsilon \leq \beta$, où $\epsilon \leq_B \beta$, ou bien de la forme $\tau \leq \beta$, où τ est un petit terme. Dans le premier cas, la contrainte résultante est $\epsilon \leq \tau_2^+ \to \alpha$, qui est de la forme 2, parce que $\epsilon \leq_B \delta^+$. (Cette dernière assertion s'obtient par transitivité de $\leq_B$.) Dans le second cas, les contraintes résultantes sont obtenues par décomposition structurelle de la contrainte $\tau \leq \tau_2^+ \to \alpha$. Or, toujours par transitivité, B contient la contrainte $\tau \leq \delta^+$, donc τ est positif. Par conséquent, les contraintes obtenues sont de la forme 3, pour le codomaine, et 4, pour le domaine. Les 35 autres cas se traitent de façon similaire, et sont laissés au lecteur.

Ce résultat établi, nous pouvons maintenant vérifier que les polarités ont bien diminué, c'est-à-dire que

$$\mathrm{dom}^+(\sigma) \subset \mathrm{dom}^+(\sigma_1) \cup \mathrm{dom}^+(\sigma_2) \cup \{\gamma\}$$
$$\mathrm{dom}^-(\sigma) \subset \mathrm{dom}^-(\sigma_1) \cup \mathrm{dom}^-(\sigma_2) \cup \{\alpha\}$$

Pour cela, il suffit de vérifier que les ensembles $P = \mathrm{dom}^+(\sigma_1) \cup \mathrm{dom}^+(\sigma_2) \cup \{\gamma\}$ et $N = \mathrm{dom}^-(\sigma_1) \cup \mathrm{dom}^-(\sigma_2) \cup \{\alpha\}$ vérifient les conditions de la définition 10.3. La première demande que $\gamma \in P$, et la seconde que $\mathrm{fv}(A_1 \sqcap A_2) \subset N$; elles sont vérifiées. La troisième demande que pour toute contrainte de la forme $\tau \leq \beta$, où τ est un petit terme et $\beta \in P$, on ait $\mathrm{fv}^+(\tau) \in P$ et $\mathrm{fv}^-(\tau) \in N$. Considérons donc une telle contrainte, et raisonnons par cas sur sa forme, selon la classification donnée plus haut :

1. $c \in B'$. Alors c est présente dans C_i, pour un certain $i \in \{1, 2\}$, donc la condition est vérifiée par définition de $\mathrm{dom}^+(\sigma_i)$ et $\mathrm{dom}^-(\sigma_i)$.
2. Ce cas ne peut pas se présenter, les deux formes considérées étant incompatibles.
3. $\tau^+ \leq \phi$, où τ est un petit terme et $\phi \in \{\alpha, \gamma\}$; alors la condition est vérifiée du fait que τ est positif.
4. Ce cas ne peut pas se présenter.
5. $\tau^+ \leq \lambda$, où τ est un petit terme et $\epsilon^- \leq_B \lambda$; alors la condition est vérifiée du fait que τ est positif.
6. Ce cas ne peut pas se présenter.

La quatrième condition se traite de façon similaire. □

En corollaire, nous obtenons le résultat attendu :

Proposition 12.5 *Si $[F]\ \Gamma \vdash_{\mathrm{I}} e : [F']\ \sigma$ est dérivable dans le système d'inférence exposé par la figure 12.1, alors σ vérifie l'invariant de mono-polarité.*

Démonstration. Il est facile de vérifier qu'aucune variable n'est bipolaire au moment de son introduction. De plus, d'après le lemme 12.4, la polarité d'une variable décroît au cours de la dérivation. Le résultat en découle immédiatement. □

Par ailleurs, on vérifie aisément que les méthodes de simplification introduites jusqu'ici (canonisation, dépoussiérage) respectent l'invariant de mono-polarité. Il en ira de même de l'algorithme de minimisation. Nous supposerons donc, dorénavant, que tous les schémas de types manipulés vérifient cet invariant.

12.4 Remarques

La justification principale de l'invariant de mono-polarité réside dans la garantie de voir le dépoussiérage produire des schémas de types clos. Cette propriété est très agréable, et simplifie sensiblement la théorie. En effet, comme mentionné plus haut, en l'absence de cette propriété, nous serions forcés soit de recalculer la clôture après le dépoussiérage, soit de développer une théorie plus complexe, permettant de travailler avec des graphes de contraintes non clos. Nous avons étudié les deux possibilités, avant de découvrir l'effet de l'invariant de mono-polarité. La première est simple, mais inélégante et inefficace. La seconde semble réalisable et n'entraîne pas d'inefficacité à l'exécution, mais elle demande des preuves assez délicates. En conclusion, l'invariant de mono-polarité semble être de loin la solution la plus simple et la plus élégante à ce problème.

Par ailleurs, cet invariant a d'autres effets intéressants. Par exemple, il contribue à simplifier la définition de l'algorithme de minimisation (cf. chapitre 13). Celui-ci pourrait être formulé en l'absence de l'invariant de mono-polarité ; il suffirait de spécifier qu'une variable bipolaire ne peut être identifiée avec aucune autre variable, c'est-à-dire que sa classe est réduite à un point. En présence de l'invariant, l'algorithme est légèrement plus simple, donc plus efficace, puisque le cas particulier des variables bipolaires ne se présente plus. De plus, il devient légèrement plus puissant. En effet, une variable bipolaire ne peut être identifiée avec aucune autre variable ; tandis que, si nous la transformons en un couple formé d'une variable négative et d'une variable positive, chacune de ces variables peut éventuellement être identifiée avec d'autres. L'invariant de mono-polarité devrait donc offrir quelques opportunités nouvelles à l'algorithme de minimisation. En pratique, l'expérience montre qu'elles sont peu nombreuses ; néanmoins, c'est historiquement pour cette raison que nous avons imaginé cet invariant.

D'anciennes implémentations de notre système contenaient une passe spécifiquement chargée de l'élimination des cycles de variables. Lorsqu'un tel cycle, de la forme $\alpha_1 \leq \ldots \leq \alpha_n \leq \alpha_1$, était détecté, tous les α_i étaient identifiés. On retrouve également cette méthode de simplification dans d'autres travaux [15, 4]. Aujourd'hui, la combinaison du dépoussiérage et de l'invariant de mono-polarité rend cette passe superflue. Supposons en effet qu'un graphe contienne une chaîne de deux contraintes $\alpha \leq \beta \leq \gamma$. Si celle-ci survit au dépoussiérage, alors nous avons $\alpha^- \leq \beta^+$ et $\beta^- \leq \alpha^+$, donc β est bipolaire. Cela est impossible ; par conséquent, il n'existe aucune chaîne après le dépoussiérage, et *a fortiori*, aucun cycle. En fait, quand un cycle apparaît, il est d'abord complété par la clôture ; puis, le dépoussiérage le détruit en ne conservant que les contraintes reliant une variable négative à une variable positive ; enfin, la minimisation identifie tous les variables positives (resp. négatives) du cycle. Le résultat obtenu est donc satisfaisant. Cependant, il serait peut-être souhaitable de détecter et d'éliminer le cycle dès son apparition, de façon à éviter des calculs de clôture superflus. C'est ce que suggèrent Fähndrich *et al.* [17] ; nous avons l'intention d'étudier cette idée.

Une autre conséquence de l'invariant de mono-polarité est le fait qu'il est maintenant illégal de remplacer une variable de types positive (resp. négative) par son unique borne inférieure (resp. supérieure). Le remplacement d'une variable par sa borne a été proposé en tant que méthode de simplification dans de nombreux articles [15, 4, 8, 43]. Pourquoi donc est-ce illégal ici ? Nous avons mentionné au chapitre 6 que le fait de remplacer une variable par un terme construit viole l'invariant des petits termes. Restait, jusqu'ici, le cas où la borne en question était également une variable. Supposons, par exemple, que α soit une variable positive, dont l'unique borne inférieure est une variable β. Du fait que la contrainte $\beta \leq \alpha$ a survécu au dépoussiérage, nous déduisons que β est négative. Par conséquent, si nous identifions les variables α et β, nous créons une variable bipolaire, et violons ainsi l'invariant de mono-polarité.

Ainsi, il est à présent entièrement interdit de remplacer une variable par son unique borne au cours du processus d'inférence et de simplification. Soulignons, cependant, que cela n'est pas un désavantage de notre système. D'abord, la perte en espace n'est que d'un facteur deux au pire, et il n'y a pas de perte d'efficacité, puisqu'au contraire, nous prétendons que l'invariant a plusieurs effets bénéfiques. Ensuite, bien que cette stratégie de substitution soit interdite au sein du moteur d'inférence, elle reste correcte d'un point de vue théorique, et pourra donc être utilisée, une fois le processus d'inférence terminé, pour améliorer la lisibilité du résultat. Pour conclure, nous avons montré que les deux objectifs de la simplification de types, à savoir lisibilité et efficacité, sont contradictoires, puisque l'efficacité demande de faire respecter certains invariants (petits termes et mono-polarité) qui vont à l'encontre de la lisibilité. Il convient donc de favoriser l'efficacité tout au long du processus d'inférence, et de n'améliorer la lisibilité que lors d'une passe finale, avant affichage.

Chapitre 13

Minimisation

Jusqu'ici, nous avons présenté peu de méthodes de simplification. D'abord, la canonisation, qui est une simplification, au sens où elle permet ensuite d'appliquer certains algorithmes incapables de fonctionner en présence des constructeurs $\sqcup$ et $\sqcap$. Ensuite, le dépoussiérage, qui se contente de détecter des contraintes superflues et de les éliminer. Or, si on parcourt la littérature concernant la simplification de contraintes, on trouve de nombreuses méthodes fondées sur l'idée de *substitution*, qui consiste à remplacer une variable par un terme sans modifier la dénotation du schéma du types considéré.

Ce chapitre est consacré à l'exposé d'une méthode de simplification, appelée *minimisation*, qui généralise toutes les méthodes de substitution connues, tout en étant remarquablement efficace. Le principe nous en a été inspiré par la méthode dite « de Hopcroft » chez Felleisen et Flanagan [19, 20].

La section 13.1 pose le problème et décrit les grandes lignes de l'algorithme de minimisation. Nous donnons ensuite une spécification formelle de la minimisation (section 13.2), puis passons à sa mise en œuvre (section 13.3). Viennent ensuite quelques exemples d'applications (section 13.4). Enfin, la section 13.5 évoque le problème de complétude de cette méthode.

13.1 Introduction

Parmi les méthodes de substitution connues, on trouve des algorithmes spécialisés, efficaces mais ne traitant qu'un cas particulier bien précis ; et des heuristiques, *a priori* relativement puissantes, mais souvent très inefficaces.

Parmi les algorithmes non heuristiques, on peut citer l'élimination des cycles de variables, ou encore le remplacement d'une variable positive (resp. négative) par son unique borne inférieure (resp. supérieure). Or, nous avons remarqué que, si l'invariant de mono-polarité est respecté, la première est automatiquement effectuée par l'algorithme de dépoussiérage (cf. section 12.4). Quant au second, nous avons constaté qu'il viole l'invariant des petits termes ou l'invariant de mono-polarité (cf. sections 6.4 et 12.4), et est donc illégal, du moins tant que le processus d'inférence est en cours – il reste valide si on le destine à l'affichage uniquement.

Toutes les heuristiques fonctionnent sur un schéma commun, en deux étapes. D'abord, on invente, d'une façon ou d'une autre, une substitution ρ. Ensuite, on détermine s'il est légal de l'appliquer au schéma de types considéré σ, c'est-à-dire si $\rho(\sigma) =^\forall \sigma$. Ce test peut être réalisé à l'aide de l'algorithme développé au chapitre 9, ou éventuellement à l'aide d'algorithmes moins puissants, comme l'axiomatisation de l'implication de contraintes donnée au chapitre 8. Nous avons proposé quelques heuristiques fondées sur ce dernier dans [43]. Cependant, le nombre de substitutions possibles est énorme. (On peut bien sûr se restreindre à des cas particuliers, mais ils sont alors souvent assez *ad hoc*.) De plus, l'algorithme d'implication utilisé lors de la seconde étape est relativement coûteux. La combinaison de ces deux facteurs donne donc des méthodes de simplification très inefficaces.

Ainsi, il semble souhaitable de mettre au point un algorithme systématique, capable de construire directement une substitution aussi puissante que possible. Précisons d'abord cette notion de puissance. On constate que, pour respecter l'invariant des petits termes, nous ne pouvons remplacer une variable que par une autre variable. Ainsi, une substitution peut être vue comme une partition de l'ensemble des variables, deux variables devant être identifiées ssi elles appartiennent à la même classe. On peut alors dire qu'une substitution est moins puissante qu'une autre ssi la partition qui lui est associée est plus fine. En tenant compte du fait que l'on doit respecter l'invariant de mono-polarité, la substitution la plus puissante est alors celle qui ne contient que deux classes, $\mathrm{dom}^+(\sigma)$ et $\mathrm{dom}^-(\sigma)$; en d'autres termes, celle qui identifie toutes les variables positives (resp. négatives) entre elles.

Nous sommes donc à la recherche d'une substitution ρ plus fine que celle décrite ci-dessus (de façon à respecter l'invariant de mono-polarité), correcte (c'est-à-dire telle que $\rho(\sigma) =^\forall \sigma$), et de puissance maximale pour ces critères. Cependant, la relation $=^\forall$ étant complexe, il n'est pas certain qu'une telle substitution existe dans le cas général. Nous allons donc remplacer le critère de correction par un critère légèrement plus restrictif, mais plus simple, car directement lié à la forme des contraintes.

Pour imaginer ce critère, on peut esquisser, de façon très informelle, un algorithme capable d'effectuer la construction. On part de la substitution la plus puissante. On peut exécuter l'algorithme de comparaison de schémas pour déterminer si cette substitution est correcte. S'il signale un échec, on en détermine la cause, et on affaiblit la substitution en coupant en deux une classe appropriée, de façon à éliminer le problème. On répète ensuite ce processus jusqu'à ce qu'aucune erreur ne soit plus signalée.

Toujours de façon informelle, on peut dire qu'un tel échec se produit lorsqu'on tente d'identifier deux variables jouant des rôles différents dans le schéma de types. Or, le « rôle » d'une variable est déterminé par la façon dont elle est reliée, à travers des contraintes, aux autres variables. On pourrait donc dire que deux variables peuvent être identifiées si chacune d'entre elles porte le même type de liens que son homologue, et si ces liens mènent à des variables qui peuvent à leur tour être identifiées.

On retrouve là des notions familières. Le raffinement d'une partition décrit plus haut rappelle le procédé utilisé lors de la minimisation des automates finis. Le critère informel donné par le paragraphe précédent est une forme de bisimulation, un concept que l'on retrouve également en théorie des automates. D'où l'idée de *minimiser* un graphe de contraintes, un peu comme l'on minimise un automate. Comme nous allons le voir à présent, cette idée a un sens, et permet même de réutiliser les algorithmes efficaces mis au point dans le cadre de la minimisation des automates finis.

L'application de techniques de minimisation aux systèmes de contraintes est due à Felleisen et Flanagan [19, 20]. Leur définition de la compatibilité (cf. définition 13.3) est quelque peu différente, et en particulier moins symétrique, parce leur langage de contraintes est assez éloigné du nôtre, de par sa syntaxe comme par sa sémantique. Néanmoins, le principe est identique. Il est indépendant de l'utilisation du sous-typage, et devrait pouvoir s'appliquer à de nombreux systèmes à base de contraintes récursives, quelle que soit leur sémantique.

13.2 Formalisation

Nous allons à présent donner une spécification formelle de l'algorithme, et en prouver la correction. Voici d'abord quelques définitions préliminaires.

Définition 13.1 *Etant donné un ensemble de variables V, relations d'équivalence sur V et partitions de V sont isomorphes. Toute relation $\equiv$ s'étend aux petits termes à variables dans V en posant*

$$\begin{aligned} \bot &\equiv \bot \\ \top &\equiv \top \\ \alpha_0 \to \alpha_1 \equiv \beta_0 \to \beta_1 &\iff (\alpha_0 \equiv \beta_0) \wedge (\alpha_1 \equiv \beta_1) \end{aligned}$$

Définition 13.2 *Soit C un graphe de contraintes. Pour $\alpha \in \mathrm{dom}(C)$, on pose*

$$\mathrm{pred}_C(\alpha) = \{\beta\,;\, \beta \leq_C \alpha\}$$
$$\mathrm{succ}_C(\alpha) = \{\beta\,;\, \alpha \leq_C \beta\}$$

Nous avons mentionné, dans la section 13.1, que la correction d'une substitution est un critère trop complexe – peut-être même indécidable – puisqu'elle met en jeu la relation $=^\forall$. Nous introduisons donc une notion de *compatibilité*, plus faible, mais dont la définition, purement syntaxique, mène directement à un algorithme. Tout notre développement sera ensuite basé sur cette notion.

Définition 13.3 *Soit $\sigma = A \Rightarrow \tau \mid C$ un schéma de types parfait. Une relation d'équivalence $\equiv$, de domaine* $\mathrm{dom}(\sigma)$, *est* compatible *avec σ ssi*

- $\alpha \equiv \beta$ *implique* $\mathrm{pred}_C(\alpha) = \mathrm{pred}_C(\beta)$ *et* $\mathrm{succ}_C(\alpha) = \mathrm{succ}_C(\beta)$ *;*
- $\alpha \equiv \beta$ *implique* $C^\downarrow(\alpha) \equiv C^\downarrow(\beta)$ *et* $C^\uparrow(\alpha) \equiv C^\uparrow(\beta)$ *;*
- $\alpha \equiv \beta$ *implique* $\mathrm{polarité}_\sigma(\alpha) = \mathrm{polarité}_\sigma(\beta)$.

Lemme 13.1 *La première condition ci-dessus peut également être formulée de la façon suivante :*

- $\alpha \leq_C \beta$ *implique* $\forall \alpha' \equiv \alpha \quad \forall \beta' \equiv \beta \quad \alpha' \leq_C \beta'$.

Démonstration. Supposons que la première condition soit vérifiée. Supposons $\alpha \leq_C \beta$, $\alpha' \equiv \alpha$ et $\beta' \equiv \beta$. Nous avons $\beta \in \mathrm{succ}_C(\alpha)$; d'après notre hypothèse, nous obtenons $\beta \in \mathrm{succ}_C(\alpha')$, c'est-à-dire $\alpha' \leq_C \beta$. En appliquant encore une fois l'hypothèse à ce résultat, nous obtenons $\alpha' \leq_C \beta'$, comme attendu.

Réciproquement, supposons que la variante de la première condition soit vérifiée. Supposons $\alpha \equiv \beta$, et $\alpha' \in \mathrm{pred}_C(\alpha)$. Alors $\alpha' \leq_C \alpha$. En appliquant l'hypothèse, nous obtenons $\alpha' \leq_C \beta$. Nous avons démontré que $\mathrm{pred}_C(\alpha) \subset \mathrm{pred}_C(\beta)$. Par symétrie, ces ensembles sont égaux. De même, on montre que $\mathrm{succ}_C(\alpha) = \mathrm{succ}_C(\beta)$. □

Il est facile de vérifier qu'il existe une unique partition compatible avec un schéma donné, et de puissance maximale pour ce critère. C'est celle-ci qui sera calculée par l'algorithme de minimisation.

Proposition 13.2 *Soit σ un schéma de types parfait. L'ensemble des partitions compatibles avec σ admet un plus grand élément, noté $\equiv_\sigma$.*

Démonstration. D'abord, il est clair que la partition triviale, où tous les points sont isolés, est compatible. Vérifions ensuite que la fusion $\equiv$ de deux partitions compatibles $\equiv_1$ et $\equiv_2$ est elle-même compatible. Rappelons que $\equiv$ est définie comme la clôture transitive de la relation $\equiv_{12} = \equiv_1 \cup \equiv_2$.

Considérons la première condition de la définition 13.3. Elle est vérifiée par $\equiv_1$ et $\equiv_2$ (puisque celles-ci sont compatibles), donc également par leur union $\equiv_{12}$. La relation $\equiv_{12}$ est donc incluse dans la relation «avoir mêmes successeurs et prédécesseurs». Or, cette dernière est transitive ; elle contient donc également la clôture transitive de $\equiv_{12}$, à savoir $\equiv$. La première condition est donc vérifiée. Les deuxième et troisième conditions se traitent de façon similaire.

L'ensemble des partitions compatibles étant non vide et clos par fusion, il admet un plus grand élément. □

Pour faire le lien entre partition et substitution, définissons le *quotient* d'un schéma de types par une partition, c'est-à-dire l'opération qui identifie toutes les variables d'une même classe.

Définition 13.4 *Soient $\sigma = A \Rightarrow \tau \mid C$ un schéma de types parfait, et $\equiv$ une partition compatible avec σ. Dans chaque classe, on choisit un représentant ; en d'autres termes, soit π une fonction de* $\mathrm{dom}(\sigma)$ *dans lui-même telle que*

– $\forall \alpha \in \mathrm{dom}(\sigma) \quad \pi(\alpha) \equiv \alpha$;
– $\forall \alpha, \beta \in \mathrm{dom}(\sigma) \quad \alpha \equiv \beta \iff \pi(\alpha) = \pi(\beta)$.

Soit $\sigma/_{\equiv}$ *le schéma de types* $\pi(\sigma)$. *Plus précisément,* $\sigma/_{\equiv} = \pi(A) \Rightarrow \pi(\tau) \mid \pi(C)$ *où* $\pi(C)$ *est le graphe de contraintes de domaine* $\pi(\mathrm{dom}(\sigma))$ *défini par*

– $\pi(\alpha) \leq_{\pi(C)} \pi(\beta)$ *ssi* $\alpha \leq_C \beta$;
– $(\pi(C))^{\downarrow}(\pi(\alpha)) = \pi(C^{\downarrow}(\alpha))$ *et* $(\pi(C))^{\uparrow}(\pi(\alpha)) = \pi(C^{\uparrow}(\alpha))$.

Le graphe $\pi(C)$ est défini ici par la donnée de deux propriétés caractéristiques. Nous devons donc vérifier qu'il existe un unique objet les satisfaisant.

Démonstration. Lorsque α et β parcourent tout $\mathrm{dom}(\sigma)$, $\pi(\alpha)$ et $\pi(\beta)$ parcourent tout $\pi(\mathrm{dom}(\sigma))$. Ces deux propriétés spécifient donc un objet unique. Cependant, chaque point de $\pi(\mathrm{dom}(\sigma))$ peut être parcouru plusieurs fois ; pour prouver l'existence de l'objet, il faut vérifier l'absence de contradiction entre ces diverses visites.

Considérons la première propriété. Supposons $\pi(\alpha) = \pi(\alpha')$ et $\pi(\beta) = \pi(\beta')$. Alors, par définition de π, $\alpha \equiv \alpha'$ et $\beta \equiv \beta'$. Puisque $\equiv$ est compatible avec σ, cela implique $(\alpha \leq_C \beta) \iff (\alpha' \leq_C \beta')$.

Passons à la seconde. Supposons $\pi(\alpha) = \pi(\alpha')$. Alors $\alpha \equiv \alpha'$. Puisque $\equiv$ est compatible, cela implique $C^{\downarrow}(\alpha) \equiv C^{\downarrow}(\alpha')$, qui à son tour implique $\pi(C^{\downarrow}(\alpha)) = \pi(C^{\downarrow}(\alpha'))$. □

On notera que la définition de $\sigma/_{\equiv}$ dépend du choix de π. (Rappelons que π est défini en choisissant un représentant non spécifié dans chaque classe.) Cependant, les différents choix de π mènent à des schémas de types α-équivalents. $\sigma/_{\equiv}$ est donc défini à α-conversion près. Cela ne posera pas de problème, car les propriétés qui nous intéressent sont indépendantes du choix de π.

Nous pouvons maintenant passer à la preuve de correction, qui indique que toute partition compatible conduit à un schéma quotient équivalent au schéma initial.

Théorème 13.1 *Soient* σ *un schéma de types parfait et* $\equiv$ *une partition compatible avec* σ. *Alors*

$$\sigma/_{\equiv} =^{\forall} \sigma$$

Démonstration. L'assertion $\sigma \leq^{\forall} \sigma/_{\equiv}$ est triviale, puisque le second est l'image du premier à travers la substitution π. Plus précisément, si ρ est solution de $\pi(C)$, alors $\rho \circ \pi$ est solution de C et constitue un témoin de cette assertion.

Réciproquement, nous souhaitons montrer que $\sigma/_{\equiv} \leq^{\forall} \sigma$. Soit ρ une solution de C. Définissons ρ' par

$$\rho'(\pi(\alpha)) = \bigsqcap_{\alpha' \equiv \alpha} \rho(\alpha') \text{ si } \pi(\alpha) \in \mathrm{dom}^+(\sigma)\,;$$
$$\rho'(\pi(\alpha)) = \bigsqcup_{\alpha' \equiv \alpha} \rho(\alpha') \text{ si } \pi(\alpha) \in \mathrm{dom}^-(\sigma).$$

Nous devons vérifier que cette définition a un sens. D'abord, notons que chaque cas est traité une fois et une seule, puisque σ n'a ni variables neutres ni variables bipolaires. Ensuite, si $\pi(\alpha) = \pi(\beta)$, alors $\alpha \equiv \beta$, donc $\{\alpha' \,;\, \alpha' \equiv \alpha\} = \{\beta' \,;\, \beta' \equiv \beta\}$. Par conséquent, ρ' est bien défini.

Vérifions à présent que ρ' est solution de $\pi(C)$. D'abord, choisissons une contrainte de $\leq_{\pi(C)}$, que nous pouvons noter $\pi(\alpha) \leq_{\pi(C)} \pi(\beta)$. D'après la définition 13.4, nous avons

$$\forall \alpha' \equiv \alpha \quad \forall \beta' \equiv \beta \quad \alpha' \leq_C \beta'$$

Puisque ρ est solution de C, ceci implique

$$\forall \alpha' \equiv \alpha \quad \forall \beta' \equiv \beta \quad \rho(\alpha') \leq \rho(\beta')$$

que l'on peut réécrire

$$\bigsqcup_{\alpha' \equiv \alpha} \rho(\alpha') \leq \bigsqcap_{\beta' \equiv \beta} \rho(\beta')$$

Or, $\alpha' \leq_C \beta'$ implique $\alpha' \in \mathrm{dom}^-(\sigma)$ et $\beta' \in \mathrm{dom}^+(\sigma)$, parce que σ est parfait et ne contient donc que des contraintes entre une variable négative et une variable positive. Puisque $\equiv$ est compatible avec σ, et puisque $\alpha' \equiv \pi(\alpha)$, α' a la même polarité que $\pi(\alpha)$. Cela implique $\pi(\alpha) \in \mathrm{dom}^-(\sigma)$. De même, $\pi(\beta) \in \mathrm{dom}^+(\sigma)$. Dans ces conditions, l'assertion ci-dessus peut être lue

$$\rho'(\pi(\alpha)) \leq \rho'(\pi(\beta))$$

ce qui indique que ρ' satisfait la contrainte choisie.

Ensuite, nous devons vérifier que $\rho'((\pi(C))^{\downarrow}(\pi(\alpha))) \leq \rho'(\pi(\alpha))$ pour tout α. Par définition de $\pi(C)$, cela est équivalent à $\rho'(\pi(C^{\downarrow}(\alpha))) \leq \rho'(\pi(\alpha))$. Si $\alpha \in \mathrm{dom}^-(\sigma)$, alors $C^{\downarrow}(\alpha) = \bot$ (parce que σ est parfait) et le résultat est immédiat. Supposons donc $\alpha \in \mathrm{dom}^+(\sigma)$. Le but devient alors

$$\rho'(\pi(C^{\downarrow}(\alpha))) \leq \bigsqcap_{\alpha' \equiv \alpha} \rho(\alpha')$$

que l'on peut réécrire

$$\forall \alpha' \equiv \alpha \quad \rho'(\pi(C^{\downarrow}(\alpha))) \leq \rho(\alpha')$$

Notons que $\alpha' \equiv \alpha$ implique $C^{\downarrow}(\alpha') \equiv C^{\downarrow}(\alpha)$, puisque $\equiv$ est compatible avec σ. Grâce à cette remarque, il suffit de montrer que pour tout $\alpha' \equiv \alpha$,

$$\rho'(\pi(C^{\downarrow}(\alpha'))) \leq \rho(\alpha')$$

Puisque ρ est solution de C, nous avons $\rho(C^{\downarrow}(\alpha')) \leq \rho(\alpha')$. Donc, il suffit de montrer que

$$\rho'(\pi(C^{\downarrow}(\alpha'))) \leq \rho(C^{\downarrow}(\alpha'))$$

Raisonnons à présent par cas sur $C^{\downarrow}(\alpha')$. S'il vaut $\bot$ ou $\top$, le résultat est immédiat. Supposons donc $C^{\downarrow}(\alpha') = \alpha'_0 \to \alpha'_1$. Le but se décompose en deux sous-buts, dont le premier est

$$\rho'(\pi(\alpha'_1)) \leq \rho(\alpha'_1)$$

(Le second étant symétrique, nous ne le traiterons pas explicitement.) Or, puisque $\alpha \in \mathrm{dom}^+(\sigma)$ et $\alpha' \equiv \alpha$, nous avons $\alpha' \in \mathrm{dom}^+(\sigma)$. D'après la définition 10.3, ceci implique $\alpha'_1 \in \mathrm{dom}^+(\sigma)$. Donc, $\pi(\alpha'_1) \in \mathrm{dom}^+(\sigma)$. Le but peut s'écrire

$$\bigsqcap_{\beta \equiv \alpha'_1} \rho(\beta) \leq \rho(\alpha'_1)$$

qui est immédiat, puisque le membre droit apparaît comme l'un des opérandes du membre gauche.

Symétriquement, on vérifie que $\rho'(\pi(\alpha)) \leq \rho'((\pi(C))^{\uparrow}(\pi(\alpha)))$ est vrai pour tout α. Ainsi, ρ' est solution de $\pi(C)$. Il reste à vérifier que $\rho'(\pi(A \Rightarrow \tau)) \leq \rho(A \Rightarrow \tau)$, ce qui est aisé, à l'aide des mêmes techniques que dans les paragraphes précédents. Pour conclure, ρ' est un témoin de l'assertion $\sigma/_{\equiv} \leq^{\forall} \sigma$. Le théorème en découle. □

13.3 Algorithme

Nous avons spécifié une méthode de simplification, qui consiste à quotienter un schéma parfait σ par la plus puissante partition compatible, remplaçant ainsi σ par $\sigma/_{\equiv_\sigma}$. Reste à savoir si ce calcul peut être effectué de façon efficace. La réponse est positive – nous pouvons calculer $\equiv_\sigma$ en temps $O(dn \log n)$, où n est le nombre de variables dans σ, et d est le degré du graphe $\leq_C$ (c'est-à-dire le nombre maximum de branches sortant ou entrant dans un nœud donné).

Pour définir l'algorithme, nous allons reformuler légèrement la définition de la compatibilité, de façon à rendre plus apparentes les différentes étapes nécessaires au calcul de $\equiv_\sigma$. Cette définition indique que si $\equiv$ est compatible, alors deux variables équivalentes ont les mêmes ensembles de prédécesseurs et de successeurs, des bornes construites équivalentes, et la même polarité. La seconde condition est la plus délicate, puisqu'elle est « récursive », c'est-à-dire que la relation $\equiv$ apparaît des deux côtés de l'implication. De plus, la relation $\equiv$ y est utilisée pour comparer deux petits termes ; pour simplifier les choses, nous nous ramenons d'abord à des comparaisons entre variables.

Lemme 13.3 *Soit $\sigma = A \Rightarrow \tau \mid C$ un schéma de types parfait. Une partition $\equiv$ est compatible avec σ ssi*

- $\alpha \equiv \beta$ *implique* $\mathrm{pred}_C(\alpha) = \mathrm{pred}_C(\beta)$ *et* $\mathrm{succ}_C(\alpha) = \mathrm{succ}_C(\beta)$ *;*
- $\alpha \equiv \beta$ *implique* $\mathrm{polarité}_\sigma(\alpha) = \mathrm{polarité}_\sigma(\beta)$ *;*
- $\alpha \equiv \beta$ *implique* $\mathrm{tête}(C^\downarrow(\alpha)) = \mathrm{tête}(C^\downarrow(\beta))$ *et* $\mathrm{tête}(C^\uparrow(\alpha)) = \mathrm{tête}(C^\uparrow(\beta))$ *;*
- *si* $\alpha \equiv \beta$, $C^\downarrow(\alpha) = \alpha_0 \to \alpha_1$ *et* $C^\downarrow(\beta) = \beta_0 \to \beta_1$, *alors* $\alpha_0 \equiv \beta_0$ *et* $\alpha_1 \equiv \beta_1$ *;*
- *si* $\alpha \equiv \beta$, $C^\uparrow(\alpha) = \alpha_0 \to \alpha_1$ *et* $C^\uparrow(\beta) = \beta_0 \to \beta_1$, *alors* $\alpha_0 \equiv \beta_0$ *et* $\alpha_1 \equiv \beta_1$.

Les trois premières conditions ci-dessus ne posent aucun problème, puisqu'elles définissent en fait une partition initiale, qu'il faudra ensuite raffiner. Les deux dernières conditions rappellent la définition d'une bisimulation, comme mentionné dans la section 13.1. Nous sommes donc prêts à définir l'algorithme.

Théorème 13.2 *Soit $\sigma = A \Rightarrow \tau \mid C$ un schéma de types parfait. Alors $\equiv_\sigma$ se calcule en temps $O(dn \log n)$, où $n = |\,\mathrm{dom}(\sigma)\,|$, et d est le degré du graphe $\leq_C$.*

Démonstration. Le calcul se fait en deux étapes : calcul d'une partition initiale, puis raffinement de celle-ci jusqu'à atteindre un point fixe.

Soit $\equiv_0$ la plus grossière partition vérifiant les trois premières conditions du lemme 13.3. Elle se calcule comme suit. D'abord, on construit une liste des variables de σ, triée selon les clefs

$$\alpha \mapsto (\mathrm{pred}_C(\alpha), \mathrm{succ}_C(\alpha), \mathrm{polarité}_\sigma(\alpha), \mathrm{tête}(C^\downarrow(\alpha)), \mathrm{tête}(C^\uparrow(\alpha)))$$

(N'importe quel ordre sur ces quintuplets convient ; l'ordre lexicographique est le plus naturel.) La construction de la liste demande un temps $O(n)$. Le tri demande $O(n \log n)$ comparaisons. On suppose les données ci-dessus disponibles dans le graphe de contraintes, donc la fonction de comparaison peut y accéder en temps constant. Les ensembles $\mathrm{pred}_C(\alpha)$ et $\mathrm{succ}_C(\alpha)$ ont un cardinal inférieur ou égal à d, par définition de d. Donc, la comparaison de deux clefs demande un temps $O(d)$, et le tri un temps $O(dn \log n)$. Une fois la liste triée disponible, une simple passe suffit à créer $\equiv_0$, puisque les éléments d'une même classe sont nécessairement adjacents dans la liste. Cette passe demande $O(n)$ comparaisons, donc prend un temps $O(dn)$.

Notons que $\equiv_\sigma$ est la plus grossière partition plus fine que $\equiv_0$ et vérifiant les deux dernières conditions du lemme 13.3. Pour $i \in \{0, 1\}$, soient Low_i et Upp_i les fonctions partielles sur $\mathrm{dom}(\sigma)$ définies par

$$\mathrm{Low}_i(\alpha) = \alpha_i \text{ si } C^\downarrow(\alpha) = \alpha_0 \to \alpha_1$$
$$\mathrm{Upp}_i(\alpha) = \alpha_i \text{ si } C^\uparrow(\alpha) = \alpha_0 \to \alpha_1$$

Nous pouvons maintenant reformuler le problème : $\equiv_\sigma$ est la plus grossière partition plus fine que $\equiv_0$ et stable vis-à-vis de ces quatre fonctions. (Une partition $\equiv$ est *stable* vis-à-vis d'une fonction f ssi pour tout bloc (i.e. classe) B de $\equiv$, soit f est non définie sur tout B, soit f est définie sur tout B et $f(B)$ est entièrement inclus dans un certain bloc B'.)

Il s'agit donc de trouver le raffinement le plus grossier d'une partition donnée qui soit stable vis-à-vis d'un nombre fini de fonctions données. Il s'agit là d'un problème bien connu.

Il apparaît, en particulier, lors de la minimisation des automates finis ; Hopcroft [30] donne un algorithme pour le résoudre en temps $O(n \log n)$. Un autre article intéressant, sur ce même sujet, est [38], bien qu'il traite d'un problème plus général.

Pour conclure, l'étape de tri initiale prend un temps $O(dn \log n)$, tandis que l'étape de raffinement s'effectue en temps $O(n \log n)$. Le résultat en découle. $\square$

Il serait agréable de disposer d'une borne de complexité exprimée en fonction de N, la taille du schéma de types σ, plutôt que de d et n. Mais, dans le cas le pire, d et n peuvent être en même temps de l'ordre de N ; ce qui nous donne une borne de $O(N^2 \log N)$.

Cependant, cette borne est grossière, car si d et n sont de l'ordre de N, alors peu de nœuds dans le graphe peuvent avoir le degré d. Dans ce cas, le degré *moyen* δ est nettement inférieur au degré maximum. Cela semble indiquer que nous devrions baser notre analyse sur δ, et non sur d. En effet, informellement parlant, si le degré moyen est δ, alors une comparaison entre deux ensembles de successeurs ou de prédécesseurs prend un temps $O(\delta)$ en moyenne. Donc, la première étape de l'algorithme demande un temps $O(\delta n \log n)$. Or, par définition du degré moyen, δn est le nombre d'arcs du graphe $\leq_C$, donc cette borne est inférieure à $O(N \log N)$. Par ailleurs, la seconde étape de l'algorithme fonctionne en temps $O(n \log n)$, qui est également inférieur à $O(N \log N)$. Nous concluons donc, informellement, que le calcul prend un temps $O(N \log N)$ en moyenne. Ce résultat est corroboré par quelques tests pratiques, sur des exemples de taille arbitraire, qui suggèrent que le comportement de l'algorithme est linéaire par rapport à la taille du schéma de types.

D'un point de vue pratique, notons que la complexité de l'algorithme de Hopcroft, si on effectue un raffinement vis-à-vis de k fonctions, est $O(k^2 n \log n)$. Il est donc important de réduire k autant que possible. La preuve du théorème 13.2 utilise $k = 4$ fonctions ; en fait, il est possible de se restreindre à deux fonctions. En effet, la fonction Low_i (pour $i \in \{0, 1\}$) n'est définie que sur les variables positives, tandis que Upp_i n'est définie que sur les variables négatives. Leur réunion $\mathrm{Low}_i \cup \mathrm{Upp}_i$ est donc une fonction. De plus, la partition initiale $\equiv_0$ sépare les variables positives des variables négatives. Par conséquent, le raffinement de $\equiv_0$ par les deux fonctions Low_i et Upp_i est identique au raffinement de $\equiv_0$ par leur réunion. La même idée s'applique lorsque l'on étend le système de types. Par exemple, si l'on introduit des types produits, on peut conserver $k = 2$ fonctions. En effet, la partition initiale sépare les variables d'après le constructeur de tête de leur borne. Par conséquent, pendant l'étape de raffinement, on peut utiliser l'indice 0 (resp. 1) pour représenter à la fois le domaine (resp. codomaine) des types fonctions et la première (resp. seconde) composante des types produits. De façon générale, si les constructeurs de types présents dans le graphe sont d'arité k au plus, alors k fonctions suffisent.

Cette remarque est particulièrement importante lors de l'introduction de types enregistrements (ou variantes). En effet, l'arité maximale k n'est alors plus bornée, ce qui augmente, en théorie, la complexité de l'algorithme. Cependant, grâce à notre remarque, k sera égal au nombre maximum d'étiquettes apparaissant au sein d'un même terme dans le schéma de types considéré, et non au nombre total d'étiquettes apparaissant dans le schéma. En pratique, k peut donc être considéré comme borné, car si un programme utilise souvent un très grand nombre d'étiquettes, il utilise rarement des enregistrements de très grande taille.

13.4 Exemples

L'algorithme de réduction est très versatile, et simplifie avec succès de nombreux problèmes. En voici quelques exemples typiques. (Pour améliorer la lisibilité, l'invariant des petits termes n'est pas entièrement respecté, et on utilisera des ensembles de contraintes, au lieu de graphes.)

Exemple (Repliage d'un point fixe). Nous supposons ici que F est un opérateur de types covariant, distinct de l'identité. (Par exemple, on peut prendre $F : \alpha \mapsto \top \to \alpha$.) Soit σ le schéma

$$\alpha^- \to \bot \mid \{\alpha^- \leq F\,\beta^-,\, \beta^- \leq F\,\beta^-\}$$

Alors, il est facile de vérifier que $\alpha \equiv_\sigma \beta$, puisque ces deux variables sont négatives, et qu'elles ont des bornes équivalentes. Donc, σ peut être remplacé par sa version simplifiée $\sigma/_{\equiv_\sigma}$, qui est

$$\alpha^- \to \bot \mid \{\alpha^- \leq F\,\alpha^-\}$$

Essentiellement, l'algorithme vient de simplifier un type récursif partiellement déplié. En effet, rappelons que le dépoussiérage élimine la borne inférieure d'une variable négative ; par conséquent, σ est équivalent à

$$\alpha \to \bot \mid \{\alpha = F\,\beta,\, \beta = F\,\beta\}$$

Ici, β représente en fait le point fixe de F, et α est égal à $F\,\beta$, donc nous pouvons – informellement – réécrire σ en

$$\alpha \to \bot \mid \{\alpha = F(\mu t.F\,t)\}$$

Ici, il est clair qu'une simplification peut être effectuée. Le type $F(\mu t.F\,t)$ est égal à $\mu t.F\,t$, par repliage du lieur μ. Si nous effectuons la simplification, nous obtenons

$$\alpha \to \bot \mid \{\alpha = \mu t.F\,t\}$$

et en revenant en arrière, nous constatons que ce schéma de types est équivalent à

$$\alpha \to \bot \mid \{\alpha \leq F\,\alpha\}$$

qui n'est autre que le schéma produit par l'algorithme de minimisation.

Nous avons donc montré que l'effet de la minimisation peut être vu comme une étape de repliage d'un point fixe, c'est-à-dire un remplacement de $F(\mu t.F\,t)$ par $\mu t.F\,t$. Bien sûr, l'algorithme est capable d'effectuer plusieurs étapes de repliage d'un seul coup. Cela est très utile en pratique. Imaginons, par exemple, une fonction qui prend un arbre binaire en argument, examine ses deux premiers niveaux (par exemple, dans le but d'effectuer un rééquilibrage), puis utilise des appels récursifs pour traiter les sous-arbres à la profondeur 2. Le domaine du type inféré pour cette fonction sera un type récursif (le type des arbres binaires), mais déplié deux fois, parce que l'utilisation explicite des nœuds des deux premiers niveaux provoque la création d'une variable fraîche pour décrire chacun d'eux. De telles situations sont donc assez fréquentes en pratique, et l'algorithme de minimisation les traite de façon satisfaisante.

Exemple. Cet exemple est similaire, en principe, au précédent, mais est digne d'être mentionné parce qu'il correspond également à une situation assez typique en pratique. Nous supposons ici que F est un opérateur binaire covariant. (Par exemple, si nous disposons de types produits et variantes, nous pouvons choisir $F : (\alpha, \beta) \mapsto$ `[ Nil | Cons of` $\alpha \times \beta$ `]`.) Soit σ le schéma

$$\gamma_1^- \to \gamma_2^- \to \lambda^+ \mid \{\gamma_1^- \leq F(\alpha^-, \gamma_1^-),\, \gamma_2^- \leq F(\alpha^-, \gamma_2^-),\, F(\beta^+, \lambda^+) \leq \lambda^+,\, \alpha^- \leq \beta^+\}$$

Alors, la plus grande partition compatible avec σ est

$$\{\{\alpha\}, \{\beta\}, \{\gamma_1, \gamma_2\}, \{\lambda\}\}$$

donc le schéma de types simplifié $\sigma/_{\equiv_\sigma}$ est

$$\gamma^- \to \gamma^- \to \lambda^+ \mid \{\gamma^- \leq F(\alpha^-, \gamma^-),\, F(\beta^+, \lambda^+) \leq \lambda^+,\, \alpha^- \leq \beta^+\}$$

L'algorithme de minimisation a réussi à identifier γ_1 et γ_2. A l'aide d'un argument similaire à celui exposé dans l'exemple précédent, on peut dire que ces deux variables représentent le point fixe $\mu t.F(\alpha, t)$, et c'est pourquoi elles peuvent être identifiées.

Une telle situation peut se produire fréquemment en pratique. Par exemple, considérons une fonction qui effectue la fusion de deux listes triées. Son type inféré pourrait être σ. L'algorithme de minimisation constate que les deux arguments de la fonction sont utilisés de la même manière, puisqu'ils portent des contraintes similaires, et en déduit qu'on peut leur donner le même type.

Exemple (Elimination des 2-couronnes). Soit σ le schéma

$$\alpha^- \to \beta^- \to \gamma^+ \times \delta^+ \mid \{\alpha^- \leq \gamma^+, \alpha^- \leq \delta^+, \beta^- \leq \gamma^+, \beta^- \leq \delta^+\}$$

Cette situation est parfois appelée 2-*couronne* dans la littérature, parce que le graphe de contraintes a l'aspect suivant :

$$\begin{tikzcd}
\gamma^+ & \delta^+ \\
\alpha^- \arrow[u] \arrow[ur] & \beta^- \arrow[u] \arrow[ul]
\end{tikzcd}$$

De semblables situations sont très courantes en pratique. Le schéma σ est engendré par le code

```
fun x y -> if true then (x, y) else (y, x)
```

Le calcul de $\equiv_\sigma$ donne

$$\{\{\alpha, \beta\}, \{\gamma, \delta\}\}$$

Donc, $\sigma/_{\equiv_\sigma}$ est

$$\alpha^- \to \alpha^- \to \gamma^+ \times \gamma^+ \mid \{\alpha^- \leq \gamma^+\}$$

Si, de plus, on remplace chaque variable par son unique borne, on obtient même

$$\alpha^\pm \to \alpha^\pm \to \alpha^\pm \times \alpha^\pm$$

Mais cela viole l'invariant de mono-polarité, et ne sera donc fait que lors de l'affichage, et non de façon interne (cf. section 15.2).

Cet exemple est intéressant à plusieurs titres. D'abord, on remarque que la méthode bien connue qui consiste à remplacer chaque variable par son unique borne échoue ici, puisque chaque variable a deux bornes. (Bien sûr, il serait possible de prétendre, par exemple, que l'unique borne supérieure de α est $\gamma \sqcap \delta$, et d'effectuer la substitution ; et de même pour β. Cependant, cela n'est pas une simplification ; le résultat est en fait plus complexe que σ.)

Ensuite, le fait que les 2-couronnes (ou même, en fait, les n-couronnes) soient efficacement éliminées est une bonne nouvelle pour l'algorithme de canonisation. Rappelons que la canonisation consiste à éliminer les occurrences de $\sqcap$ et $\sqcup$ introduites par le calcul de clôture. Une phase lui est dédiée, après la phase de clôture. Si l'algorithme découvre l'expression $\gamma \sqcap \delta$, il la remplace par une variable fraîche α, et crée les contraintes $\alpha \leq \gamma$ et $\alpha \leq \delta$. Si, au cours d'une passe de canonisation ultérieure, il découvre la même expression, il créera une nouvelle variable β et ajoutera les contraintes $\beta \leq \gamma$ et $\beta \leq \delta$. Une 2-couronne a donc été créée. Grâce à l'algorithme de minimisation, cela ne pose pas de problème : elle sera efficacement éliminée. Sans cela, des usages répétés de l'algorithme de canonisation pourraient mener à une accumulation de « couronnes ».

Exemple. Notre dernier exemple éclaircit un point de détail de la définition 13.3. La première condition pour qu'une partition soit compatible avec σ est

– $\alpha \leq_C \beta$ implique $\forall \alpha' \equiv \alpha \quad \forall \beta' \equiv \beta \quad \alpha' \leq_C \beta'$.

Or, nous avons parlé, pendant notre introduction informelle, de bisimulation, et la définition d'une bisimulation devrait plutôt avoir l'aspect suivant :

– $\alpha \leq_C \beta$ implique $\forall \alpha' \equiv \alpha \quad \exists \beta' \equiv \beta \quad \alpha' \leq_C \beta'$.

Voici un exemple montrant que cette dernière condition n'est pas correcte. Soit σ le schéma

$$\alpha \to \beta \to \gamma \times \delta \mid \{\alpha \leq \gamma, \beta \leq \delta\}$$

Si on adopte cette définition modifiée de la compatibilité, alors la plus grande partition compatible est

$$\{\{\alpha, \beta\}, \{\gamma, \delta\}\}$$

et le schéma quotient est

$$\alpha \to \alpha \to \gamma \times \gamma \mid \{\alpha \leq \gamma\}$$

Or, ce schéma n'est clairement pas équivalent à σ. Le schéma σ décrit en fait deux flots de données entièrement distincts, et bien qu'ils aient la même forme, il n'est pas correct de les identifier.

La condition en forme de « $\forall\exists$ » n'est donc pas correcte ; c'est bien la condition en forme de « $\forall\forall$ » qui convient. Notons que chez Felleisen et Flanagan [19, 20], c'est une condition « $\forall\exists$ » qui apparaît dans le cas des contraintes entre variables ; mais au contraire, une condition « $\forall\forall$ » est utilisée dans le cas des contraintes mettant en jeu le destructeur dom. Le principe reste donc le même, mais des différences techniques apparaissent, du fait de la différence des formalismes adoptés.

13.5 Complétude

Les exemples ci-dessus montrent que l'algorithme de minimisation est très puissant. Aussi, une question se pose : est-il complet, c'est-à-dire, est-il capable d'effectuer toutes les substitutions correctes ? (Rappelons ce qui a été dit dans la section 13.1, à savoir qu'une substitution correcte est une partition des variables qui respecte les polarités et qui conduit à un schéma quotient équivalent au schéma initial.)

La réponse est négative. Il fallait s'y attendre, puisque nous avons volontairement remplacé le critère de correction par le critère de compatibilité, qui est plus simple et autorise un calcul efficace, mais est aussi plus faible. L'algorithme est donc complet par rapport à sa spécification, mais celle-ci n'autorise pas toutes les substitutions correctes. En voici un exemple.

Exemple. Soit F un opérateur de types covariant, distinct de l'identité. (Par exemple, on prendra $F : \alpha \mapsto \top \to \alpha$.) Soit σ le schéma

$$\alpha^- \to \beta^- \to \gamma^+ \mid \{\alpha^- \leq F\,\alpha^-,\, \beta^- \leq F\,\beta^-,\, F\,\gamma^+ \leq \gamma^+,\, \alpha^- \leq \gamma^+\}$$

Ici, α et β ne peuvent pas être dans la même classe. S'ils l'étaient, alors la présence de la contrainte $\alpha \leq \gamma$ demanderait que $\beta \leq \gamma$ soit également présente, ce qui n'est pas le cas.

Cependant, la contrainte $\alpha \leq \gamma$ est superflue ; c'est-à-dire qu'elle est impliquée par les autres contraintes. (On pourra utiliser l'axiomatisation de l'implication de contraintes, donnée par la figure 8.2 page 83, pour vérifier que $\alpha \leq F\,\alpha$ et $F\,\gamma \leq \gamma$ impliquent $\alpha \leq \gamma$.) En conséquence, σ est équivalent à

$$\alpha^- \to \beta^- \to \gamma^+ \mid \{\alpha^- \leq F\,\alpha^-,\, \beta^- \leq F\,\beta^-,\, F\,\gamma^+ \leq \gamma^+\}$$

Dans ce nouveau schéma de types, α et β peuvent être mis dans la même classe ; il est donc équivalent à

$$\alpha^- \to \alpha^- \to \gamma^+ \mid \{\alpha^- \leq F\,\alpha^-,\, F\,\gamma^+ \leq \gamma^+\}$$

Nous pouvons maintenant rajouter la contrainte $\alpha \leq \gamma$ si nous le souhaitons ; ainsi, nous avons vérifié que σ est équivalent à $[\beta \leftarrow \alpha]\,\sigma$.

Ainsi, toutes les substitutions correctes ne sont pas compatibles. Le problème, dans l'exemple ci-dessus, est que la contrainte $\beta \leq \gamma$ n'est pas explicitement présente dans le graphe, bien qu'elle soit dérivable à l'aide des autres contraintes.

Nous pourrions donner une définition plus puissante de la compatibilité en modifiant la première condition de la définition 13.3. Il suffirait en fait de redéfinir les ensembles de prédécesseurs et de successeurs à l'aide de l'implication de contraintes :

$$\begin{aligned}\mathrm{pred}_C(\alpha) &= \{\beta\,;\, C \Vdash \beta \leq \alpha\}\\ \mathrm{succ}_C(\alpha) &= \{\beta\,;\, C \Vdash \alpha \leq \beta\}\end{aligned}$$

Il est possible qu'avec cette modification, la notion de compatibilité coïncide avec celle de correction. Cependant, cela ne présenterait que peu d'intérêt pratique, pour deux raisons. D'abord, nous ne pourrions calculer qu'une approximation des ensembles de prédécesseurs et de successeurs, puisque nous ne disposons d'aucun algorithme complet de décision de l'implication de contraintes. L'algorithme de minimisation deviendrait donc incomplet par rapport à sa spécification. Ensuite, le coût de l'algorithme d'implication incomplet étant relativement élevé, on subirait probablement une perte de performance importante, pour un gain de puissance marginal.

En pratique, on peut imaginer d'effectuer, à intervalles espacés (par exemple après chaque phrase *toplevel*), une phase d'élimination des contraintes superflues, comme la contrainte $\alpha \leq \gamma$ dans l'exemple précédent. Leur détection se fait à l'aide de l'algorithme d'implication de contraintes. Nous avons implémenté cette phase ; il en ressort un gain de l'ordre de quelques pourcents, en termes de performance comme de résultats.

Troisième partie

Discussion

Chapitre 14

Extensions

On pourrait formuler l'opinion, tout à fait justifiée, que l'ensemble de types bruts proposé, formé uniquement à partir de $\bot$, $\top$ et $\rightarrow$, est d'une remarquable pauvreté. On remarquera même que, dans un modèle aussi restreint, tout graphe de contraintes admet une solution : la substitution qui à toute variable associe l'arbre régulier $\mu t.t \rightarrow t$. (Cette propriété va de pair avec le fait, mentionné dans la démonstration de la proposition 5.10, qu'aucune erreur d'exécution ne peut se produire en λ-calcul pur.)

Plusieurs de nos énoncés admettent donc des démonstrations triviales, si l'on tire parti du fait que $\rightarrow$ est le seul constructeur de type en dehors de $\bot$ et $\top$. Cependant, nos preuves n'utilisent pas cette propriété, et sont donc aisément extensibles à des systèmes de types beaucoup plus riches. Nous aurions pu choisir un tel système et le traiter de façon explicite, ou encore tenter de paramétrer l'ensemble de notre théorie par un treillis de types quelconque, vérifiant quelques propriétés adéquates. Cependant, il nous semble que cela aurait augmenté significativement, sinon la complexité intrinsèque des preuves, du moins leur complexité apparente. Nous avons donc préféré exposer notre théorie dans le cadre le plus simple possible.

Par ailleurs, d'autres extensions sont aisément concevables, que nous qualifions *d'orthogonales*, parce leur interaction avec le sous-typage est faible ou nulle. Ces extensions sont largement indépendantes de notre théorie, et c'est pourquoi nous ne les avons pas mentionnées jusqu'ici. Elles augmentent cependant considérablement la puissance de notre système, et méritent donc d'être étudiées ici.

Au cours de ce chapitre, nous esquissons d'abord, de façon assez formelle, ce qu'aurait pu être la paramétrisation de notre théorie (section 14.1). Dans les sections suivantes, nous détaillons quelques cas particuliers de celle-ci, qui correspondent à des notions classiques : types de base (section 14.2), constructeurs de types n-aires isolés (section 14.3), puis types enregistrements et variantes polymorphes (section 14.4). Ensuite, nous présentons deux extensions orthogonales : l'ajout de variables de rangée (section 14.5) et d'une analyse des exceptions (section 14.6). L'expressivité des variables de rangée peut sembler, à première vue, rendre superflu l'utilisation même du sous-typage ; nous approfondissons ce point au cours de la section 14.7.

14.1 Paramétrisation de la théorie

Comme nous l'avons mentionné ci-dessus, nous aurions pu paramétrer notre théorie par le choix du langage de types bruts. Pour ce faire, nous aurions défini le treillis des types bruts de façon plus abstraite ; au lieu de le spécifier entièrement, nous aurions seulement requis une série de conditions suffisantes pour échafauder notre théorie. Nous avons préféré éviter cette approche afin de ne pas masquer les idées fondamentales de ce travail derrière une couche d'abstraction supplémentaire. Cependant, il est intéressant de savoir quelles seraient ces conditions suffisantes, afin d'évaluer la généralité de notre approche. Cette section définit donc une notion de *signature brute*, qui spécifie partiellement le treillis des types bruts. Elle reprend ensuite les résultats du chapitre 1 en les paramétrant par une signature brute quelconque.

Définition 14.1 *Soit $\mathcal{L}$ un ensemble dénombrable d'*étiquettes. *On appelle* arité *un ensemble fini d'étiquettes. Une* signature brute Σ_b *consiste en :*

- *un ensemble de* constructeurs c*, chacun étant muni d'une arité* $a(c)$ *;*
- *une fonction de* variance v*, qui à toute étiquette associe un élément de* $\{-,+\}$ *;*
- *une relation d'ordre* $\leq_b$ *sur les constructeurs, telle que*
  - *l'ensemble des constructeurs, muni de cet ordre, forme un treillis ;*
  - *si* $c_1 \leq_b c_2 \leq_b c_3$*, et si* $l \in a(c_1) \cap a(c_3)$*, alors* $l \in a(c_2)$ *(on pourrait appeler cette condition « convexité de l'arité ») ;*
  - *les constructeurs* $\perp$ *et* $\top$ *sont constants, c'est-à-dire d'arité* $\varnothing$*.*

On définit alors l'ensemble des arbres bruts, en fonction d'une signature Σ_b, comme suit.

Définition 14.2 *Un* chemin *est un élément de* $\mathcal{L}^*$*. La* parité *d'un chemin est le nombre d'éléments de variance négative qu'il contient, modulo 2. Un* arbre brut τ *est une fonction partielle des chemins vers* Σ_b*, de domaine non vide et clos par préfixe, telle que pour tout* $l \in \mathcal{L}$*,* $\tau(pl)$ *est défini ssi* $l \in a(\tau(p))$*.*

La définition du sous-typage est inchangée (cf. définition 1.4). Il est facile de vérifier qu'il s'agit d'un ordre ; la réflexivité est immédiate, l'antisymétrie est démontrée comme dans la proposition 1.4, et la transitivité s'obtient aisément, en utilisant la propriété de convexité de l'arité définie plus haut.

Notons que cette propriété de convexité est nécessaire. Supposons, en effet, qu'elle soit contredite pour c_1, c_2, c_3 et l donnés. Alors nous avons $c_1(\top) \leq c_2$ et $c_2 \leq c_3(\perp)$, mais pas $c_1(\top) \leq c_3(\perp)$, car cela entraînerait $\top \leq \perp$. (Cette assertion pourrait être vraie, mais alors le treillis serait réduit à un point ; donc c_1, c_2 et c_3 seraient identiques, et la propriété ne pourrait pas être contredite.) La relation $\leq$ n'est donc pas transitive. (Nous avons commis ici un léger abus de langage, puisque nous avons utilisé les constructeurs c_1, c_2 et c_3 sans spécifier quelle valeur nous associions aux étiquettes autres que l ; n'importe quelle valeur fixée convient.)

On vérifie ensuite que la relation de sous-typage définit un treillis. La définition des opérations $\sqcup$ et $\sqcap$ est similaire à celle donnée dans la démonstration de la proposition 1.4, et la démonstration elle-même est inchangée. $\sqcup$ et $\sqcap$ sont caractérisés par les équations suivantes :

$$\begin{array}{rl} & (\tau_1 \sqcup \tau_2)(\epsilon) = \tau_1(\epsilon) \sqcup_b \tau_2(\epsilon) \\ \forall l \in a((\tau_1 \sqcup \tau_2)(\epsilon)) & (\tau_1 \sqcup \tau_2)(l) = \tau_1(l) \sqcup^{v(l)} \tau_2(l) \end{array}$$

(Ces deux équations concernent $\sqcup$; il faut les compléter par deux équations symétriques concernant $\sqcap$.) Dans la seconde équation ci-dessus, $\sqcup^+$ signifie $\sqcup$ et $\sqcup^-$ signifie $\sqcap$. Par ailleurs, $\tau_1(l)$ et $\tau_2(l)$ peuvent ne pas être définis ; il faut alors les lire comme l'élément neutre de l'opération $\sqcup^{v(l)}$.

Ces équations expriment que les opérations $\sqcup$ et $\sqcap$ sont distributives. Elles se calculent en effectuant d'abord une opération élémentaire (définie par la signature brute) sur les constructeurs de tête, puis en distribuant les opérations (modulo la variance) sur les sous-termes.

L'ensemble de la théorie peut être reconstitué sur ces nouvelles bases. (Nous n'avons pas indiqué comment étendre les règles de typage, donc le chapitre 5 n'est pas concerné ; mais la majeure partie de la théorie est purement située dans l'univers des types et des schémas de types, donc peut être étendue.) Si nous ne démontrons pas cette affirmation, c'est uniquement parce que nous avons pris le parti de la simplicité tout au long de notre exposé théorique. Elle est de toute façon soutenue par l'expérience pratique, puisque notre prototype implémente un langage de types très riche, qui utilise largement les possibilités offertes par la définition 14.1.

A ce propos, on remarquera que, si l'ensemble de la théorie peut être paramétrée par une signature brute, il en va de même de l'implémentation. (A l'exception toutefois du générateur de contraintes, puisque celui-ci implémente les règles d'inférence de types, qui font explicitement référence à une signature connue.) Les algorithmes de manipulation et de simplification de contraintes peuvent aisément être écrits sous forme de foncteurs paramétrés par cette signature. Il en résulte une simplification appréciable du code, puisque les différentes variétés de types (types fonctions, types produits, types enregistrements, etc.) sont alors traitées de façon uniforme. Cependant, cela se fait au prix d'une perte de performance, puisqu'on ne peut plus optimiser les divers cas particuliers. En pratique, nous avons effectué cette uniformisation ; nous avons mesuré une pénalité d'environ 10% en temps d'exécution. (Les types enregistrements et variantes restent toutefois traités séparément, à cause de la présence des variables de rangée – cf. section 14.5.) Cette pénalité est acceptable, étant donné le gain de lisibilité et de généralité du code qui en résulte.

Enfin, nous n'avons pas encore expliqué pourquoi nous avons exigé que les constructeurs $\bot$ et $\top$ soient constants. Si ce n'était pas le cas, alors le plus petit, ou le plus grand, élément du treillis des types serait un terme infini. (Par exemple, si $\top$ est unaire, le plus grand type brut sera $\mu t.\top(t)$.) Cela poserait des problèmes relativement inextricables, du point de vue théorique comme du point de vue pratique, parce que les types bruts infinis ne sont pas exprimables dans notre langage des types. Cette condition ne semblant pas fort restrictive, nous l'adoptons donc.

14.2 Types de base

La plupart des langages de programmation fournissent des types de données élémentaires comme les entiers, les réels, les booléens, etc. L'ajout de ces types de base est une application élémentaire du mécanisme d'extension décrit précédemment. Dans le cas le plus simple, nous pouvons définir des types de base comme des constructeurs constants isolés :

`bool`   `unit`

Cependant, nous pouvons également définir des liens de sous-typage entre types de base, comme le montrent ces exemples classiques :

$$\begin{tikzcd} \texttt{float} & \texttt{string} \\ \texttt{int} \arrow[u] & \texttt{char} \arrow[u] \end{tikzcd}$$

De façon générale, il est possible de définir un ordre partiel arbitrairement complexe sur les types de base, pourvu que la signature brute Σ_b reste un treillis. Par exemple, on pourrait imaginer la situation suivante :

$$\begin{tikzcd} & \texttt{rat} & \\ \texttt{int} \arrow[ur] & & \texttt{posrat} \arrow[ul] \\ & \texttt{posint} \arrow[ul] \arrow[ur] & \end{tikzcd}$$

Dans cet exemple un peu inhabituel, `posint` représente les entiers positifs, `int` les entiers, `posrat` les rationnels positifs et `rat` les rationnels.

14.3 Constructeurs n-aires isolés

Une deuxième possibilité d'extension, toujours dans le cadre donné par la section 14.1, consiste à ajouter au langage un constructeur n-aire isolé. Un tel constructeur est dit isolé car incompatible, du point de vue du sous-typage, avec tout autre constructeur (à l'exception, bien sûr, de $\bot$ et $\top$). Le constructeur des paires $*$, et le constructeur $\rightarrow$ lui-même, sont des exemples de constructeurs binaires isolés.

Nous sommes limités, dans la définition d'un tel constructeur c, par une condition provenant de la définition 14.1 : chacun de ses n arguments doit être soit *covariant*, soit *contravariant* ; il ne doit pas être *invariant*. Rappelons rapidement le sens de ces termes.

Formellement parlant, une étiquette $l \in \mathcal{L}$ est covariante si la fonction d'arité $a(c)$ lui associe le signe $+$, et contravariante si elle lui associe le signe $-$. De façon plus imagée, un argument est contravariant s'il inverse la relation de sous-typage lors de la décomposition structurelle, et covariant dans le cas contraire. Par exemple, le premier argument du constructeur $\rightarrow$ est contravariant, parce que $\tau_1 \rightarrow \tau_2 \leq \tau'_1 \rightarrow \tau'_2$ implique $\tau'_1 \leq \tau_1$, et non $\tau_1 \leq \tau'_1$; le second est covariant.

Un argument est dit invariant s'il est à la fois covariant et contravariant, c'est-à-dire si une inéquation entre types construits implique l'égalité entre fils. Remarquons que la définition 14.1 n'autorise pas les arguments invariants. En effet, le phénomène d'invariance pose problème, pour plusieurs raisons. D'abord, les opérations du treillis ne sont pas distributives sur un argument invariant. Cela nous empêche de repousser les constructeurs $\sqcup$ et $\sqcap$ jusqu'aux variables, et donc d'effectuer la canonisation. Par ailleurs, il devient impossible de faire respecter l'invariant de mono-polarité, puisque si une variable apparaît en position d'argument invariant, alors elle sera soit neutre, soit bipolaire.

A titre d'exemple, considérons le cas du type référence. Plusieurs langages fonctionnels proposent un constructeur de type unaire appelé `ref` : τ `ref` est le type des cellules dont le contenu est de type τ. Puisque la lecture et l'écriture sont autorisées, $\tau\ \texttt{ref} \leq \tau'\ \texttt{ref}$ implique nécessairement $\tau = \tau'$, de façon à respecter la sûreté du langage vis-à-vis de la sémantique. Par conséquent, $(\tau\ \texttt{ref}) \sqcup (\tau'\ \texttt{ref})$ est égal à $(\tau\ \texttt{ref})$ si $\tau = \tau'$, et à $\top$ dans le cas contraire. Ainsi, $\sqcup$ n'est pas distributive vis-à-vis de `ref`.

Pour éviter ce problème, nous sommes amenés à faire de `ref` un constructeur binaire : (τ_0, τ_1) `ref` sera le type des cellules dans lesquelles on peut écrire des données de type τ_0 et en provenance desquelles on peut lire des données de type τ_1. (Cette idée nous a été suggérée par Luca Cardelli ; elle a été découverte indépendamment par Smith et Trifonov.) Avec cette convention, il est correct de poser que $(\tau_0, \tau_1)\ \texttt{ref} \leq (\tau'_0, \tau'_1)\ \texttt{ref}$ est équivalent à $\tau'_0 \leq \tau_0 \wedge \tau_1 \leq \tau'_1$. Le constructeur `ref` est donc contravariant en ce qui concerne son premier argument, et covariant en ce qui concerne le second. Ainsi, il se comporte exactement comme le constructeur $\rightarrow$. Les trois primitives utilisées pour créer une cellule, en lire le contenu et le modifier admettent les schémas de types suivants :

$$\begin{aligned} \texttt{ref}\ &: \alpha \rightarrow (\alpha, \alpha)\ \texttt{ref} \\ \texttt{!}\ &: (\alpha, \beta)\ \texttt{ref} \rightarrow \beta \\ \texttt{:=}\ &: (\alpha, \beta)\ \texttt{ref} \rightarrow \alpha \rightarrow \texttt{unit} \end{aligned}$$

Historiquement, l'idée de rendre binaire le constructeur `ref` est donc apparue d'abord pour éviter un problème technique. Cependant, on peut remarquer qu'elle est en accord avec la philosophie du sous-typage. En effet, celui-ci permet essentiellement de garder trace de la direction du flot de données, tandis se baser sur l'égalité mène à un graphe de flot non orienté, et donc plus grossier. Utiliser un constructeur `ref` à un seul argument revient à confondre lecture et écriture, donc implique nécessairement une perte de finesse. Considérons par exemple le programme

```
(fun x -> x := Non; !x) (ref Oui)
```

(`Oui` et `Non` sont des constructeurs de données ; ils seront introduits, avec les types variantes polymorphes, dans la section 14.4.) Si on choisit un constructeur `ref` unaire, alors ce programme est mal typé, même en présence de sous-typage. En effet, `ref Oui` reçoit le type `[ Oui ] ref`, et l'instruction `x := Non` engendre la contrainte insoluble `[ Non ]` $\leq$ `[ Oui ]`. Au contraire, si on utilise un constructeur `ref` binaire, alors lecture et écriture sont correctement différenciées, et le type de l'expression est `[ Oui ]` $\sqcup$ `[ Non ]`, c'est-à-dire `[ Oui | Non ]`. En conclusion, on peut arguer que l'utilisation d'un constructeur `ref` binaire offre plus de finesse, et se justifie de façon naturelle dans le cadre du sous-typage.

14.4 Enregistrements et variantes polymorphes

Les extensions mentionnées jusqu'ici ne permettent pas un usage très poussé du sous-typage. La présence du type $\bot$ permet de prouver, grâce au typage, que certaines expressions ne terminent pas, et de les utiliser alors dans n'importe quel contexte. La présence du type $\top$ permet de manipuler des objets de type inconnu. D'un point de vue pratique, cela reste anecdotique. Quant au sous-typage entre types de base, il peut être utile, mais ne justifie pas en soi le développement d'une théorie aussi puissante. Les choses deviennent plus intéressantes lorsque l'on ajoute des enregistrements (*records*) ou variantes (également appelées sommes) polymorphes. Cette extension est, elle aussi, un cas particulier de la théorie esquissée par la section 14.1.

On suppose donnée une famille dénombrable d'étiquettes $l \in \mathcal{L}$. Un type enregistrement est de la forme

$$\texttt{\{ } l\texttt{: } \tau_l \texttt{ \}}_{l \in L}$$

où L est un sous-ensemble fini de $\mathcal{L}$. La relation de sous-typage entre types enregistrements est définie par

$$\texttt{\{ } l\texttt{: } \tau_l \texttt{ \}}_{l \in L} \leq \texttt{\{ } l\texttt{: } \tau'_l \texttt{ \}}_{l \in L'} \iff \forall l \in L' \quad (l \in L \wedge \tau_l \leq \tau'_l)$$

Ainsi, si nous masquons certains des champs d'un type enregistrement donné, nous en obtenons un supertype. En d'autres termes, le sous-typage permet d'oublier la présence de certains champs ; dans un contexte ou un certain type enregistrement est attendu, on peut fournir un enregistrement contenant plus d'information que nécessaire. C'est la raison pour laquelle ces types sont dits *polymorphes* : une valeur enregistrement donnée admet de nombreux types différents, selon que chaque champ est exposé ou caché.

Par exemple, on a

$$\texttt{\{ a: } \alpha\texttt{; b: } \beta \texttt{ \}} \leq \texttt{\{ a: } \alpha \texttt{ \}}$$

Grâce à cela, l'expression suivante est valide, bien que l'argument fourni ait plus de champs que la fonction n'en exige :

```
(fun x -> x.a) { a = 0; b = true }
```

Les enregistrements polymorphes peuvent servir de base à divers codages de la programmation avec objets dans un langage fonctionnel. L'envoi d'un message donné à un objet nécessitera typiquement l'accès à un champ donné dans un enregistrement. Cette dernière opération pouvant s'appliquer à des enregistrements contenant un nombre arbitraire d'autres champs, un message défini par une certaine classe pourra être envoyé aux objets appartenant à ses sous-classes.

Cette extension est une application de la paramétrisation exposée au début de ce chapitre. Les constructeurs introduits sont les $\texttt{\{\}}_L$, où l'ensemble L varie parmi les parties finies de $\mathcal{L}$. Le constructeur $\texttt{\{\}}_L$ a pour arité L, et toutes les étiquettes $l \in \mathcal{L}$ sont covariantes. L'ordre sur les constructeurs est l'inverse de l'inclusion entre arités : $\texttt{\{\}}_L \leq_b \texttt{\{\}}_M$ est vrai si et seulement si $L \supset M$. Il s'agit donc bien d'un treillis. Enfin, la condition de convexité de l'arité est satisfaite ; en fait, l'arité est ici une fonction décroissante.

Les types variantes polymorphes sont entièrement symétriques des types enregistrements. Ils sont de la forme

$$\texttt{[} l\texttt{: } \tau_l \texttt{]}_{l \in L}$$

La relation de sous-typage entre types variantes est donnée par

$$\texttt{[} l\texttt{: } \tau_l \texttt{]}_{l\in L} \leq \texttt{[} l\texttt{: } \tau'_l \texttt{]}_{l\in L'} \iff \forall l \in L \quad (l \in L' \wedge \tau_l \leq \tau'_l)$$

Ceci exprime que, par ajout de nouveaux cas un type variante donné, nous en obtenons un supertype. Ainsi, une valeur construite pourra être passée à une fonction traitant plusieurs cas. Par exemple,

$$\texttt{[None: unit]} \leq \texttt{[None: unit | Some: } \alpha \texttt{]}$$

Cela rend valide l'expression suivante, bien que le domaine de la fonction mentionne plusieurs cas et que le type de son argument effectif n'en spécifie qu'un seul :

```
(fun f None     -> ()
   | f (Some x) -> f x) print_int (Some 3)
```

On notera que tout ceci se fait sans aucune déclaration de type de la part de l'utilisateur. Une même étiquette peut ainsi apparaître dans plusieurs types enregistrements différents, ou un même constructeur de données dans plusieurs types variantes différents. Cela offre une grande flexibilité au langage.

14.5 Enregistrements et variantes extensibles

Les types enregistrements polymorphes, tels que nous les avons présentés dans la section 14.4, sont d'une puissance relativement limitée. Ils permettent d'ignorer la présence de certains champs dans un enregistrement, mais pas de faire référence à ces champs inconnus, ne serait-ce que pour les recopier en bloc.

De ce fait, il est impossible d'attribuer un type satisfaisant aux fonctions d'extension. (La fonction d'extension sur le champ l accepte un enregistrement r et une valeur v, et renvoie un enregistrement identique à r, sauf que le champ l a été créé si nécessaire, et a pris la valeur v.) Il est impossible d'exprimer le fait que les types des champs autres que l sont inchangés. Si on se restreint au cas particulier où on suppose que le champ l existe déjà et avec le même type, on pourrait naïvement attribuer à la fonction d'extension le schéma de types

$$\alpha \to \beta \to \alpha \mid \{ \alpha \leq \texttt{\{ } l\texttt{: } \beta \texttt{ \}} \}$$

Cependant, celui-ci est incorrect ! En effet, β étant négatif, il peut être remplacé par sa borne supérieure sans modifier la dénotation du schéma :

$$\alpha \to \top \to \alpha \mid \{ \alpha \leq \texttt{\{ } l\texttt{: } \top \texttt{ \}} \}$$

Ici, le deuxième argument de la fonction peut prendre n'importe quel type, et il n'est donc pas correct de renvoyer α en résultat. Le problème vient du fait que deux valeurs quelconques ont toujours un type commun, à savoir $\top$. Par conséquent, exiger que deux quantités soient « du même type » n'a aucun effet.

14.5.1 Description

Pour résoudre ce problème, Rémy [45] propose l'introduction de variables de rangée. Nous ne les décrirons ici que de façon informelle, puisque c'est là une théorie entièrement indépendante, qui n'interagit que très simplement avec le sous-typage. Elle est exposée de façon précise dans [45]. Ce n'est donc pas là une nouveauté, et la notion de variable de rangée ne dépend en rien du sous-typage ; mais il est intéressant de constater que ces notions s'intègrent sans difficulté.

On donne d'abord une définition plus élaborée des types enregistrements, laquelle fait intervenir des *termes de rangée* θ :

$$\begin{aligned}\tau &::= \ldots \mid \texttt{\{ } l_1 \texttt{: } \theta_1 \texttt{; } \ldots \texttt{; } l_n \texttt{: } \theta_n \texttt{; } \theta \texttt{ \}} \\ \theta &::= \rho \mid \texttt{Abs} \mid \texttt{Pre } \tau\end{aligned}$$

Un terme de rangée fournit une information sur un champ. Il peut valoir `Abs`, pour indiquer l'absence de ce champ, ou `Pre` τ pour indiquer que le champ est présent et contient une valeur de type τ. Il peut également s'agir d'une *variable de rangée* ρ, dans le cas où cette information est inconnue. Dans un type enregistrement, on fait correspondre à chaque champ un terme de rangée. De plus, en dernière position, on trouve un terme de rangée sans étiquette, qui représente l'ensemble (infini) de tous les champs non mentionnés explicitement.

Tout type enregistrement a donc, en quelque sorte, une infinité de champs, dont seuls un nombre fini sont explicitement nommés. La relation de sous-typage sur les types enregistrements se définit alors champ par champ. Pour permettre d'ignorer certains champs, on pose `Pre` $\tau \leq$ `Abs` pour tout τ. Par ailleurs, le constructeur `Pre` est bien sûr covariant. L'assertion

$$\texttt{\{ a: Pre } \alpha\texttt{; b: Pre } \beta\texttt{; } \rho_1 \texttt{ \}} \leq \texttt{\{ a: Abs; b: Pre } \gamma\texttt{; } \rho_2 \texttt{ \}}$$

est donc équivalente à $\beta \leq \gamma \wedge \rho_1 \leq \rho_2$. On notera que les variables de rangée peuvent, comme les variables de types habituelles, recevoir des contraintes.

Il est alors possible d'attribuer à la fonction d'extension sur le champ l un schéma de types satisfaisant, à savoir

$$\texttt{\{ } l\texttt{: Abs; } \rho \texttt{ \}} \rightarrow \alpha \rightarrow \texttt{\{ } l\texttt{: Pre } \alpha\texttt{; } \rho \texttt{ \}}$$

Ce type indique que le champ l peut être absent de l'enregistrement initial r. Grâce au mécanisme de sous-typage, le champ l peut *a fortiori* être présent, avec n'importe quel type. Dans l'enregistrement renvoyé par la fonction, le champ l apparaît avec le type α, qui est le type de la nouvelle valeur fournie, et tous les autres champs, représentés par la variable de rangée ρ, sont inchangés.

Le trait le plus délicat des variables de rangée est *l'expansion à la demande*. Imaginons, par exemple, qu'au cours du processus d'inférence, apparaisse une contrainte de la forme

$$\texttt{\{ a: Pre } \alpha\texttt{; b: Pre } \beta\texttt{; } \rho \texttt{ \}} \leq \texttt{\{ a: Pre } \alpha'\texttt{; } \rho' \texttt{ \}}$$

L'algorithme de clôture doit décomposer cette contrainte. En ce qui concerne le champ `a`, on obtient `Pre` $\alpha \leq$ `Pre` α', puis $\alpha \leq \alpha'$. Mais que se passe-t-il pour le champ `b` ? Dans le type enregistrement de droite, l'information concernant ce champ est comprise dans la variable ρ', qui décrit tous les champs non nommés explicitement. Pour « extraire » cette information, on *expanse* la variable ρ', c'est-à-dire qu'on remplace toutes les occurrences de ρ' par $\langle \texttt{b: } \rho'_1\texttt{; } \rho'_2 \rangle$, où ρ'_1 et ρ'_2 sont des variables fraîches. La contrainte ci-dessus devient donc

$$\texttt{\{ a: Pre } \alpha\texttt{; b: Pre } \beta\texttt{; } \rho \texttt{ \}} \leq \texttt{\{ a: Pre } \alpha'\texttt{; b: } \rho'_1\texttt{; } \rho'_2 \texttt{ \}}$$

qui peut être décomposée en $\alpha \leq \alpha' \wedge$ `Pre` $\beta \leq \rho'_1 \wedge \rho \leq \rho'_2$. On notera que la variable ρ' pouvait apparaître non seulement dans des types enregistrements, comme ci-dessus, mais aussi dans des contraintes, par exemple $\rho' \leq \rho''$. L'expansion de ρ' provoque alors celle de ρ'', dont toutes les occurrences seront remplacées par $\langle \texttt{b: } \rho''_1\texttt{; } \rho''_2 \rangle$. La contrainte $\rho' \leq \rho''$ peut alors être décomposée, et remplacée par $\rho'_1 \leq \rho''_1 \wedge \rho'_2 \leq \rho''_2$. Le mécanisme d'expansion des variables de rangée doit donc être intégré à l'algorithme de clôture, ce qui ne pose pas de problème pratique.

Il est possible de construire des termes mal formés, comme $\texttt{\{ a: Pre } \alpha\texttt{; } \rho \texttt{ \}} \rightarrow \texttt{\{ } \rho \texttt{ \}}$, où le type enregistrement de droite ne contient aucune information sur le champ `a`. Pour les interdire, on met en place un système de *sortes* très simple. Les types fournis par l'utilisateur sont rejetés s'ils sont mal sortés. Quant à ceux construits par les règles de typage, ils sont nécessairement bien sortés. Notons que les simplifications que nous avons décrites préservent naturellement cette propriété, sauf l'opération de minimisation ; il faut donc explicitement exiger que deux variables soient de la même sorte pour pouvoir être identifiées.

14.5.2 Variables de rangée et types variantes

Nous avons décrit l'utilisation des variables de rangée dans les types enregistrements. On les introduit également, de façon symétrique, dans les types variantes. Les termes de rangée sont formés des variables de rangée et de deux constructeurs que nous noterons également `Pre` et `Abs`, par abus de langage. Ils se comportent de façon similaire à leurs homologues des types enregistrements, mais la relation de sous-typage est $\texttt{Abs} \leq \texttt{Pre}\ \tau$. Cela autorise l'ajout de nouveaux cas dans un type variante :

$$\texttt{[Nil: Pre unit; Abs]} \leq \texttt{[Nil: Pre unit; Cons: Pre } \alpha \times \beta\texttt{; Abs]}$$

Le dual, dans les variantes, de la fonction d'extension des enregistrements est la fonction de filtrage élémentaire sur un constructeur `A`, nommée $m_{\texttt{A}}$, dont la sémantique est donnée, de façon informelle, par le code suivant :

```
fun oui non v -> match v with
  A x -> oui x
| autre -> non autre
```

On peut lui attribuer le type

$$(\alpha \to \beta) \to (\texttt{[A: Abs; } \rho \texttt{]} \to \beta) \to \texttt{[A: Pre } \alpha\texttt{; } \rho \texttt{]} \to \beta$$

qui n'est pas sans rappeler le type de la fonction d'extension dans les enregistrements. La fonction `non` est appelée si la valeur `v` n'est pas construite à l'aide du constructeur `A`. Il n'est donc pas nécessaire qu'elle soit capable de traiter ce cas, et on lui demande seulement d'avoir le type $\texttt{[A: Abs; } \rho \texttt{]} \to \beta$. Les cas qu'elle est capable de traiter sont représentés par la variable de rangée ρ, variable que l'on retrouve ensuite dans le type autorisé pour `v`. Ainsi, de même que les variables de rangée permettent de typer l'extension des enregistrements, elles permettent de typer l'extension des filtrages.

Donnons quelques indications sur la façon dont les filtrages sont typés. Les fonctions de filtrage élémentaires sont primitives, c'est-à-dire qu'elles sont ajoutées au langage en tant que constantes. Le type de chacune de ces constantes est fourni par une δ-règle, c'est-à-dire que la constante m_K, qui représente la fonction de filtrage élémentaire sur le constructeur K, admettra par définition le schéma de types

$$(\alpha \to \beta) \to (\texttt{[} K\texttt{: Abs; } \rho \texttt{]} \to \beta) \to \texttt{[} K\texttt{: Pre } \alpha\texttt{; } \rho \texttt{]} \to \beta$$

On se donne également une primitive `reject`, de type

$$\texttt{[Abs]} \to \bot$$

Ensuite, tout filtrage simple, c'est-à-dire correspondant à un unique branchement n-aire, se compile aisément à l'aide de ces primitives. Par exemple, l'expression

```
fun (A x) -> x
```

peut être codée par

$$m_{\texttt{A}}\,(\lambda x.x)\,\texttt{reject}$$

qui a bien le type attendu, à savoir

$$\texttt{[A: Pre } \alpha\texttt{; Abs]} \to \alpha$$

La primitive `reject` est une fonction qui n'accepte aucun argument ; elle sert à « fermer » le filtrage, et c'est de son type que provient le terme de rangée `Abs` dans le type ci-dessus. Il n'est pas nécessaire de l'utiliser lorsque le filtrage se termine par une clause irréfutable, comme ceci :

```
fun (A x) -> x
  | (B y) -> y
  | z -> 0
```

Ce filtrage se code en effet par

$$m_{\mathtt{A}}\,(\lambda x.x)\,(m_{\mathtt{B}}\,(\lambda y.y)\,(\lambda z.z))$$

qui a le type

$$\texttt{[A: Pre } \alpha\texttt{; B: Pre } \alpha\texttt{; Pre } \top \texttt{]} \rightarrow \alpha \mid \{\texttt{int} \leq \alpha\}$$

Les filtrages complexes, c'est-à-dire ceux où des constructeurs peuvent apparaître à une profondeur arbitraire, peuvent également être compilés à l'aide de ces constantes élémentaires. Pour typer de tels filtrages, on commence donc par les compiler, puis on type l'expression obtenue. L'avantage de cette méthode est évidemment sa simplicité : la théorie est inchangée, et il est clair que le type obtenu est correct. En particulier, la notion de filtrage non exhaustif disparaît ; on a la garantie que toute fonction est réellement capable de traiter tous les arguments spécifiés par son type. (On comparera cela au cas de ML, où une analyse séparée est nécessaire.) L'inconvénient de cette méthode est le fait que le type obtenu dépend de la façon dont les filtrages sont compilés, ce qui le rend *a priori* imprévisible. Par exemple, dans notre implémentation, la fonction

```
fun (A, _) -> 0
  | (_, B) -> 0
```

reçoit le type

$$\texttt{[A: Pre unit; Pre } \top \texttt{]} \times \texttt{[B: Pre unit; Abs]} \rightarrow \texttt{int}$$

On constate que ce type exige que la deuxième composante de l'argument soit `B`, diminuant ainsi la puissance de la fonction et brisant la symétrie. On peut cependant arguer que ce problème est inévitable, puisqu'une description précise de cette fonction demanderait de conserver la corrélation entre les deux composantes, c'est-à-dire d'exprimer le fait que soit la première vaut `A`, soit la seconde vaut `B`, ce qui dépasse les possibilités de notre système de types. Si on écrit cette même fonction de façon plus explicite et plus restrictive, par exemple

```
fun (A, (A|B)) -> 0
  | ((A|B), B) -> 0
```

alors on obtient bien le type attendu $\tau \times \tau \rightarrow \texttt{int}$, où τ est égal à

$$\texttt{[A: Pre unit; B: Pre unit; Abs]}$$

Cette façon de typer les filtrages complexes fonctionne, en pratique, dans de nombreux cas. Cependant, elle n'est pas totalement satisfaisante, puisqu'elle reflète la façon dont les filtrages sont compilés, ce qui la rend dissymétrique et imprévisible. Il serait intéressant de donner des règles de typage explicites pour ces filtrages, plutôt que de passer par une phase de compilation implicite, mais cela semble très délicat.

14.6 Analyse des exceptions

En ML, l'évaluation d'une expression peut prendre fin de deux façons : soit celle-ci renvoie un résultat, soit elle lève une *exception*. D'un point de vue logique, résultats et exceptions peuvent être présentés sur un pied d'égalité : il s'agit simplement de deux façons distinctes pour une fonction de transmettre une valeur à son appeleur. Cependant, dans le système de typage de ML, ils sont traités de façon bien différente : toute la puissance du système est mise au service de l'analyse des types des résultats, tandis que les exceptions reçoivent toutes le type `exc` et sont traitées de façon purement monomorphe. Or, pour peu que l'on dispose d'un système de types suffisamment expressif, il est possible de rétablir la symétrie, et d'analyser résultats et exceptions avec la même précision.

14.6.1 Description

Rappelons d'abord le fonctionnement des exceptions. Le langage des expressions est étendu par deux nouvelles constructions :

$$e ::= \ldots \mid \texttt{raise} \mid \texttt{try}\ e\ \texttt{with}\ e$$

La constante `raise` permet de lancer une exception ; on l'utilise comme une fonction à un argument, qui est la valeur de l'exception à lancer. La construction $\texttt{try}\ e_1\ \texttt{with}\ e_2$ évalue l'expression e_1 ; si une exception en résulte, alors celle-ci est passée en argument au gestionnaire e_2. Cette construction est plus concise, et plus puissante, que la notation ML. L'expression ML

$$\texttt{try}\ e_1\ \texttt{with A}\ x\ \texttt{->}\ e_2$$

s'écrirait ici

$$\texttt{try}\ e_1\ \texttt{with}\ m_{\texttt{A}}\,(\lambda x.e_2)\,(\lambda e.\texttt{raise}\ e)$$

La sémantique opérationnelle s'étend également de façon simple pour rendre compte du fonctionnement des exceptions. Plutôt que de détailler les modifications à apporter à la sémantique, nous préférons considérer le codage suivant :

$$\begin{aligned}
[\![x]\!] &= \texttt{Val}\ x \\
[\![\lambda x.a]\!] &= \texttt{Val}\ (\lambda x.[\![a]\!]) \\
[\![a_1\ a_2]\!] &= m_{\texttt{Val}}\,(\lambda v_1.m_{\texttt{Val}}\,(\lambda v_2.v_1\ v_2)\,(\lambda e.e)\,[\![a_2]\!])\,(\lambda e.e)\,[\![a_1]\!] \\
[\![X]\!] &= \texttt{Val}\ X \\
[\![\texttt{let}\,X = a_1\ \texttt{in}\ a_2]\!] &= m_{\texttt{Val}}\,(\lambda v_1.\texttt{let}\,X = v_1\ \texttt{in}\ [\![a_2]\!])\,(\lambda e.e)\,[\![a_1]\!] \\
[\![\texttt{raise}]\!] &= \texttt{Val}\ (\lambda x.\texttt{Exc}\ x) \\
[\![\texttt{try}\ a_1\ \texttt{with}\ a_2]\!] &= m_{\texttt{Exc}}\,(\lambda e_1.m_{\texttt{Val}}\,(\lambda v_2.v_2\ e_1)\,(\lambda e.e)\,[\![a_2]\!])\,(\lambda v.v)\,[\![a_1]\!]
\end{aligned}$$

Ce codage, défini par induction sur la structure des expressions, élimine les constructions nouvellement introduites `raise` et $\texttt{try}\ e_1\ \texttt{with}\ e_2$. L'expression $[\![a]\!]$ peut donc être évaluée à l'aide de la sémantique initiale. On vérifie que son résultat est $\texttt{Val}\ v$ ssi l'expression a renvoie le résultat v, et $\texttt{Exc}\ e$ ssi l'expression a lève l'exception e. Ce codage est donc une façon de définir la sémantique des exceptions.

L'intérêt de ce codage est de rétablir la symétrie entre résultats et exceptions. Dans les deux cas, il s'agit de valeurs calculées par l'expression ; la seule distinction provient du constructeur utilisé, à savoir `Val` ou `Exc`. L'existence de ce codage montre que le typage des exceptions n'est pas plus difficile que celui des résultats. En effet, il suffit de typer $[\![a]\!]$ pour obtenir non seulement le type de a, mais aussi le type des éventuelles exceptions lancées par a.

Cependant, en pratique, plutôt que de passer explicitement par ce codage, nous préférons introduire un jeu de règles de typage modifiées, donné par la figure 14.1 page suivante. Le principe reste le même ; le fait d'utiliser des règles dédiées évite de mélanger les types introduits par l'utilisateur avec ceux provenant du codage, et améliore donc la lisibilité.

Le constructeur $\rightarrow$ est maintenant ternaire ; il s'écrit $\alpha \rightarrow \beta\ \texttt{raises}\ \gamma$, où le premier argument est contravariant et les deux derniers covariants, et où γ représente le type des exceptions éventuellement lancées par la fonction. (Dans le codage, la fonction aurait admis le

$$\frac{\mathrm{dom}(A) = \mathrm{dom}_\lambda(\Gamma)}{\Gamma \vdash x : A \Rightarrow A(x)\ \texttt{raises}\ \bot \mid C} \qquad (\textsc{Var})$$

$$\frac{\mathrm{lift}_x(\Gamma); x \vdash e : (A; x : \tau) \Rightarrow \tau'\ \texttt{raises}\ \tau_e \mid C}{\Gamma \vdash \lambda x.e : A \Rightarrow (\tau \to \tau'\ \texttt{raises}\ \tau_e)\ \texttt{raises}\ \bot \mid C} \qquad (\textsc{Abs})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : A \Rightarrow (\tau_2 \to \tau\ \texttt{raises}\ \tau_e)\ \texttt{raises}\ \tau_e \mid C \\ \Gamma \vdash e_2 : A \Rightarrow \tau_2\ \texttt{raises}\ \tau_e \mid C\end{array}}{\Gamma \vdash e_1\, e_2 : A \Rightarrow \tau\ \texttt{raises}\ \tau_e \mid C} \qquad (\textsc{App})$$

$$\frac{\Gamma(X) = \sigma}{\Gamma \vdash X : \sigma} \qquad (\textsc{LetVar})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \sigma_1 \\ \sigma_1 = A_1 \Rightarrow \tau_1\ \texttt{raises}\ \tau_e' \mid C_1 \\ \sigma_1' = A_1 \Rightarrow \tau_1\ \texttt{raises}\ \bot \mid C_1 \\ \Gamma; X : \sigma_1' \vdash e_2 : A_2 \Rightarrow \tau_2\ \texttt{raises}\ \tau_e \mid C_2 \\ \sigma_1 \leq^\forall A_2 \Rightarrow \top\ \texttt{raises}\ \tau_e \mid C_2\end{array}}{\Gamma \vdash \texttt{let}\, X = e_1\ \texttt{in}\ e_2 : A_2 \Rightarrow \tau_2\ \texttt{raises}\ \tau_e \mid C_2} \qquad (\textsc{Let})$$

$$\frac{\Gamma \vdash e : \sigma \qquad \sigma \leq^\forall \sigma'}{\Gamma \vdash e : \sigma'} \qquad (\textsc{Sub})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : A \Rightarrow \tau\ \texttt{raises}\ \tau_e \mid C \\ \Gamma \vdash e_2 : A \Rightarrow (\tau_e \to \tau\ \texttt{raises}\ \tau_e')\ \texttt{raises}\ \tau_e' \mid C\end{array}}{\Gamma \vdash \texttt{try}\ e_1\ \texttt{with}\ e_2 : A \Rightarrow \tau\ \texttt{raises}\ \tau_e' \mid C} \qquad (\textsc{Try})$$

FIG. 14.1: Règles de typage, étendues pour l'analyse des exceptions

type $\alpha \to$ `[ Val: Pre` β`; Exc: Pre` γ`; Abs ]`.) Dans le même ordre d'idées, les schémas de types sont à présent de la forme $A \Rightarrow \tau$ `raises` $\tau_e \mid C$, où τ_e est le type des exceptions éventuellement lancées par l'expression. Dans les deux cas, une annotation `raises` $\perp$ signifie que la fonction ou que l'expression considérée ne lance pas d'exceptions, et pourra être omise pour plus de concision.

Ceci posé, les modifications apportées aux règles sont simples, et reflètent les contraintes que l'on obtiendrait en typant l'expression produite par le codage. On note l'apparition de la règle (Try). Quant à la construction `raise`, il n'est pas nécessaire de lui dédier une règle ; il suffit d'en faire une primitive, dont le schéma de types est ($\alpha \to \perp$ `raises` α) `raises` $\perp$.

Par souci de clarté, nous nous sommes limités à réécrire les règles de typage. Les règles d'inférence de types peuvent également être modifiées sans difficulté ; on prendra soin de respecter l'invariant de mono-polarité en introduisant, quand cela est nécessaire, deux variables fraîches reliées par une contrainte, plutôt qu'une unique variable (cf. section 12.3).

14.6.2 Application

Grâce au mécanisme de variables de rangée (cf. section 14.5) dans les types variantes, les filtrages reçoivent des types précis. Or, les gestionnaires d'exceptions sont souvent – et même toujours, en ML – à base de filtrages. L'analyse des exceptions qui en découle est donc suffisamment fine. Par exemple, le gestionnaire

```
fun (A x) -> x
  | e -> raise e
```

reçoit le type

$$\texttt{[A: Pre } \alpha\texttt{; } \rho \texttt{]} \to \alpha \texttt{ raises [A: Abs; } \rho \texttt{]}$$

Ce type est suffisamment expressif pour indiquer que l'exception e lancée par cette fonction ne peut pas être de la forme `A`.

En fait, d'une certaine façon, le type ci-dessus est une façon de coder la *soustraction* de l'élément `A` d'un ensemble de constructeurs. En effet, soit τ un type variante. τ peut être considéré comme une façon de coder un ensemble de constructeurs. Considérons alors la contrainte $\tau \leq$ `[ A: Pre` α`;` ρ `]`. Si cette contrainte est satisfaite, alors `[ A: Abs;` ρ `]` est un type variante identique à τ (à condition de choisir ρ minimal), mais privé du constructeur `A`. Il représente donc le même ensemble de constructeurs que τ, moins le constructeur `A`. Nous avons donc, d'une certaine façon, codé en termes de types une opération que l'on retrouve en termes ensemblistes dans d'autres analyses [25].

Cela explique pourquoi la fonction suivante :

```
function x ->
  try
    raise (if true then (A x) else (B x))
  with A x ->
    x
```

admet le type

$$\alpha \to \alpha \texttt{ raises [B: Pre } \alpha\texttt{; Abs]}$$

Le gestionnaire d'exceptions utilisé ici est précisément celui décrit plus haut. L'opération de soustraction a bien lieu : bien que le cœur de la fonction lance les exceptions `A` et `B`, le type indique, correctement, que seule l'exception `B` est observable de l'extérieur. Par ailleurs, on note l'utilisation du polymorphisme : le type du résultat de cette fonction est égal au type de son argument, bien que le flot de données passe, de façon interne, à travers une exception. Cette flexibilité ne serait pas possible en ML, où les arguments des exceptions sont obligatoirement monomorphes.

14.7 Sous-typage contre variables de rangée

Le lecteur pourrait avoir l'impression que l'analyse des exceptions présentée ci-dessus, ou, plus généralement, notre mécanisme de typage des filtrages, doivent leur finesse principalement à la présence des variables de rangée, et non à l'utilisation du sous-typage. Cette impression ne serait pas entièrement injustifiée : les variables de rangée constituent un raffinement très intéressant, qui fournit des résultats précis, même si on n'utilise que des contraintes d'unification. Cependant, cette précision n'est obtenue qu'à la condition de disposer d'un degré suffisant de polymorphisme, ce qui n'est pas toujours possible dans un langage où le polymorphisme n'est que de premier ordre. Le sous-typage, au contraire, fonctionne parfaitement en l'absence de tout polymorphisme. Détaillons quelque peu cette comparaison.

Rappelons que l'apport principal du sous-typage consiste à mémoriser la *direction* du flot de données, créant ainsi un graphe de contraintes *orienté* ; tandis qu'une analyse classique, basée sur l'unification, conduit à un graphe non orienté. Le second est *a priori* plus grossier, puisqu'il met ainsi en communication des points qui n'ont pas besoin de l'être. Cependant, en présence de polymorphisme, le mécanisme de généralisation et d'instantiation nous conduit à renommer certaines parties du graphe, brisant ainsi les communications indésirables.

Prenons un exemple. Supposons que nous disposions de simples types variantes polymorphes, c'est-à-dire de sous-typage sans variables de rangée. Soit x une variable, liée par λ, de type α. Si l'on passe x à deux fonctions dont l'une traite les cas `A` et `B` et l'autre les cas `A` et `C`, on engendre les contraintes

$$\alpha \leq \texttt{[A: unit | B: unit]}$$
$$\alpha \leq \texttt{[A: unit | C: unit]}$$

On constate immédiatement que cet ensemble de contraintes est clos, donc soluble, et cette double application est bien typée. (En fait, ces deux contraintes se réduisent en $\alpha \leq \texttt{[A: unit]}$, indiquant ainsi que x doit nécessairement être égal à `A`.)

Si, au contraire, nous disposons de variables de rangée, mais non de sous-typage, nous obtiendrons les contraintes

$$\alpha = \texttt{[A: Pre unit; B: Pre unit; Abs]}$$
$$\alpha = \texttt{[A: Pre unit; C: Pre unit; Abs]}$$

Par transitivité sur α, il nous faut unifier $\langle\texttt{B: Pre unit; Abs}\rangle$ avec $\langle\texttt{C: Pre unit; Abs}\rangle$, ce qui est incohérent. Les contraintes n'ont donc pas de solution. Ainsi, en remplaçant les inéquations par des équations, nous avons mis en communication `B` et `C`, ce qui conduit à une approximation trop grossière.

Cependant, dans certains cas, le polymorphisme permet d'éliminer le problème. Si, au lieu d'être introduit par un λ, x était lié par un construction `let` x = `A in` ..., alors le schéma de types associé à x dans l'environnement serait

$$\forall\rho.\texttt{[A: Pre unit; }\rho\texttt{]}$$

où ρ est une variable de rangée. Alors, chacune des deux applications utiliserait une instance différente de x, et les contraintes engendrées seraient

$$\texttt{[A: Pre unit; }\rho_1\texttt{]} = \texttt{[A: Pre unit; B: Pre unit; Abs]}$$
$$\texttt{[A: Pre unit; }\rho_2\texttt{]} = \texttt{[A: Pre unit; C: Pre unit; Abs]}$$

Ici, aucune transitivité n'a lieu, et les contraintes sont satisfiables. On constate que grâce au polymorphisme, les deux parties du graphe ont été séparées, évitant ainsi tout problème lié à la grossièreté de l'égalité.

Ainsi, variables de rangée et sous-typage sont complémentaires. Les premières sont indispensables au typage des enregistrements (ou des filtrages) extensibles. De plus, pour peu que l'on dispose d'un degré suffisant de polymorphisme, elles atténuent le besoin de sous-typage dans de nombreuses situations. Le sous-typage reste cependant intéressant dans les cas où l'on ne dispose pas de polymorphisme, par exemple parce qu'on manipule une variable liée par un λ. Ces cas peuvent se présenter en pratique : l'exemple ci-dessus, où une donnée inconnue (c'est-à-dire liée par λ) x est passée à deux fonctions de domaines différents, semble naturel. La présence de sous-typage est donc souhaitable dès que l'on manipule des types variantes. Cet exemple peut se transposer, symétriquement, au cas des enregistrements. Le problème se manifeste lorsqu'un message inconnu m – par « message », on entend ici fonction d'accès de la forme $\lambda x.(x.\texttt{champ})$ – est appliqué à deux enregistrements possédant des champs différents. Le besoin de sous-typage se fait donc sentir lorsque l'on désire manipuler les messages comme des valeurs de première classe.

Chapitre 15

Implémentation

NOTRE EXPOSÉ THÉORIQUE est à présent terminé – nous avons en main tous les outils nécessaires à la réalisation d'un moteur d'inférence et de simplification de types efficace. Il ne reste plus qu'à réaliser l'assemblage, et à compléter le tout par un module d'affichage.

La section 15.1 rappelle les différents composants qui constituent le moteur, et décrit la façon dont ils s'assemblent. La phase de simplification « externe », effectuée immédiatement avant l'affichage d'un schéma de types, est décrite par la section 15.2. Le fonctionnement de l'ensemble est ensuite résumé à l'aide d'un exemple (section 15.3). Enfin, la section 15.4 donne une mesure des performances de notre implémentation.

15.1 Moteur

Ce que nous appellerons le *moteur* constitue la partie centrale du système. Son travail consiste d'abord à construire une dérivation d'inférence pour le programme fourni, à l'aide des règles données par la figure 12.1 page 128. D'après les résultats de la section 5.6, il s'agit là d'une passe linéaire sur le λ-terme. Celle-ci fournit un schéma de types principal, à condition de vérifier que les contraintes obtenues ont bien une solution.

Pour cela, le moteur doit calculer la clôture du graphe de contraintes construit par cette première passe. Cela se fait aisément à l'aide de l'algorithme de clôture incrémentale décrit par la définition 7.1. Si l'algorithme signale un échec, le programme est mal typé. Sinon, il est bien typé, et l'algorithme produit un graphe de contraintes clos.

Enfin, le moteur effectue la simplification du schéma inféré. Pour cela, il lui applique les trois algorithmes de simplification décrits dans cette thèse : canonisation, dépoussiérage et minimisation. Avant la canonisation, le schéma est donc clos, et contient éventuellement des occurrences des constructeurs $\sqcup$ et $\sqcap$. Après la canonisation, toute occurrence de $\sqcup$ ou $\sqcap$ a disparu, et le schéma est, d'après la proposition 11.11, une version partiellement dépoussiérée d'un schéma simplement clos. Il est donc légal de calculer ses polarités et de lui appliquer l'algorithme de dépoussiérage. La phase de dépoussiérage produit un schéma parfait, qui peut être minimisé, pour donner finalement un autre schéma parfait. Les trois phases de simplification se combinent donc parfaitement.

Nous venons de décrire le fonctionnement du moteur en trois phases bien distinctes : génération des contraintes, clôture et simplification. En théorie, ces trois phases peuvent avoir lieu successivement. Cependant, en pratique, il est fondamental, pour des raisons d'efficacité, que la simplification soit effectuée au fur et au mesure que les contraintes apparaissent. En effet, la construction `let` permet de placer dans l'environnement le schéma de types associé à une certaine sous-expression ; chaque utilisation de la variable X liée par cette construction engendrera alors une nouvelle copie de ce schéma. Il est donc important de ne placer dans l'environnement que des schémas entièrement simplifiés, sous peine de réaliser ensuite plusieurs fois le même travail de clôture et de simplification.

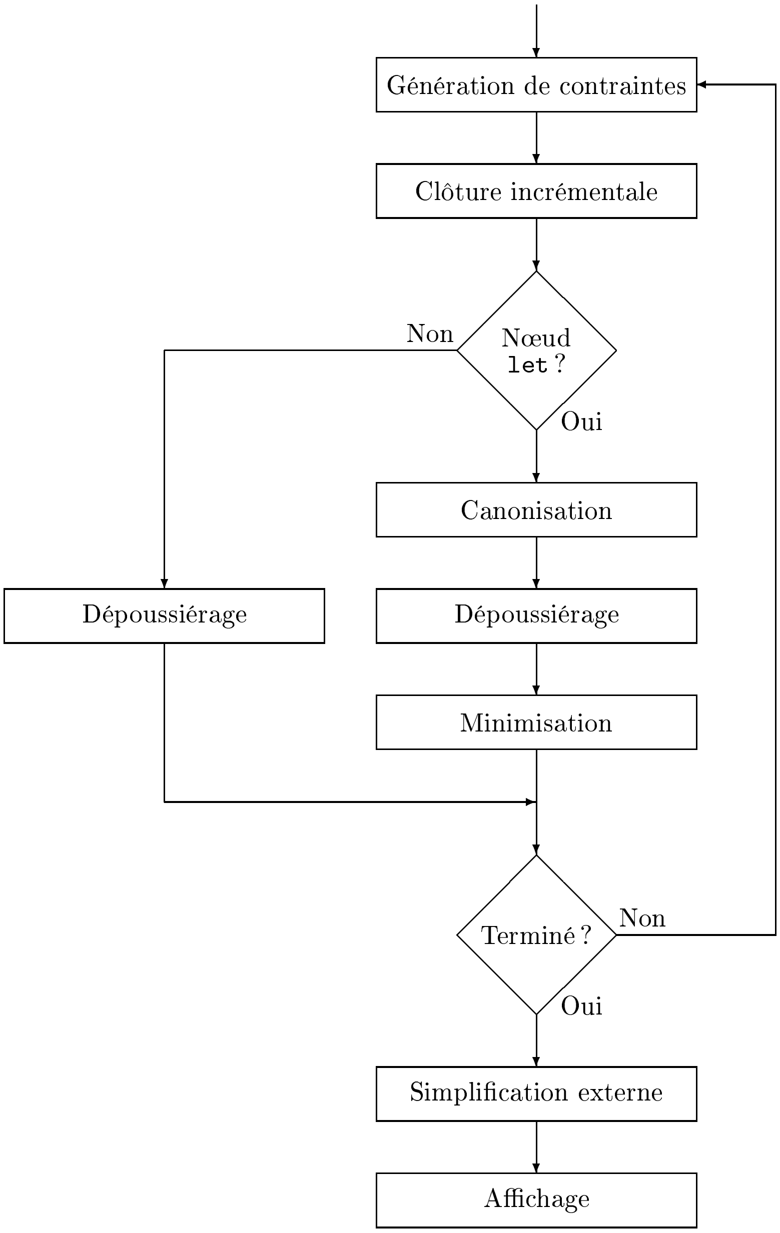

FIG. 15.1: Processus d'inférence et de simplification

Cela ne pose pas de problème. Il suffit, lorsqu'on rencontre un nœud `let`, d'interrompre la phase de ramassage des contraintes, et d'effectuer la clôture et la simplification du schéma obtenu, avant de l'ajouter à l'environnement. La simplification produit un schéma parfait, donc en particulier clos, et ne gêne donc pas le calcul de clôture incrémental qui suivra.

On peut aller plus loin et effectuer certains calculs non seulement aux nœuds `let`, mais à tous les nœuds. L'algorithme de clôture, par exemple, est incrémental, et il n'y a donc aucun inconvénient à mettre à jour la clôture à chaque fois qu'une nouvelle contrainte apparaît.

Quant aux trois algorithmes de simplification, aucun n'est incrémental ; il faut donc expérimenter pour savoir s'il est rentable de les exécuter à tous les nœuds. Chaque algorithme est *a priori* coûteux, puisqu'il analyse l'ensemble du graphe de contraintes, et non seulement les contraintes ajoutées récemment. Cependant, s'il est suffisamment puissant, il peut apporter une simplification sensible et accélérer ainsi les phases suivantes. Ainsi, l'algorithme de dépoussiérage est peu coûteux, parce qu'il se contente de détruire des contraintes et ne nécessite donc (presque) aucune allocation, et très efficace, parce qu'il élimine un grand nombre de variables intermédiaires. Il est donc rentable de l'exécuter à chaque nœud, car cela facilitera les calculs de clôture qui lui succèderont. Par exemple, dans un cas typique, le fait d'effectuer le dépoussiérage à tous les nœuds augmente le temps qui lui est consacré d'un facteur 2, mais diminue le temps consacré à la clôture d'un facteur 5. Quant à la canonisation, elle ne facilite que peu les calculs de clôture à venir ; au contraire, elle annule le léger gain d'efficacité lié à l'utilisation des constructeurs $\sqcup$ et $\sqcap$. Enfin, la minimisation, bien que quasi-linéaire, est assez coûteuse. En pratique, le dépoussiérage sera donc la seule simplification effectuée à chaque nœud. (Notons que pour l'effectuer avant la canonisation, il faut prouver qu'il est correct en présence des constructeurs $\sqcup$ et $\sqcap$, ce qui ne pose pas de problème, sachant que les polarités diminuent lors de la canonisation.)

Le moteur d'inférence constitue la partie supérieure du diagramme de la figure 15.1 page précédente.

Disons quelques mots de la représentation des graphes de contraintes en machine. Rappelons que dans une dérivation d'inférence, aucune variable n'est partagée entre deux branches distinctes (cf. lemme 5.2). Par conséquent, à tout moment de l'exécution du moteur d'inférence, toute variable appartient à au plus un graphe de contraintes. Il est donc possible de stocker les contraintes la concernant au sein de la variable elle-même, de sorte qu'elles sont accessibles et modifiables en temps constant. Du point de vue de la machine, un schéma de types consiste simplement en un contexte A et un corps τ ; le graphe de contraintes associé s'obtient implicitement, en suivant les contraintes à partir de ces points d'entrée. Cette représentation est légère d'emploi et efficace. Elle permet même l'élimination automatique des variables inaccessibles par le glaneur de cellules, sans qu'il soit nécessaire d'utiliser le dépoussiérage pour les éliminer explicitement !

15.2 Affichage

Nous avons expliqué, après l'introduction de l'invariant de mono-polarité, que certaines méthodes de « simplification » tendent à augmenter l'efficacité du moteur d'inférence, tandis que d'autres cherchent en fait à améliorer la lisibilité du résultat (cf. section 12.4). Nous avons constaté que les secondes entrent nécessairement en conflit avec les premières, puisqu'elles violent les invariants fondamentaux que sont l'invariant des petits termes et l'invariant de mono-polarité. C'est pourquoi il ne faut les utiliser que lors d'une dernière phase avant l'affichage, et non pendant le fonctionnement du moteur d'inférence. Ainsi, on pourra qualifier les premières *d'internes*, puisqu'elles font partie intégrante du moteur, tandis que les secondes seront appelées *externes*, parce qu'elles ne constituent somme toute qu'un dispositif d'affichage.

Il faut insister sur cette distinction, car elle est fondamentale. Vouloir améliorer en même temps efficacité et lisibilité serait une grave erreur de conception – erreur que nous avons d'ailleurs commise, historiquement, et qui nous a mené à des problèmes inextricables, puisque certaines « simplifications » construisaient des termes que d'autres détruisaient, le tout ayant un comportement fort aléatoire.

On retrouve d'ailleurs une distinction similaire en ML, où les types sont présentés à l'utilisateur comme des arbres, mais sont en fait représentés par des graphes de façon interne. Si on travaillait directement sur des arbres, on perdrait le partage de certains nœuds et on pourrait subir une perte d'efficacité exponentielle, comme l'ont montré les premières versions de certains typeurs ML. De même, dans notre système, ne pas respecter l'invariant des petits termes reviendrait, comme nous l'avons expliqué dans la section 6.4, à une perte de partage.

La simplification classique qui consiste à remplacer une variable par son unique borne [15, 4, 8, 43] doit donc être considérée comme externe, et effectuée uniquement avant l'affichage. Elle constitue la partie inférieure du diagramme de la figure 15.1.

Nous n'avons pas encore décrit cette simplification en termes formels. Nous allons le faire à présent ; elle est extrêmement simple.

Définition 15.1 *Soit $\sigma = A \Rightarrow \tau \mid C$ un schéma de types parfait. Soit $\alpha \in \mathrm{dom}^+(\sigma)$. Alors*

- *si $\mathrm{pred}_C(\alpha) = \varnothing$, alors α admet pour unique borne $C^{\downarrow}(\alpha)$;*
- *si $\mathrm{pred}_C(\alpha) = \{\beta\}$ et $C^{\downarrow}(\alpha) = \bot$, alors α admet pour unique borne β.*

Une définition symétrique s'applique aux variables négatives.

Proposition 15.1 *Soit σ un schéma de types parfait. Soit σ' le schéma obtenu en ajoutant à σ la contrainte $\alpha = \tau$ à chaque fois qu'une variable α admet pour unique borne un terme τ. Alors $\sigma =^{\forall} \sigma'$.*

Démonstration. Soit α une variable positive admettant pour unique borne un terme construit τ. Alors on ajoute la contrainte $\alpha \leq \tau$. Puisque l'unique borne inférieure de α est τ, la seule conséquence par transitivité de cet ajout est $\tau \leq \tau$, qui est trivialement contenue dans le graphe. Donc, le graphe reste clos.

Soit maintenant α une variable positive admettant pour unique borne une variable β. Alors on ajoute la contrainte $\alpha \leq \beta$. Pour que le graphe reste clos, il faut ajouter les contraintes $\alpha \leq \gamma$ pour tout γ tel que $\beta \leq_C \gamma$, ainsi que la contrainte $\alpha \leq C^{\uparrow}(\beta)$. Du fait que α n'a pas d'autre borne inférieure que β, le calcul de clôture s'arrête là.

Le cas des variables négatives est symétrique. Nous avons donc mis le schéma σ' sous une forme close. Il est alors légal d'effectuer sur cette forme un calcul de dépoussiérage. Or, on vérifie immédiatement que les polarités ne sont pas modifiées, puisque les bornes inférieures (resp. supérieures) construites des variables positives (resp. négatives) n'ont pas changé. Il en découle que toutes les contraintes nouvellement ajoutées sont éliminées par le dépoussiérage. Cela prouve l'équivalence $\sigma =^{\forall} \sigma'$. □

On peut donc imposer l'égalité entre une variable et son unique borne, pour peu que celle-ci existe, sans modifier la dénotation du schéma considéré. Par conséquent, on peut également substituer la borne à la variable, pour peu que la variable n'apparaisse pas libre dans sa borne. C'est donc en cela que consiste notre méthode de simplification externe, destinée à améliorer la lisibilité d'un schéma de types avant sa présentation à l'utilisateur.

Exemple. Voici une fonction classique, qui calcule la longueur d'une liste :

```
let rec list_length = function
  Nil -> 0
| Cons (_, rest) -> succ (list_length rest);;
```

Le schéma de types produit par le moteur d'inférence est α_1^+ `raises` $\alpha_2^+ \mid C$, où C est le graphe de contraintes suivant :

$$\begin{aligned} \alpha_8^- \to \alpha_3^+ \ \texttt{raises}\ \alpha_2^+ &\leq \alpha_1^+ \\ \bot &\leq \alpha_2^+ \end{aligned}$$

$$\begin{aligned}
\texttt{int} \leq \alpha_3^+ & \\
\alpha_4^- &\leq \texttt{Abs} \\
\alpha_5^- &\leq \texttt{unit} \\
\alpha_6^- &\leq \texttt{Pre } \alpha_5^- \\
\alpha_7^- &\leq \top \\
\alpha_8^- &\leq \texttt{[Nil: } \alpha_6^-\texttt{; Cons: } \alpha_{10}^-\texttt{; } \alpha_4^- \texttt{]} \\
\alpha_9^- &\leq \alpha_7^- \times \alpha_8^- \\
\alpha_{10}^- &\leq \texttt{Pre } \alpha_9^-
\end{aligned}$$

Ici, toutes les variables admettent une unique borne. En effectuant autant de substitutions que possible, nous obtenons le schéma $\alpha_8^- \rightarrow \texttt{int} \mid D$, où D est donné par

$$\alpha_8^- \leq \texttt{[Nil: Pre unit; Cons: Pre } \top \times \alpha_8^-\texttt{; Abs]}$$

(Les annotations `raises` portées par la flèche et par le schéma ont disparu, car nous avons convenu de les omettre quand elles sont de la forme $\texttt{raises } \bot$.) Bien que α_8 admette une unique borne supérieure, la substitution n'a pas pu être effectuée, car α_8 apparaît dans sa propre borne. De façon informelle, on pourrait introduire un lieur μ, et on obtiendrait alors le schéma

$$\mu\alpha_8.\texttt{[Nil: Pre unit; Cons: Pre } \top \times \alpha_8\texttt{; Abs]} \rightarrow \texttt{int}$$

On note que, en présence de contraintes récursives, l'ordre dans lequel les substitutions sont effectuées peut mener à différents résultats. Par exemple, si on avait d'abord remplacé α_8 par sa borne, on aurait pu obtenir $\texttt{[Nil: Pre unit; Cons: } \alpha_{10}^-\texttt{; Abs]} \rightarrow \texttt{int} \mid D'$, où D' est donné par

$$\alpha_{10}^- \leq \texttt{Pre } \top \times \texttt{[Nil: Pre unit; Cons: } \alpha_{10}^-\texttt{; Abs]}$$

Ce schéma est bien sûr équivalent, mais moins lisible, parce qu'un type récursif a été partiellement déplié. Il serait donc intéressant de pouvoir choisir quelles variables seront substituées et quelles variables resteront au contraire visibles, de telle façon que la représentation textuelle qui en découle soit optimale. Traçons un graphe reliant chaque variable aux variables libres de son unique borne construite, si celle-ci existe :

$$\begin{tikzcd}
\alpha_3 & \alpha_1 \arrow[l] \arrow[r] \arrow[d] & \alpha_2 & \\
\alpha_6 \arrow[d] & \alpha_8 \arrow[l] \arrow[r] \arrow[d] & \alpha_4 & \\
\alpha_5 & \alpha_{10} \arrow[r] & \alpha_9 \arrow[r] \arrow[ul] & \alpha_7
\end{tikzcd}$$

L'impossibilité de substituer une variable par un terme dans lequel celle-ci apparaît se traduit alors par le fait que dans ce graphe, tout cycle doit comporter au moins une variable visible. Dans l'exemple ci-dessus, on remarque un cycle entre α_8, α_{10} et α_9 ; l'une au moins de ces variables doit rester visible.

On pourrait décider, dans chaque cycle, de choisir de préférence une variable aussi proche que possible des points d'entrée du graphe. Ici, les points d'entrée étant α_1 et α_2, c'est α_8 qui est la plus proche. En choisissant de laisser α_8 visible et de substituer α_9 et α_{10}, on obtient effectivement une représentation optimale.

Cependant, en général, dans un cycle, la variable la plus proche des points d'entrée n'est pas nécessairement unique. De plus, d'autres critères semblent également valables. Par exemple, si une variable donnée appartient à plusieurs cycles, il serait intéressant de la laisser visible. Ce problème semble donc difficile.

Pour conclure, nous n'avons pas étudié ce problème d'optimalité plus avant. D'une part, il semble délicat. D'autre part, il semble d'importance relativement mineure, puisqu'en pratique, un choix aléatoire conduit à de bons résultats dans la plupart des cas.

15.3 Un exemple complet

Cette section décrit l'opération du moteur d'inférence et de l'afficheur sur un exemple de petite taille, mais assez typique, à savoir la fonction classique `map` sur les listes. Pour plus de simplicité, nous utilisons ici des types variantes polymorphes (sans variables de rangée), et nous n'effectuons pas l'analyse des exceptions.

```
rec map in function f -> function
    Nil -> Nil
  | Cons (x, rest) -> Cons (f x, map f rest)
```

Pour simplifier les choses, nous supposerons que les algorithmes de simplification ne sont exécutés qu'après le ramassage des contraintes.

Les règles d'inférence indiquent que l'expression a pour type v_{22}, accompagné des contraintes suivantes, dans le désordre :

$v_3 \leq v_4$	$v_5 \leq v_6$	$v_7 \leq v_8$	$v_4 \leq v_6 \to v_7$
$v_{11} \leq v_{12}$	$v_{13} \leq v_{14}$	$v_1 \leq v_{20}$	$v_{12} \leq v_{14} \to v_{15}$
$v_8 * v_{16} \leq v_{17}$	$v_{19} \to v_{20} \leq v_{21}$	$v_2 \leq v_5 * v_{13}$	$\texttt{[Cons of } v_{17} \texttt{]} \leq v_{18}$
$v_{18} \leq v_{20}$	$v_{22} \leq v_9$	$v_9 \leq v_{10}$	$v_{19} \leq \texttt{[Nil | Cons of } v_2 \texttt{]}$
$\texttt{[Nil]} \leq v_1$	$v_3 \to v_{21} \leq v_{22}$	$v_{15} \leq v_{16}$	$v_{10} \leq v_4 \to v_{11}$

Calculons la clôture de ces contraintes. Elle est donnée ci-dessous, sous la forme d'un graphe de contraintes.

$$
\begin{array}{rcl}
\texttt{[Nil]} \leq & v_1 & \leq v_{15},\, v_{16},\, v_{20} \\
 & v_2 & \leq v_5 * v_{13} \\
v_4 \leq & v_3 & \leq v_4,\, v_6 \to v_7 \\
v_3 \leq & v_4 & \leq v_3,\, v_6 \to v_7 \\
 & v_5 & \leq v_6 \\
v_5 \leq & v_6 & \\
 & v_7 & \leq v_8 \\
v_7 \leq & v_8 & \\
v_3 \to v_{21},\, v_{22} \leq & v_9 & \leq v_{10},\, v_4 \to v_{11} \\
v_3 \to v_{21},\, v_9,\, v_{22} \leq & v_{10} & \leq v_4 \to v_{11} \\
v_{19} \to v_{20},\, v_{21} \leq & v_{11} & \leq v_{12},\, v_{14} \to v_{15} \\
v_{19} \to v_{20},\, v_{11},\, v_{21} \leq & v_{12} & \leq v_{14} \to v_{15} \\
 & v_{13} & \leq v_{14},\, v_{19},\, \texttt{[Nil | Cons of } v_2 \texttt{]} \\
v_{13} \leq & v_{14} & \leq v_{19},\, \texttt{[Nil | Cons of } v_2 \texttt{]} \\
\texttt{[Nil | Cons of } v_{17} \texttt{]},\, v_1,\, v_{18},\, v_{20} \leq & v_{15} & \leq v_{16} \\
\texttt{[Nil | Cons of } v_{17} \texttt{]},\, v_1,\, v_{18},\, v_{15},\, v_{20} \leq & v_{16} & \\
v_8 * v_{16} \leq & v_{17} & \\
\texttt{[Cons of } v_{17} \texttt{]} \leq & v_{18} & \leq v_{15},\, v_{16},\, v_{20} \\
v_{13},\, v_{14} \leq & v_{19} & \leq \texttt{[Nil | Cons of } v_2 \texttt{]} \\
\texttt{[Nil | Cons of } v_{17} \texttt{]},\, v_1,\, v_{18} \leq & v_{20} & \leq v_{15},\, v_{16} \\
v_{19} \to v_{20} \leq & v_{21} & \leq v_{11},\, v_{12},\, v_{14} \to v_{15} \\
v_3 \to v_{21} \leq & v_{22} & \leq v_9,\, v_{10},\, v_4 \to v_{11}
\end{array}
$$

Ici, le calcul de clôture n'a fait apparaître aucun constructeur $\sqcup$ ou $\sqcap$. En fait, certains sont apparus pour combiner des bornes multiples en une seule borne, mais ils ont été aussitôt éliminés par les règles de mise en forme normale de la définition 2.2. Par exemple, $\texttt{[Nil]} \sqcup \texttt{[Cons of } v_{17} \texttt{]} = \texttt{[Nil | Cons of } v_{17} \texttt{]}$. L'algorithme de canonisation n'a donc aucun effet.

Passons au calcul des polarités. Nous commençons par marquer l'unique point d'entrée, v_{22}, positif, puis nous laissons les marques se propager à travers le graphe jusqu'à atteindre un point fixe. Nous apprenons ainsi que $v_6, v_8, v_{16}, v_{17}, v_{20}, v_{21}, v_{22}$ sont positives, tandis que $v_2, v_3, v_5, v_7, v_{13}, v_{19}$ sont négatives.

Grâce à cette information, nous pouvons effectuer le dépoussiérage. Les variables positives (resp. négatives) perdent leur bornes supérieures (resp. inférieures) ; de plus, les contraintes entre variables ne sont conservées que si celle de gauche est négative et celle de droite positive. On obtient

$$
\begin{aligned}
v_2 &\leq v_5 * v_{13} \\
v_3 &\leq v_6 \to v_7 \\
v_5 &\leq v_6 \\
v_5 &\leq v_6 \\
v_7 &\leq v_8 \\
v_7 &\leq v_8 \\
v_{13} &\leq \texttt{[Nil | Cons of } v_2 \texttt{]} \\
\texttt{[Nil | Cons of } v_{17} \texttt{]} &\leq v_{16} \\
v_8 * v_{16} &\leq v_{17} \\
v_{19} &\leq \texttt{[Nil | Cons of } v_2 \texttt{]} \\
\texttt{[Nil | Cons of } v_{17} \texttt{]} &\leq v_{20} \\
v_{19} \to v_{20} &\leq v_{21} \\
v_3 \to v_{21} &\leq v_{22}
\end{aligned}
$$

Nous pouvons à présent effectuer la minimisation. On calcule la plus grande partition telle que deux variables équivalentes aient la même polarité, les mêmes ensembles de prédécesseurs et de successeurs, et des bornes construites équivalentes. Ici, il est facile de voir que les seules classes d'équivalence non triviales sont $\{v_{13}, v_{19}\}$ et $\{v_{16}, v_{20}\}$.

Notons que remplacer v_{19} par v_{13} revient en fait à dérouler un point fixe partiellement déplié. En fait, cela revient essentiellement à remplacer $F(\mu t.F(t))$ par $\mu t.F(t)$, où F est l'opérateur de types

$$t \mapsto \texttt{[Nil | Cons of } v_5 * t \texttt{]}$$

Une remarque similaire peut être formulée à propos de v_{16} et v_{20}. Après passage au quotient, le graphe de contraintes devient

$$
\begin{aligned}
v_2 &\leq v_5 * v_{13} \\
v_3 &\leq v_6 \to v_7 \\
v_5 &\leq v_6 \\
v_5 &\leq v_6 \\
v_7 &\leq v_8 \\
v_7 &\leq v_8 \\
v_{13} &\leq \texttt{[Nil | Cons of } v_2 \texttt{]} \\
\texttt{[Nil | Cons of } v_{17} \texttt{]} &\leq v_{16} \\
v_8 * v_{16} &\leq v_{17} \\
v_{13} \to v_{16} &\leq v_{21} \\
v_3 \to v_{21} &\leq v_{22}
\end{aligned}
$$

La simplification est terminée ; le travail du moteur s'arrête là. Reste à appliquer la méthode de simplification externe, de façon à rendre le schéma plus lisible. Nous remplaçons donc chaque variable par son unique borne, sauf quand celle-ci est récursive. Nous obtenons finalement pour `map` le schéma $(v_6 \to v_8) \to v_{13} \to v_{16} \mid C$, où C est le graphe de contraintes

$$
\begin{aligned}
v_{13} &\leq \texttt{[Nil | Cons of } v_6 * v_{13} \texttt{]} \\
\texttt{[Nil | Cons of } v_8 * v_{16} \texttt{]} &\leq v_{16}
\end{aligned}
$$

Ce résultat est optimal, dans la mesure où nous travaillons uniquement avec des types concrets, c'est-à-dire que nous n'avons pas défini de type « liste ». Si l'on désire aller un peu

	Lignes	Sec.	/	Clô.	Can.	Dép.	Min.	Caml-Light
Lib. Graphes	900	2s	450	45%	10%	30%	15%	1s
Lib. standard CL	4300	5s	860	35%	20%	25%	15%	6s
Lib. Format	1100	7s	160	25%	10%	25%	40%	1s
Lib. MLgraph	9900	27s	370	35%	10%	25%	30%	13s

FIG. 15.2: Performances de l'implémentation

plus loin, on peut introduire des lieurs μ comme dans la section 15.2. Le schéma devient alors

$$(v_6 \rightarrow v_8) \rightarrow \mu t.[\texttt{ Nil | Cons of } v_6 * t \texttt{]} \rightarrow \mu t.[\texttt{ Nil | Cons of } v_8 * t \texttt{]}$$

15.4 Performances

Les algorithmes décrits dans cette thèse ont été implémentés au sein d'un prototype, dont le code source devrait être prochainement disponible électroniquement. Ce typeur prototype analyse un λ-calcul enrichi dont la syntaxe est proche de celle de Caml, y compris les aspects impératifs. Le langage de types contient toutes les extensions exposées au chapitre 14.

La figure 15.2 présente les résultats de quelques mesures de performance de ce prototype. Celui-ci, compilé en langage machine par Objective Caml 1.07, a été testé sur un processeur Pentium Pro à une fréquence de 150MHz.

Les programmes échantillons utilisés pour ce test ont été écrits par différents auteurs en Caml-Light, et adaptés à notre langage à l'aide d'un traducteur sommaire. Ils n'utilisent donc nullement la présence de sous-typage ; néanmoins, ils constituent une mesure pertinente de l'efficacité de notre moteur d'inférence. Notre prototype ne supportant pas la compilation séparée, chaque exemple a été mis par le traducteur sous la forme d'un unique module, ne contenant aucune information de type. Les informations de typage – en particulier les déclarations de types abstraits – contenues dans l'interface des modules Caml-Light n'ont donc pas été utilisées. Ainsi, les types obtenus sont, la plupart du temps, beaucoup plus précis que ceux inférés par Caml-Light, et la tâche du typeur est nettement plus lourde.

La librairie Graphes (écrite par Laurent Chéno) contient quelques primitives de gestion de graphes. Les deux entrées suivantes concernent la librairie standard du langage Caml-Light, dont nous avons isolé le module Format, singulièrement plus difficile à traiter que les autres modules, parce qu'il met en jeu des types beaucoup plus complexes. Enfin, la librairie MLgraph (écrite par Emmanuel Chailloux et Guy Cousineau) fournit un jeu de primitives de manipulation d'objets graphiques.

Dans chaque cas, on a d'abord indiqué la taille de l'exemple considéré, en nombre de lignes de code. On a ensuite mesuré le temps nécessaire au typage, en secondes de temps utilisateur ; d'où on déduit le nombre de lignes traitées par seconde. Les quatre colonnes suivantes donnent la répartition du temps entre les diverses phases du moteur d'inférence : création des contraintes et clôture incrémentale ; canonisation ; dépoussiérage ; et minimisation. Enfin, la dernière colonne indique, à titre de référence, le temps demandé par le compilateur Caml-Light pour traiter ce code.

Les librairies Graphes et MLgraph sont constituées de fonctions de taille moyenne. Les performances de notre typeur, dans ces cas que l'on peut probablement considérer comme typiques, sont de l'ordre de 400 lignes par seconde. L'écart vis-à-vis des temps obtenus par le compilateur Caml-Light, lequel est connu pour sa rapidité, n'est alors que d'un facteur 2. Il s'agit donc d'un résultat satisfaisant.

La librairie standard est constituée de fonctions de petite taille ; la vitesse de traitement double alors. La librairie Format, enfin, contient des fonctions assez complexes, qui se voient attribuer des types concrets de grande taille ; la performance diminue alors de moitié. Ainsi, quelques difficultés se présentent encore lorsque l'on rencontre des schémas de types de très grande taille. Rappelons que l'apparition de tels schémas est due en grande partie à notre utilisation exclusive de types concrets ; dans un langage réaliste, la modularité permettrait l'introduction de types abstraits, ce qui conduirait à des schémas de taille largement inférieure. La décision de manipuler uniquement des types concrets permet donc de mettre notre implémentation à l'épreuve ; les problèmes de performance rencontrés ici n'apparaîtraient pas nécessairement lors du typage d'un langage plus réaliste et modulaire.

Comment résoudre ces problèmes résiduels ? D'abord, on peut souhaiter mettre au point des algorithmes de simplification incrémentaux. En effet, un algorithme non incrémental, même linéaire, conduit à un comportement quadratique, puisqu'il est exécuté à chaque nœud de l'arbre de syntaxe (ou a chaque nœud `let`). La question d'un algorithme de canonisation incrémental a été soulevée à la section 11.5. Il semble également possible d'utiliser l'algorithme de minimisation de façon plus incrémentale, au prix d'une certaine perte de puissance. En effet, si on sait établir une distinction entre parties « anciennes » et « récentes » dans le graphe de contraintes, alors on peut choisir une partition initiale dans laquelle toutes les variables anciennes sont isolées. L'algorithme ne pourra ainsi découvrir de partage qu'entre variables récentes. En contrepartie, tous les liens entre variables anciennes, et toutes les variables anciennes non reliées à une variable récente, pourront être ignorés lors de la phase de raffinement, d'où un temps de calcul dépendant seulement de la taille de la partie récente. Reste à savoir si un tel compromis est rentable.

Cependant, notons que ces problèmes d'efficacité ne sont pas dûs uniquement à nos algorithmes de simplification. En effet, dans tous les cas considérés, la phase de création des contraintes et de clôture occupe une part constante du temps, à savoir environ un tiers. On peut donc dire que les phases de simplification ne sont pas un facteur limitant : dans le cas idéal où la simplification serait effectuée sans aucun coût, la complexité du processus d'inférence resterait la même. Il s'agit là d'un résultat fort satisfaisant, qui était loin d'être acquis lorsque ce travail a été entrepris. Il en découle que, pour améliorer nos performances, nous devrons à présent également tenter d'accélérer la phase de clôture. On peut songer à effectuer une analyse topologique des contraintes, avant ou pendant la phase de clôture. Analyse qui permettrait par exemple d'accélérer la convergence de l'algorithme de clôture en lui présentant les contraintes dans un ordre optimal, ou encore d'en amoindrir la tâche en effectuant des simplifications au vol, comme l'élimination des cycles [17]. Ces idées restent actuellement à étudier. Par ailleurs, il est également possible qu'une formulation différente de notre système de typage permette d'engendrer un moindre nombre de contraintes ; cette idée sera détaillée au chapitre 16.

Pour conclure, l'efficacité obtenue est très satisfaisante pour des programmes de petite taille. Le temps dédié à la simplification semble du même ordre que celui consacré à la clôture, ce qui indique que nos algorithmes de simplification ne sont pas un facteur limitant, et témoigne donc en leur faveur. En revanche, notre implémentation n'est pas suffisamment efficace pour traiter aisément des programmes de grande taille. Le problème mérite donc d'être étudié plus avant ; le chapitre suivant propose, pour cela, quelques pistes de recherche.

Chapitre 16

Difficultés et perspectives

Le système étudié au cours de cette thèse bénéficie d'une formalisation relativement simple et bien comprise. Néanmoins, il n'est pas parfait, et plusieurs points laissent à désirer. Cela se traduit, en pratique, par des problèmes d'expressivité ou d'efficacité. Nous passons ici en revue quelques-uns de ces problèmes, puis envisageons diverses façons de les résoudre. Bien entendu, il ne s'agit là que de pistes de recherche ; ces questions demandent une étude plus approfondie.

Le premier des problèmes considérés est l'ajout de constructions impératives au langage (section 16.1). La chose est aisée, tant qu'on se limite au typage d'une expression unique, c'est-à-dire d'un programme non modulaire ; cependant, des limitations apparaissent lorsque l'on désire obtenir la compilation séparée. La seconde faille, mise en évidence par la section 16.2, est une inefficacité qui semble inhérente à notre utilisation du λ-*lifting*. Celle-ci provoque en effet la duplication d'une partie de l'information de typage, augmentant ainsi inutilement la charge des algorithmes de simplification. En réponse à ces problèmes, nous proposons quelques solutions éventuelles : la création automatique d'abréviations de types (section 16.3), ou l'adoption d'un système de typage « à la ML » (section 16.4).

16.1 Typage des programmes impératifs

Le langage que nous avons étudié jusqu'ici est purement fonctionnel, c'est-à-dire que sa sémantique s'exprime simplement par une relation de réécriture entre termes, sans qu'il soit nécessaire de faire intervenir la notion d'état mémoire. Cependant, certains algorithmes s'expriment de façon plus simple, ou plus efficace, sous forme impérative. Il est donc souhaitable, comme en ML, d'offrir à l'utilisateur la possibilité de manipuler des *cellules mémoire mutables*, ou *références*.

Nous ajoutons donc au langage trois opérations primitives : `ref`, `!` et `:=`, permettant respectivement de créer une cellule, d'en lire le contenu et de le modifier. Comme indiqué dans la section 14.3, ces primitives acceptent les schémas de types suivants :

$$\begin{aligned} \texttt{ref} &: \alpha \rightarrow (\alpha, \alpha)\ \texttt{ref} \\ \texttt{!} &: (\alpha, \beta)\ \texttt{ref} \rightarrow \beta \\ \texttt{:=} &: (\alpha, \beta)\ \texttt{ref} \rightarrow \alpha \rightarrow \texttt{unit} \end{aligned}$$

où le constructeur de types `ref` est binaire, contravariant vis-à-vis de son premier argument et covariant vis-à-vis du second.

On reformule ensuite la sémantique du langage, de façon à donner une signification formelle à ces trois opérations. Cela est tout à fait classique [52, 34], aussi ne donnerons-nous pas explicitement cette sémantique étendue. Reste à vérifier que le système de typage est toujours correct vis-à-vis de la sémantique. Or, il est bien connu que ce résultat est faux, à cause de l'interaction entre références et polymorphisme. Par exemple, le programme

```
let x = ref (fun x -> x) in
x := succ;
!x true
```

est bien typé, parce que la construction `let` crée une « référence polymorphe » `x`, qui peut ensuite être utilisée indifféremment avec des entiers ou des booléens ; cependant, ce programme conduit à une erreur d'exécution, puisqu'il calcule `succ true`.

Il convient donc, pour éviter ce problème, d'adopter une restriction du polymorphisme. Diverses possibilités ont été étudiées, dans le cadre de ML, par Leroy [34]. Cependant, la solution la plus simple est probablement celle proposée par Wright [50, 51] ; c'est celle que nous choisissons ici. Elle consiste, sur la base d'un critère syntaxique extrêmement simple, à identifier les constructions `let` dont le corps est une expression *expansive* (c'est-à-dire, dont l'évaluation risque de créer de nouvelles cellules), et à refuser de généraliser le type de ces expressions. Cependant, rappelons que le système qui nous intéresse ici n'admet pas la notion de variable de types monomorphe ou non généralisée ; aussi cette formulation de la restriction de Wright ne nous convient-elle pas. Heureusement, ce premier problème se résout aisément ; il nous suffit d'adopter un point de vue équivalent, qui consiste à réécrire les constructions `let` expansives en de simples β-rédex. Ainsi, le programme ci-dessus devient

```
(fun x -> x := succ; !x true) (ref (fun x -> x))
```

Cette fois, les expressions `x := succ` et `!x true` sont typées dans un contexte où `x` est lié par λ, et engendrent donc la contrainte $\texttt{bool} \leq \texttt{int}$. Ainsi, le programme est correctement rejeté.

Reste le cas des constructions `let` à *toplevel*, c'est-à-dire celles qui servent à définir les différents composants d'un module. De façon concrète, un module se présente comme une série de déclarations :

$$\begin{array}{l}\texttt{let}\, X_1 = e_1\,\texttt{;;}\\ \dots\\ \texttt{let}\, X_n = e_n\,\texttt{;;}\end{array}$$

Pour obtenir la compilation séparée, ce groupe de déclarations doit être analysé isolément. Par conséquent, il est impossible, dans le cas où l'expression e_i est expansive, d'appliquer la méthode ci-dessus, qui consisterait à abstraire le reste du programme par rapport à la variable X_i.

Puisque notre système ne supporte pas la notion de variable non généralisée, et puisque le passage par une réécriture n'est plus possible ici, nous typons ces déclarations `let` de façon normale. On utilise donc l'algorithme d'inférence pour attribuer à chaque expression e_i un schéma de types σ_i, et la variable X_i se voit associer ce schéma. Cependant, il nous faut alors imposer une condition suffisante pour assurer la correction vis-à-vis de la sémantique. Une condition simple consiste à exiger que σ_i soit trivial à chaque fois que e_i est expansive. En effet, comme l'indique la proposition 12.3, un schéma trivial n'offre aucun polymorphisme. Le fait de le généraliser n'apporte aucune puissance supplémentaire, et n'est donc pas dangereux.

Cette façon de procéder est correcte, mais plus restrictive que celle utilisée, par exemple, par Objective Caml. Premièrement, le système de typage de ML permet à une construction `let` à *toplevel* de produire un schéma de types partiellement généralisé. Par exemple, si un module est constitué de la déclaration

```
let x = ref (fun x -> x);;
```

alors, le schéma de types associé à `x` est $(\alpha \rightarrow \alpha)$ `ref`, où α n'est pas quantifiée universellement. Deuxièmement, dans le cas de ML, la relation de comparaison entre schémas de types, utilisée pour comparer le schéma inféré au schéma déclaré, s'étend aisément au cas où le schéma de gauche contient des variables non quantifiées. On peut donc déclarer, dans l'interface :

```
val x: (int -> int) ref
```

Lors de la comparaison, la variable monomorphe α sera instanciée à la valeur `int`, et le module sera accepté avec cette signature. Notre système est moins flexible ; toute instantiation doit être effectuée dès la définition de la valeur, et non lors de la comparaison entre module et interface. Le module devra donc être écrit, par exemple,

```
let x = [ ref (fun x -> x) : (int -> int, int -> int) ref ];;
```

(La construction `[` e `:` τ `]` simule une utilisation de l'expression e avec le type τ. Si le schéma de types associé à e est $A \Rightarrow \tau_e \mid C$, cette construction engendre la contrainte supplémentaire $\tau_e \leq \tau$. Ici, cette contrainte permet d'associer un schéma de types trivial à `x`.) On pourra ensuite indiquer, dans la signature :

```
val x : (int -> int, int -> int) ref
```

Tout programme acceptable pour Objective Caml l'est également pour notre système, à condition d'ajouter suffisamment d'annotations explicites pour rendre monomorphe toute déclaration expansive. En pratique, ces indications de typage pourraient être automatiquement extraites de la signature et insérées dans le programme.

Cette restriction n'est imposée que par l'objectif de la compilation séparée. Dans le cas d'une boucle interactive, où l'utilisateur fournit, une par une, une série de déclarations `let` à *toplevel*, le problème est moins difficile, et on peut à nouveau utiliser la technique de réécriture des constructions `let` expansives en β-rédex. Cependant, l'intérêt en est relativement limité ; il ne paraît pas souhaitable d'offrir à l'utilisateur plus de flexibilité dans la boucle interactive que dans le système de programmation modulaire.

Pour résoudre ce problème, il semble souhaitable de revenir à un système dans lequel toutes les variables ne sont pas nécessairement quantifiées. Il est relativement facile de formuler un tel système, en abandonnant le λ-*lifting* et en se rapprochant de ML. Cependant, la règle de sous-typage obtenue est assez complexe. Nous présentons ce système à la section 16.4.

16.2 Inconvénients du λ-lifting

Le système étudié au cours de cette thèse utilise la technique du λ-*lifting*. Le principal avantage de cette décision réside dans le fait que tous les schémas de types manipulés sont alors clos, ce qui simplifie l'énoncé de la règle de sous-typage, duquel dépend ensuite l'essentiel de notre théorie. Cependant, elle n'est pas sans inconvénients.

Le plus immédiat de ces inconvénients réside dans le fait que le λ-*lifting* est un mécanisme peu classique et peu intuitif pour l'utilisateur, même s'il est fondé sur une idée relativement simple. Rappelons brièvement son fonctionnement, en considérant le traitement de l'expression $\lambda x.(x, x)$ par l'algorithme d'inférence.

Lorsque l'algorithme rencontre le lieur λx, il ajoute le nom x à l'environnement, mais sans lui associer de variable de types. Les deux occurrences de x sont donc typées entièrement indépendamment, et les schémas qu'elles obtiennent ne partagent aucune variable. Notons ces schémas $(\varnothing; x : \alpha_i) \Rightarrow \beta_i \mid \{\alpha_i \leq \beta_i\}$, pour $i \in \{1, 2\}$. Lorsqu'on type la paire (x, x), on doit, comme lorsqu'on type une application, réaliser l'intersection des contextes en provenance des deux branches. Le schéma attribué à cette paire sera donc

$$(\varnothing; x : \alpha_1 \sqcap \alpha_2) \Rightarrow \gamma \mid \{\alpha_1 \leq \beta_1,\, \alpha_2 \leq \beta_2,\, \beta_1 \times \beta_2 \leq \gamma\}$$

et l'expression complète se voit associer le schéma

$$(\alpha_1 \sqcap \alpha_2) \rightarrow \gamma \mid \{\alpha_1 \leq \beta_1,\, \alpha_2 \leq \beta_2,\, \beta_1 \times \beta_2 \leq \gamma\}$$

Si nous appliquons à présent l'algorithme de canonisation, nous obtenons

$$\alpha \to \gamma \mid \{\alpha \leq \alpha_1,\, \alpha \leq \beta_1,\, \alpha \leq \alpha_2,\, \alpha \leq \beta_2,\, \alpha_1 \leq \beta_1,\, \alpha_2 \leq \beta_2,\, \beta_1 \times \beta_2 \leq \gamma\}$$

Une passe de dépoussiérage donne

$$\alpha \to \gamma \mid \{\alpha \leq \beta_1,\, \alpha \leq \beta_2,\, \beta_1 \times \beta_2 \leq \gamma\}$$

Enfin, en exécutant l'algorithme de minimisation, on obtient un résultat optimal, à savoir

$$\alpha \to \gamma \mid \{\alpha \leq \beta,\, \beta \times \beta \leq \gamma\}$$

Ainsi, le système de typage ne tire pas profit de l'environnement pour partager l'information entre les différentes branches de la dérivation. Le partage est d'abord volontairement perdu, conduisant ainsi chaque branche à être typée isolément, puis explicitement rétabli par l'opération d'intersection des contextes. Dans un système sans λ-*lifting*, comme celui de ML, une variable fraîche α aurait été associée à x lors de l'entrée sous le lieur λx, et les deux occurrences de x auraient reçu cette même variable pour type. Le type optimal $\alpha \to \alpha \times \alpha$ aurait ainsi été obtenu sans qu'aucune simplification soit nécessaire.

Il est clair que l'utilisation du λ-*lifting* peut gêner la compréhension du système par le non-spécialiste, et pose donc un problème pédagogique. Mais cela va plus loin : l'exemple ci-dessus suggère qu'elle provoque également une certaine perte d'efficacité. En effet, il montre que les algorithmes de canonisation, de dépoussiérage et de minimisation passent une partie de leur temps à rétablir un partage qui a été volontairement perdu de par la formulation des règles de typage. De façon générale, considérons une expression de la forme

$$\lambda x.\mathtt{let}\; X = e_1 \;\mathtt{in}\; e_2$$

L'expression e_1 peut utiliser la variable x de façon complexe, par exemple en lui faisant subir un filtrage profond. Le schéma de types associé à e_1 sera donc de la forme $(\ldots; x : \alpha \ldots) \Rightarrow \ldots \mid (C \cup \ldots)$, où C contient un certain nombre de contraintes concernant α, définissant ainsi le type attendu pour x. Pendant le typage de e_2, chaque utilisation de la variable X provoque une duplication de ces contraintes ; puis, l'opération d'intersection des contextes rétablit le partage entre les différentes copies. Le schéma inféré pour e_2 sera ainsi de la forme $(\ldots; x : \alpha_1 \sqcap \cdots \sqcap \alpha_n \ldots) \Rightarrow \ldots \mid (C_1 \cup \cdots \cup C_n \cup \ldots)$, où n est le nombre d'occurrences de X dans e_2, et où les $\alpha_i \mid C_i$ sont des copies de $\alpha \mid C$. Encore une fois, des calculs de simplification sont nécessaires pour éliminer cette redondance.

Quelle solution apporter à ce problème ? Dans ce qui suit, nous considérons deux possibilités. La première, abordée au cours de la section 16.3, consiste à réduire l'impact de ces duplications inutiles en ajoutant au système un mécanisme de création automatique d'abréviations. L'idée est de représenter un couple $\alpha \mid C$ par une abréviation, c'est-à-dire un simple nom, éventuellement paramétré. Seule l'abréviation sera ensuite dupliquée, d'où, espère-t-on, un gain d'efficacité. La seconde possibilité, décrite par la section 16.4, consiste à abandonner le λ-*lifting* et à adopter une formulation plus classique du système de typage, déjà mentionnée dans la section précédente. La variable α sera alors monomorphe, donc ne sera pas dupliquée ; sous réserve d'effectuer une analyse à cet effet, il en ira de même des contraintes concernant α contenues dans le graphe C. Notons d'ailleurs que ces deux idées ne sont pas mutuellement exclusives ; la notion d'abréviation présente un intérêt, indépendamment du système sous-jacent.

16.3 Abréviations automatiques

La création automatique d'abréviations, utilisée par Rémy et Vouillon [47], consiste à représenter un morceau de structure par un simple nom. Par exemple, la paire $(0, 0)$ admet le schéma de types

$$\beta \mid \{\mathtt{int} \leq \alpha,\, \alpha \times \alpha \leq \beta\}$$

Ce schéma est trivial, au sens de la définition 12.3, c'est-à-dire que les variables qu'il contient n'introduisent aucun polymorphisme. Elles servent seulement, comme on l'a expliqué lors de l'introduction de l'invariant des petits termes, à étiqueter chaque nœud. Par conséquent, la duplication de ce schéma est inutile. Par exemple, lorsqu'on type l'expression

$$\texttt{let}\, X = (0,0)\, \texttt{in}\ (X, X)$$

chaque occurrence de X se voit associer une copie du schéma ci-dessus, et il est nécessaire d'utiliser l'algorithme de minimisation pour rétablir le partage, inutilement perdu, entre ces deux occurrences. Aussi, on peut imaginer de mettre en place le mécanisme suivant : avant d'introduire X dans l'environnement, on définit une abréviation

$$\texttt{intpair} = \forall\alpha \mid \{\texttt{int} \leq \alpha\}.\, \alpha \times \alpha$$

X se voit alors associer le schéma de types $\beta \mid \{\texttt{intpair} \leq \beta\}$ dans l'environnement. Ce schéma est dupliqué à chaque occurrence de X ; ainsi, on duplique la variable β, mais non la structure représentée par l'abréviation `intpair`. On a donc gagné en incrémentalité.

L'exemple ci-dessus est particulièrement simple, puisque le schéma de types considéré est trivial. Parce qu'il n'offre aucun polymorphisme, il peut être entièrement représenté par un simple nom, `intpair`. De façon plus générale, un schéma contient un certain degré de polymorphisme, et il sera nécessaire de créer des abréviations *paramétrées*. Considérons par exemple le schéma $(\varnothing; l : \alpha_1) \Rightarrow \alpha_9 \mid C$, où C est donné par

$$\begin{array}{rl} \alpha_1 & \leq \texttt{[Nil: } \alpha_2\texttt{; Cons: } \alpha_3\texttt{; } \alpha_4 \texttt{]} \\ \alpha_2 & \leq \texttt{Pre } \alpha_5 \\ \alpha_3 & \leq \texttt{Pre } \alpha_6 \\ \alpha_4 & \leq \texttt{Abs} \\ \alpha_5 & \leq \texttt{unit} \\ \alpha_6 & \leq \alpha_7 \times \alpha_1 \\ \alpha_7 & \leq \alpha_8 \\ \alpha_7 \leq \alpha_8 & \\ \alpha_8 \times \alpha_8 \leq \alpha_9 & \end{array}$$

(Ce schéma pourrait être celui d'une expression qui lit une liste dans la variable l, et construit une paire avec deux de ses éléments.) Ce schéma n'est pas trivial, car on y trouve une contrainte reliant deux variables, $\alpha_7 \leq \alpha_8$. Les autres variables servent uniquement à étiqueter des nœuds, et pourront être cachées par les abréviations. α_7 et α_8, au contraire, doivent rester visibles, car leur duplication est nécessaire pour éviter une perte de généralité ; elles apparaîtront donc comme paramètres de ces abréviations. Concrètement, nous allons créer ici deux abréviations, c'est-à-dire une pour chacun des points d'entrée du schéma. Posons

$$\begin{aligned} \alpha_7 \ \texttt{list} &= \exists\alpha_1 \ldots \alpha_6 \mid C_{1-6}.\, \texttt{[Nil: } \alpha_2\texttt{; Cons: } \alpha_3\texttt{; } \alpha_4 \texttt{]} \\ \alpha_8 \ \texttt{pair} &= \alpha_8 \times \alpha_8 \end{aligned}$$

(On désigne par C_{1-6} le graphe de contraintes constitué par les six premières lignes du graphe C.) Le schéma original peut alors s'écrire

$$(\varnothing; l : \alpha_1) \Rightarrow \alpha_9 \mid \{\alpha_1 \leq \alpha_7 \ \texttt{list}, \alpha_7 \leq \alpha_8, \alpha_8 \ \texttt{pair} \leq \alpha_9\}$$

(Si α_1 et α_9 apparaissent toujours dans ce schéma, c'est uniquement pour respecter l'invariant des petits termes. En principe, on aurait pu les cacher également.) Encore une fois, ce schéma peut à présent être dupliqué sans que le partage de sa structure interne soit perdu. Notons que l'abréviation `list` est destinée à être utilisée en position négative, d'où l'utilisation de quantificateurs existentiels dans sa définition. L'abréviation `pair` est destinée à être utilisée en position positive ; elle utilise donc, comme `intpair`, des quantificateurs universels. En fait, dans le cas de `pair`, aucun quantificateur n'apparaît, car aucune variable n'est cachée par `pair` ; on peut, en pratique, se dispenser d'introduire cette abréviation, puisqu'on n'en retire aucun gain.

Dans chaque cas, les seules variables libres dans le corps de la définition sont les paramètres de l'abréviation. Pour expanser une abréviation, on la remplace par son corps, en substituant les paramètres effectifs aux paramètres formels, et une variable fraîche à chaque variable liée dans la définition. De nombreuses opérations peuvent s'effectuer sans nécessiter l'expansion des abréviations : duplication de schémas, dépoussiérage, minimisation... L'algorithme de minimisation ne peut alors plus obtenir un partage optimal, mais il est utilisé de façon plus incrémentale. En effet, lorsque l'algorithme est appelé (c'est-à-dire à chaque nœud `let`, avant la phase de création des abréviations), la partie récente du graphe de contraintes est constituée de types concrets, nouvellement introduits par application des règles de typage, tandis que les parties plus anciennes sont cachées par des abréviations, considérées comme abstraites par l'algorithme. Certaines possibilités de partage entre parties anciennes ne pourront donc pas être découvertes ; en contrepartie, le temps nécessité par la phase de minimisation devrait être *grosso modo* proportionnel à la taille des parties récentes, d'où une meilleure incrémentalité. Par ailleurs, l'expansion des abréviations reste bien sûr nécessaire pendant le calcul de clôture. Par exemple, l'apparition d'une contrainte de la forme α `list` $\leq$ `[ Nil:` β`; Cons:` γ`;` ρ `]` nécessite l'expansion du terme de gauche, pour permettre la décomposition structurelle. L'algorithme de clôture étant incrémental, seules les abréviations atteintes par les contraintes récentes seront expansées.

Nous ne formaliserons pas ici ce mécanisme d'abréviation automatique. Pour résumer, il se compose de deux éléments : le générateur d'abréviations, appelé par exemple à chaque nœud `let`, et le mécanisme d'expansion, appelé à la demande par les algorithmes de clôture et de canonisation. Le premier engendre plusieurs abréviations par schéma, celles-ci étant délimitées par les contraintes entre variables, comme la contrainte $\alpha_7 \leq \alpha_8$ plus haut. Sa formalisation devrait être relativement simple. Le second est en principe très simple, mais son introduction brise les invariants des algorithmes de clôture et de canonisation ; aussi une description formelle sera-t-elle plus difficile.

Nous avons réalisé une implémentation de ce système d'abréviations et obtenu, dans la plupart des cas, un bénéfice proche de zéro. Comment expliquer cela ? Il est possible que les abréviations engendrées aient souvent un grand nombre de paramètres. En effet, la technique du λ-*lifting* nous conduit à abstraire chaque expression par rapport à tout l'environnement. L'expression apparaît donc comme très polymorphe, et l'abréviation associée comporte de nombreux paramètres. Dans le cadre du système « à la ML » que nous allons décrire ci-dessous, au contraire, les variables apparaissant dans l'environnement sont monomorphes et peuvent donc être cachées par l'abréviation. Il serait donc intéressant de mettre en œuvre le mécanisme d'abréviations dans ce système.

16.4 Un système de typage « à la ML »

Nous allons à présent exposer un système de typage dont la formulation est plus proche de celle de ML, et prouver sa correction. Toutefois, sa règle de sous-typage étant complexe, il n'est pas immédiat d'en déduire des algorithmes de comparaison et de simplification de jugements, comme nous l'avons fait pour notre système. C'est pourquoi nous nous sommes contentés, au cours de cette thèse, d'un système moins flexible, mais plus simple d'un point de vue théorique.

16.4.1 Présentation

Le nouveau système est donné par la figure 16.1 page ci-contre. Les jugements de typage y sont de la forme $C \Rightarrow \Gamma, A \vdash_{\mathrm{ML}} e : \tau$. Le graphe de contraintes C porte sur les variables libres de Γ, A et τ ; pour que le jugement soit considéré comme valide, C doit être soluble. L'environnement Γ associe à chaque variable liée par `let` un schéma de types de la forme $\forall\bar{\beta} \mid C.\,\tau$, tandis que l'environnement A associe à chaque variable liée par λ un type τ. (Ces

$$C \Rightarrow \Gamma, A \vdash_{\text{ML}} x : A(x) \qquad (\text{Var}_{\text{ML}})$$

$$\frac{C \Rightarrow \Gamma, (A; x : \tau) \vdash_{\text{ML}} e : \tau'}{C \Rightarrow \Gamma, A \vdash_{\text{ML}} \lambda x.e : \tau \to \tau'} \qquad (\text{Abs}_{\text{ML}})$$

$$\frac{C \Rightarrow \Gamma, A \vdash_{\text{ML}} e_1 : \tau_2 \to \tau \qquad C \Rightarrow \Gamma, A \vdash_{\text{ML}} e_2 : \tau_2}{C \Rightarrow \Gamma, A \vdash_{\text{ML}} e_1\, e_2 : \tau} \qquad (\text{App}_{\text{ML}})$$

$$\frac{\Gamma(X) = \forall\bar{\beta} \mid C.\,\tau \qquad \rho \text{ substitution de domaine } \bar{\beta}}{\rho(C) \Rightarrow \Gamma, A \vdash_{\text{ML}} X : \rho(\tau)} \qquad (\text{LetVar}_{\text{ML}})$$

$$\frac{\begin{array}{c} C_1 \Rightarrow \Gamma, A \vdash_{\text{ML}} e_1 : \tau_1 \qquad \bar{\beta} \cap \text{fv}(A) = \varnothing \\ C_2 \Rightarrow (\Gamma; X : \forall\bar{\beta} \mid C_1.\,\tau_1), A \vdash_{\text{ML}} e_2 : \tau_2 \end{array}}{C_2 \Rightarrow \Gamma, A \vdash_{\text{ML}} \texttt{let}\, X = e_1 \,\texttt{in}\; e_2 : \tau_2} \qquad (\text{Let}_{\text{ML}})$$

$$\frac{\begin{array}{c} C \Rightarrow \Gamma, A \vdash_{\text{ML}} e : \tau \\ \forall \rho' \vdash C' \quad \exists \rho \vdash C \quad \rho'(\Gamma') \leq^{\forall} \rho(\Gamma) \wedge \rho'(A') \leq \rho(A) \wedge \rho(\tau) \leq \rho'(\tau') \end{array}}{C' \Rightarrow \Gamma', A' \vdash_{\text{ML}} e : \tau'} \qquad (\text{Sub}_{\text{ML}})$$

Fig. 16.1: Règles de typage « à la ML »

deux environnements sont distingués ici pour mieux mettre en valeur leurs rôles respectifs, en particulier dans la règle de sous-typage ; cependant, cette distinction n'est pas indispensable.) Il est important de noter que les schémas de types ne sont plus nécessairement clos.

Quels sont les avantages espérés de ce système ? D'abord, sa présentation est plus simple et plus classique, ce qui est intéressant, ne serait-ce que du point de vue pédagogique. Ensuite, l'interaction entre polymorphisme et constructions impératives y est sensiblement plus facile à traiter. Troisièmement, le schéma de types associé à une expression donnée est moins polymorphe, puisqu'il n'est plus abstrait par rapport à l'environnement. Ainsi, dans le cadre de la création automatique d'abréviations, on peut espérer diminuer le nombre de paramètres des abréviations. Enfin, la construction `let` ne généralise pas les variables apparaissant libres dans l'environnement, et on élimine ainsi le phénomène de duplication/intersection des contextes mis en évidence par la section 16.2.

A ce propos, notons qu'une analyse très simple est nécessaire, à chaque nœud `let`, pour éviter des duplications inutiles. Elle ne dépend pas du sous-typage, et peut se formuler de façon identique dans le cadre de l'unification. Elle a été proposée et implémentée par Rémy [44, 46], puis légèrement améliorée par Birkedal *et al.* [11]. En voici le principe. Si on applique la règle (Let_{ML}) aveuglément, on généralise toutes les variables non libres dans l'environnement. Or, on l'a dit, certaines de ces variables ne servent qu'à étiqueter des nœuds, non à permettre le polymorphisme, et il est donc inutile de les dupliquer. Prenons l'exemple de l'expression

$$\lambda x.\texttt{let}\, X = (x, x) \,\texttt{in}\; (X, X)$$

Dans l'environnement $(\varnothing; x : \alpha)$, l'expression (x, x) admet le type β, sous la contrainte $\{\alpha \times \alpha \leq \beta\}$. Lorsqu'on introduit X dans l'environnement, on est en droit de généraliser β. On attribue alors à X le schéma $\forall\beta \mid \{\alpha \times \alpha \leq \beta\}.\,\beta$. Chaque occurrence de X se verra associer une instance différente de β. Or, c'est inutile ; α étant monomorphe, ces deux instances auront la même borne inférieure, à savoir $\alpha \times \alpha$, et seront identifiées par l'algorithme de minimisation. Il est donc plus efficace de ne pas généraliser β. De façon générale, l'ensemble des variables à généraliser se calcule par point fixe. Premièrement,

$$A \vdash_{\mathrm{B}} x : A(x) \qquad (\textsc{Var}_{\mathrm{B}})$$

$$\frac{A; x : \tau \vdash_{\mathrm{B}} e : \tau'}{A \vdash_{\mathrm{B}} \lambda x.e : \tau \to \tau'} \qquad (\textsc{Abs}_{\mathrm{B}})$$

$$\frac{A \vdash_{\mathrm{B}} e_1 : \tau_2 \to \tau \qquad A \vdash_{\mathrm{B}} e_2 : \tau_2}{A \vdash_{\mathrm{B}} e_1\, e_2 : \tau} \qquad (\textsc{App}_{\mathrm{B}})$$

$$\frac{A \vdash_{\mathrm{B}} e : \tau \quad A' \leq A \quad \tau \leq \tau'}{A' \vdash_{\mathrm{B}} e : \tau'} \qquad (\textsc{Sub}_{\mathrm{B}})$$

FIG. 16.2: Règles de typage « brutes »

toute variable « non triviale », c'est-à-dire portant un lien vers une autre variable, doit être généralisée. Deuxièmement, si la borne d'une variable α contient une variable à généraliser, alors α elle-même doit être généralisée. Cette analyse est intéressante, puisqu'elle permet d'éviter une certaine duplication, à faible coût. Notons toutefois qu'elle est moins puissante que le mécanisme d'abréviations automatiques décrit précédemment. En effet, ici, si un terme contient une feuille polymorphe, alors tout le chemin qui relie la racine du terme à cette feuille sera dupliqué. Au contraire, si on introduit une abréviation, cette feuille en devient un paramètre, et le chemin reste interne à l'abréviation ; ainsi, seule la feuille sera dupliquée. L'introduction d'abréviations a donc toujours un sens dans ce nouveau système.

16.4.2 Correction

Avant de nous intéresser aux problèmes posés par le système $\vdash_{\mathrm{ML}}$, il convient de vérifier que celui-ci est correct vis-à-vis de la sémantique. Cela est nécessaire pour nous assurer que les règles données par la figure 16.1 sont formulées de façon sensée, et que leur étude a un intérêt.

Nous nous contenterons ici de donner les grandes lignes de la preuve de correction. La méthode proposée ici est différente de celle adoptée au chapitre 5. Nous aurions pu procéder de façon similaire, en affaiblissant d'abord la règle ($\textsc{Sub}_{\mathrm{ML}}$) pour obtenir un système plus simple, mais complet par rapport au système initial, puis en prouvant la stabilité du typage par réduction dans ce second système. L'approche utilisée ici fournit des résultats moins satisfaisants : en particulier, elle prouve seulement la correction du système, non la stabilité du typage par réduction. Cependant, elle est fondée sur une intuition intéressante, qui nous semble justifier son adoption ici.

Le principe de la preuve consiste à démontrer que si un jugement de typage est vérifié dans le système $\vdash_{\mathrm{ML}}$, alors toute instance brute de ce jugement est également valide. Par instance brute, on entend ici un jugement de typage obtenu par substitution à partir du jugement initial, et énoncé dans un système où n'interviennent que des types bruts. Ce second système, extrêmement simple, étant correct, il suffira alors de vérifier que tout jugement admet au moins une instance brute pour obtenir la correction du système $\vdash_{\mathrm{ML}}$.

La figure 16.2 définit ce système de typage auxiliaire, que nous qualifierons de « brut ». Les types mis en jeu par ces règles sont exclusivement des types bruts, dénués de toute variable de types ; la règle ($\textsc{Sub}_{\mathrm{B}}$) utilise donc directement la relation de sous-typage $\leq$. Il est aisé de vérifier que le typage y est préservé par réduction. Bien entendu, ce système n'a pas de notion de polymorphisme, et ne supporte donc pas la construction `let`. Celle-ci devra donc être explicitement expansée par notre preuve.

Définition 16.1 *L'opération de* `let`-expansion *est définie par*

$$\begin{aligned} \mathrm{LE}(x) &= x \\ \mathrm{LE}(\lambda x.e) &= \lambda x.\mathrm{LE}(e) \\ \mathrm{LE}(e_1\, e_2) &= \mathrm{LE}(e_1)\,\mathrm{LE}(e_2) \\ \mathrm{LE}(\texttt{let}\, X = e_1 \texttt{ in } e_2) &= [\mathrm{LE}(e_1)/X]\,\mathrm{LE}(e_2) \end{aligned}$$

Nous pouvons maintenant énoncer le lemme principal.

Lemme 16.1 *On suppose $C \Rightarrow \Gamma, A \vdash_{\mathrm{ML}} e : \tau$. Soit ρ une solution de C. On suppose donnés un environnement A_0 et une substitution g, de domaine* $\mathrm{dom}(\Gamma)$*, vérifiant les conditions suivantes :*

- $A_0 \leq \rho(A)$;
- $\forall X \in \mathrm{dom}(\Gamma) \quad \forall \theta \quad (\mathrm{dom}(\theta) = \bar{\beta}_X \wedge \rho\theta \vdash C_X) \Rightarrow A_0 \vdash_{\mathrm{B}} g(X) : \rho\theta(\tau_X)$.

(On note ici $\Gamma(X) = \forall \bar{\beta}_X \mid C_X . \tau_X$, et θ dénote une substitution brute.) Alors

$$A_0 \vdash_{\mathrm{B}} g(\mathrm{LE}(e)) : \rho(\tau)$$

Démonstration. Par induction sur la dérivation de $C \Rightarrow \Gamma, A \vdash_{\mathrm{ML}} e : \tau$. La preuve ne présente pas de difficulté particulière, aussi nous l'omettrons par souci de brièveté. La seconde condition ci-dessus exprime le fait que la substitution g constitue une implémentation satisfaisante de l'environnement Γ ; en effet, chaque expression $g(X)$ admet tous les types bruts obtenus par instantiation du schéma $\rho(\Gamma(X))$. L'introduction du paramètre A_0, et de la première condition, est rendue nécessaire par la présence de la règle de sous-typage. Si l'on impose $A_0 = \rho(A)$, l'énoncé reste bien sûr correct, puisque nous en avons pris un cas particulier, mais la preuve par induction n'est plus possible. □

En particulier, on obtient en corollaire le théorème suivant :

Théorème 16.1 *On suppose $C \Rightarrow \varnothing, \varnothing \vdash_{\mathrm{ML}} e : \tau$. Soit ρ une solution de C. Alors*

$$\varnothing \vdash_{\mathrm{B}} \mathrm{LE}(e) : \rho(\tau)$$

Démonstration. Immédiate, en choisissant pour A_0 l'environnement vide et pour g la substitution de domaine vide. □

Le système $\vdash_{\mathrm{B}}$ étant correct, il découle de ce résultat que, si $C \Rightarrow \varnothing, \varnothing \vdash_{\mathrm{ML}} e : \tau$ est prouvable et si C admet une solution, alors l'expression $\mathrm{LE}(e)$ ne provoque pas d'erreur d'exécution. En d'autres termes, pour peu que e et $\mathrm{LE}(e)$ aient la même sémantique, le système $\vdash_{\mathrm{ML}}$ est correct.

On notera que, étant donnée notre définition de la `let`-expansion, on doit choisir une sémantique d'appel par nom pour que e et $\mathrm{LE}(e)$ aient la même sémantique. Si on préfère utiliser une sémantique d'appel par valeur, on pourra définir

$$\mathrm{LE}(\texttt{let}\, X = e_1 \texttt{ in } e_2) = [\mathrm{LE}(e_1)/X]\,((\lambda\_.\mathrm{LE}(e_2))\, X)$$

Pour préserver le lemme 16.1, il faudra alors ajouter à la règle ($\mathrm{L{\scriptstyle ET}_{ML}}$) la prémisse $C_2 \Vdash C_1$. On prend ainsi en compte l'effet sur l'environnement des contraintes en provenance de l'expression e_1. (Le même choix existe dans notre système, et a été mentionné dans la section 5.3.)

Pour conclure, le résultat obtenu ici montre que notre formulation du système $\vdash_{\mathrm{ML}}$ est satisfaisante. La méthode de preuve utilisée ici présente des inconvénients : en particulier, nous n'avons pas montré la stabilité du typage par réduction, et l'utilisation de la `let`-expansion empêche l'addition de traits impératifs. Néanmoins, il nous a semblé intéressant de développer le lien avec un système de typage « brut », plutôt que de répéter le processus de preuve utilisé au chapitre 5.

Notons, à ce propos, l'existence d'une troisième méthode de preuve. On aurait pu tenter de démontrer l'équivalence entre les jugements $\varnothing \vdash e : \varnothing \Rightarrow \tau \mid C$ et $C \Rightarrow \varnothing, \varnothing \vdash_{\text{ML}} e : \tau$. Ce résultat indique que les deux systèmes acceptent les mêmes programmes, et permet de transporter le théorème de stabilité du typage du système original au système $\vdash_{\text{ML}}$. Nous n'avons pas étudié de près cette possibilité, mais elle semble également viable.

16.4.3 Commentaire

Toutes les règles de la figure 16.1 sont classiques, à l'exception de la règle ($\textsc{Sub}_{\text{ML}}$), qui mérite un commentaire approfondi. De façon générale, une règle de sous-typage permet d'affaiblir un jugement de typage, en le remplaçant par un jugement moins précis. Elle découle donc directement d'une relation de comparaison entre jugements. Dans le cas du système exposé au chapitre 5, les jugements sont de la forme $\Gamma \vdash e : \sigma$. La règle ($\textsc{Sub}$) n'autorise pas de modification à l'environnement Γ ; aussi découle-t-elle directement de la relation de comparaison entre schémas de types. Elle peut s'écrire

$$\frac{\begin{array}{c}\Gamma \vdash e : A \Rightarrow \tau \mid C \\ \forall \rho' \vdash C' \quad \exists \rho \vdash C \quad \rho'(A') \leq \rho(A) \wedge \rho(\tau) \leq \rho'(\tau')\end{array}}{\Gamma \vdash e : A' \Rightarrow \tau' \mid C'} \; (\textsc{Sub})$$

La règle ($\textsc{Sub}_{\text{ML}}$) présentée ici est donc une généralisation de celle-ci au cas où l'environnement contient des schémas de types non clos. La différence vis-à-vis de la règle ($\textsc{Sub}$) est l'apparition de la condition $\rho'(\Gamma') \leq^{\forall} \rho(\Gamma)$. Il s'agit d'une comparaison point par point des deux environnements. Puisque ceux-ci contiennent des schémas de types, il faut utiliser une relation de comparaison entre schémas, et non la simple relation de sous-typage. Ainsi, si l'on note $\Gamma(X) = \forall \bar{\beta}_X \mid C_X . \tau_X$ et $\Gamma'(X) = \forall \bar{\beta}'_X \mid C'_X . \tau'_X$, alors la condition $\rho'(\Gamma') \leq^{\forall} \rho(\Gamma)$ s'écrit

$$\begin{array}{lll}\forall X \in \text{dom}(\Gamma) & \forall \theta & (\text{dom}(\theta) = \bar{\beta}_X \wedge \rho\theta \vdash C_X) \quad \Rightarrow \\ & \exists \theta' & (\text{dom}(\theta') = \bar{\beta}'_X \wedge \rho'\theta' \vdash C'_X \wedge \rho'\theta'(\tau'_X) \leq \rho\theta(\tau_X))\end{array}$$

La seconde prémisse de la règle ($\textsc{Sub}_{\text{ML}}$) est donc une assertion logique quantifiée par $\forall\exists\forall\exists$, tandis que notre règle ($\textsc{Sub}$) nécessitait seulement une quantification par $\forall\exists$. On voit donc apparaître un nouveau degré de complexité.

On nous objectera peut-être que la règle ($\textsc{Sub}_{\text{ML}}$) autorise le remplacement de l'environnement Γ par un nouvel environnement Γ', possibilité qui n'était pas offerte par la règle ($\textsc{Sub}$) ; que c'est là la cause du problème, et qu'il suffit de retirer cette possibilité pour le résoudre. Répondons à cette objection par trois remarques.

Premièrement, si la règle ($\textsc{Sub}$) n'autorise pas la modification de l'environnement Γ, c'est parce que celle-ci n'est nullement nécessaire en pratique. En effet, lorsqu'on applique la règle ($\textsc{Let}$) pour introduire un nouveau schéma de types dans l'environnement, on peut d'abord simplifier celui-ci par une application de la règle ($\textsc{Sub}$). L'environnement contiendra donc, à tout instant, des schémas de types déjà simplifiés, et il ne sera pas nécessaire de le modifier. Dans le cas du système $\vdash_{\text{ML}}$, on peut également simplifier chaque schéma au moment de son entrée dans l'environnement. Cependant, les schémas de types sont cette fois *ouverts* ; par conséquent, l'apparition de nouvelles contraintes peut engendrer de nouvelles possibilités de simplification de l'environnement. On peut donc souhaiter que la règle de sous-typage autorise la modification de l'environnement.

Deuxièmement, même si l'on se contente d'une règle de sous-typage incapable de modifier l'environnement, restreindre la règle ($\textsc{Sub}_{\text{ML}}$) au cas où $\Gamma = \Gamma'$ ne suffit pas à résoudre le problème, car la condition $\rho'(\Gamma') \leq^{\forall} \rho(\Gamma)$ ne devient pas triviale pour autant. En effet, celle-ci s'écrit alors

$$\forall X \in \text{dom}(\Gamma) \quad \rho'(\Gamma(X)) \leq^{\forall} \rho(\Gamma(X))$$

Si le schéma $\Gamma(X)$ contient des variables libres, alors $\rho'(\Gamma(X))$ n'est pas nécessairement égal à $\rho(\Gamma(X))$. Pour garantir l'égalité, on peut ajouter, pour chaque variable $\alpha \in \mathrm{fv}(\Gamma)$, la condition $\rho(\alpha) = \rho'(\alpha)$. La règle de sous-typage ainsi obtenue fait alors appel à une assertion du même type que celle figurant dans la règle (Sub). Le calcul des polarités et le dépoussiérage s'effectuent donc comme dans notre système, à la différence près que toute variable $\alpha \in \mathrm{fv}(\Gamma)$ reçoit le signe $\pm$. Ainsi, cette restriction conduit à des algorithmes de simplification proches de ceux présentés dans cette thèse, et aisés à implémenter. Cependant, le fait que toutes les variables libres dans l'environnement soient obligatoirement bipolaires est gênant. D'une part, cela brise l'invariant de mono-polarité, dont nous perdons les bénéfices. Par ailleurs, cela rend le dépoussiérage plus grossier, puisque la direction du flot de données n'est pas entièrement prise en compte ; on se rapproche ainsi d'une simple élimination des variables inaccessibles. Si cette solution vaut la peine d'être mentionnée au passage pour sa simplicité, elle n'est donc pas satisfaisante.

Troisièmement, la complexité de la règle (Sub$_{\mathrm{ML}}$) se retrouve, pour partie, dans le problème de la comparaison entre module et interface. Rappelons qu'il s'agit de comparer un schéma de types inféré à celui déclaré par l'utilisateur dans la signature. Dans le système étudié au cours de cette thèse, tous les schémas sont clos ; aussi le problème est-il identique à celui posé par la règle (Sub). Dans le système $\vdash_{\mathrm{ML}}$, le schéma figurant dans la signature est nécessairement clos, pour garantir la sûreté du système de compilation séparée ; cependant, le schéma inféré ne l'est pas. Le problème se présente donc sous la forme d'une assertion $\exists\forall\exists$, *a priori* moins complexe que l'assertion $\forall\exists\forall\exists$ de la règle (Sub$_{\mathrm{ML}}$), mais déjà plus difficile que le problème $\forall\exists$ étudié au chapitre 9. Il s'agit donc là d'un argument supplémentaire en faveur de l'étude du problème général. En cas de succès, l'algorithme obtenu sera utile non seulement du point de vue théorique, pour une meilleure compréhension de la règle (Sub$_{\mathrm{ML}}$) et pour la mise au point d'algorithmes de simplification, mais également du point de vue pratique, pour effectuer la comparaison entre modules et interfaces.

Ainsi, si le système $\vdash_{\mathrm{ML}}$ se formule de façon très naturelle, évitant en particulier l'emploi du λ-*lifting* et utilisant un mécanisme de généralisation et d'instantiation identique à celui de ML, il exige l'étude d'assertions plus complexes, et c'est pourquoi nous ne l'avons pas adopté. Rappelons que les algorithmes de simplification des jugements inférés, dans un système à base de sous-typage, découlent directement de la forme de la règle de sous-typage. Ainsi, au chapitre 9, nous avons étudié un algorithme (incomplet) de comparaison entre jugements de typage. De cet algorithme, on déduisait ensuite, de façon fort naturelle, les notions de polarité et de dépoussiérage. Or, il n'est pas immédiat de généraliser cet algorithme au problème de la comparaison entre jugements dans le système $\vdash_{\mathrm{ML}}$.

Rappelons que le problème est de décider une assertion de la forme suivante (on utilise ici, pour plus de légèreté, une notation quelque peu informelle) :

$$\forall\bar{\alpha}' \quad \exists\bar{\alpha} \quad \forall\bar{\beta}' \vdash C' \quad \exists\bar{\beta} \vdash C + (\beta \leq \beta')$$

(Dans le cas de la comparaison entre module et interface, on a $\bar{\alpha}' = \varnothing$, et le problème se simplifie légèrement.) Voici une ébauche d'algorithme. On affaiblit d'abord l'assertion en intervertissant les deux quantificateurs centraux :

$$\forall\bar{\alpha}'\bar{\beta}' \vdash C' \quad \exists\bar{\alpha}\bar{\beta} \vdash C + (\beta \leq \beta')$$

On peut appliquer à cette assertion simplifiée l'algorithme développé au chapitre 9. S'il réussit, alors il fournit un graphe de contraintes décrivant la valeur des $\bar{\alpha}\bar{\beta}$ en fonction de la valeur des $\bar{\alpha}'\bar{\beta}'$. On cherche alors, en renforçant éventuellement les contraintes présentes dans ce graphe, à éliminer les dépendances entre $\bar{\alpha}$ et $\bar{\beta}'$, c'est-à-dire à exprimer les $\bar{\alpha}$ en fonction des $\bar{\alpha}'$ uniquement. Pour cela, une approche possible consiste à calculer les valeurs extrêmes de chaque β', en fonction des $\bar{\alpha}'$, lorsque les $\bar{\beta}'$ varient. Une dépendance indésirable envers une variable β' peut ensuite être éliminée en remplaçant celle-ci par la borne ainsi calculée.

Cette description est bien sûr informelle et incomplète. Tout reste à faire : éclaircir les détails techniques, prouver la correction, déterminer si l'algorithme ainsi obtenu est d'une puissance suffisante en pratique, enfin en déduire les notions de polarité et de dépoussiérage associées. Nous n'avons pas suffisamment étudié ce problème pour pouvoir en dire plus ici, mais il s'agit probablement là d'une piste de recherche intéressante.

Conclusion

PARVENUS AU TERME DE CE TRAVAIL, permettons-nous de récapituler le chemin parcouru au cours de ces quelques années, en mentionnant au passage quelques détours qui n'apparaissent pas dans le présent ouvrage.

Notre point de départ est l'algorithme d'inférence de types publié par Eifrig, Smith et Trifonov [16]. On y trouve le principe de l'analyse à base de contraintes, ainsi que l'algorithme de clôture. Ce système accepte d'ores et déjà le même ensemble de programmes que celui présenté dans cette thèse ; cependant, sa règle de sous-typage est extrêmement rudimentaire, car basée sur une simple inclusion entre ensembles de contraintes ; elle n'autorise donc aucune forme de simplification.

Notre premier objectif a été d'augmenter la puissance de la règle de sous-typage, ce que nous avons fait à l'aide de la relation d'implication de contraintes. Dans [16], la correction du système de typage est prouvée de façon entièrement syntaxique, en se basant uniquement sur la définition de la clôture, sans avoir recours à la notion de solution d'un ensemble de contraintes. Nous avons suivi cette voie un certain temps, et notre définition initiale de l'implication de contraintes était également purement syntaxique. Au lieu d'envisager toutes les solutions de l'ensemble d'hypothèses, elle considérait tous les contextes possibles, un contexte étant lui-même un ensemble de contraintes. On obtenait ainsi une définition plus « observationnelle » de l'implication de contraintes. Cette méthode donnait cependant lieu à des preuves très lourdes ; aussi avons-nous préféré introduire le treillis des types bruts, et la notion de solution, pour simplifier la théorie. On trouvera dans [43] un exposé des deux approches ; seule la seconde apparaît dans cette thèse.

Munis de la notion d'implication de contraintes, nous avons augmenté la puissance de la règle de sous-typage, et utilisé celle-ci pour justifier certaines simplifications à base de substitutions. Une substitution était dite valide si on pouvait revenir au jugement initial, à partir du jugement simplifié, par application de la règle de la sous-typage. En pratique, nous utilisions alors une heuristique pour proposer des substitutions plausibles, et notre axiomatisation de l'implication de contraintes pour vérifier leur validité. Un effort considérable a été consacré à la recherche d'un algorithme de décision de l'implication, par nous et par plusieurs autres équipes, sans succès à ce jour. Du point de vue théorique, cela reste un problème ouvert intéressant.

A la même époque, nous avons proposé un algorithme d'élimination des contraintes inaccessibles, de conception fort simple. Cependant, du point de vue théorique, sa correction ne pouvait être prouvée par un simple appel à la règle de sous-typage ; un méta-théorème, fondé sur une réécriture des dérivations de typage, était nécessaire. Trifonov et Smith [49] éliminent ce problème en proposant une version plus générale de la règle de sous-typage, basée sur la comparaison polymorphe de schémas, qui est celle que l'on trouve dans cette thèse. En même temps, ils améliorent notre algorithme en introduisant la notion de polarité, obtenant ainsi l'algorithme de dépoussiérage.

Le problème de la simplification n'est cependant pas encore résolu. D'une part, l'utilisation d'heuristiques pour le choix des substitutions est extrêmement coûteux. La mise au point d'une heuristique demande un pénible compromis entre coût et bénéfice escomptés ; de plus, la distinction entre les objectifs d'efficacité et de lisibilité n'étant pas clairement comprise à l'époque, certaines heuristiques destinées à améliorer la lisibilité provoquent une perte d'efficacité. D'autre part, les différents composants du système sont encore mal intégrés ; en particulier, canonisation et dépoussiérage produisent des ensembles de contraintes non clos, nécessitant ainsi de nouvelles passes de clôture.

Nous résolvons ici le premier de ces problèmes en introduisant l'algorithme de minimisation, accompagné de l'invariant des petits termes. Toute heuristique est donc éliminée, au profit d'un algorithme efficace, capable d'effectuer d'un seul coup une substitution quasi-optimale. L'adoption de l'invariant des petits termes, nécessaire au bon fonctionnement de l'algorithme, privilégie l'objectif d'efficacité tout au long des calculs. L'objectif de lisibilité est reconnu comme secondaire, et la phase de simplification « externe » ne sera effectuée qu'immédiatement avant l'affichage.

Le second de ces problèmes, c'est-à-dire celui de l'intégration entre les différents algorithmes, a également été éliminé. D'abord, nous donnons une description précise de l'algorithme de canonisation, qui produit un graphe de contraintes (simplement) clos, et nous l'optimisons en le combinant avec une phase de dépoussiérage partiel. Ensuite, nous proposons de modifier les règles d'inférence pour respecter l'invariant de mono-polarité, ce qui est probablement la façon la plus simple de s'assurer que l'algorithme de dépoussiérage produit un graphe clos.

Nous pensons que le système d'inférence et de simplification ainsi obtenu représente un tout homogène et bien compris. D'un point de vue pratique, ce système est simple. Il est constitué d'un petit nombre de composants : moteur de génération des contraintes (basé sur les règles d'inférence), algorithme de clôture incrémentale, algorithmes de canonisation, de dépoussiérage et de minimisation. L'implémentation de ces algorithmes correspond exactement à la description – et à la preuve – fournies dans cette théorie, et leur intégration est parfaite, puisque chacun produit une donnée acceptable par son successeur. Dans le cadre de la compilation séparée, la comparaison entre modules et interfaces nécessite l'algorithme de comparaison de schémas proposé par Trifonov et Smith, dont nous avons fourni une preuve détaillée.

Cependant, tout n'est pas satisfaisant. D'une part, le typage des constructions impératives est effectué de façon légèrement détournée, en particulier dans le cadre de la compilation séparée. D'autre part, les performances obtenues, bien que largement supérieures à ce qui était possible au début de ce travail, ne sont pas tout à fait suffisantes pour traiter de programmes de grande taille. L'étude approfondie d'un système de typage « à la ML » pourrait contribuer à résoudre ces deux problèmes ; de plus, elle semble poser des problèmes théoriques intéressants. C'est, par conséquent, un sujet auquel nous espérons continuer à nous intéresser. Par ailleurs, on peut imaginer d'autres solutions au problème d'efficacité, parmi lesquelles l'utilisation d'abréviations, ou la mise au point de versions incrémentales de nos algorithmes. Ce domaine reste donc riche en pistes à explorer.

Bibliographie

Des URLs ont été fournies dans la mesure du possible. Elles datent de fin 1997. Une version électronique de cette bibliographie devrait être disponible «en ligne» à l'adresse http://pauillac.inria.fr/~fpottier/biblio/these-en.html.

[1] Martín Abadi et Luca Cardelli. A theory of primitive objects — untyped and first-order systems. In Masami Hagiya et John C. Mitchell (rédacteurs), *Theoretical Aspects of Computer Software*, *Lecture Notes in Computer Science*, volume 789, pp. 296–320. Springer-Verlag, avril 1994. URL: http://research.microsoft.com/research/cambridge/luca/Papers/PrimObj1stOrder.A4.ps.

[2] Martín Abadi et Luca Cardelli. A theory of primitive objects — second-order systems. In D. Sannella (rédacteur), *Proc. of European Symposium on Programming*, *Lecture Notes in Computer Science*, volume 788, pp. 1–25, New York, N.Y., 1994. Springer Verlag. URL: http://research.microsoft.com/research/cambridge/luca/Papers/PrimObj2ndOrder.A4.ps.

[3] Alexander S. Aiken. The Illyria system. URL: http://http.cs.berkeley.edu:80/~aiken/Illyria-demo.html.

[4] Alexander S. Aiken et Manuel Fähndrich. Making set-constraint based program analyses scale. Rapport technique CSD-96-917, University of California, Berkeley, septembre 1996. URL: http://http.cs.berkeley.edu/~manuel/papers/scw96.ps.gz.

[5] Alexander S. Aiken et Edward L. Wimmers. Solving systems of set constraints. In Andre Scedrov (rédacteur), *Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science*, pp. 329–340, Santa Cruz, CA, juin 1992. IEEE Computer Society Press. URL: http://http.cs.berkeley.edu/~aiken/ftp/lics92.ps.

[6] Alexander S. Aiken et Edward L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming & Computer Architecture*, pp. 31–41. ACM Press, juin 1993. URL: http://http.cs.berkeley.edu/~aiken/ftp/fpca93.ps.

[7] Alexander S. Aiken, Edward L. Wimmers, et T. K. Lakshman. Soft typing with conditional types. In *Principles of Programming Languages*, pp. 163–173, janvier 1994. URL: http://http.cs.berkeley.edu/~aiken/ftp/popl94.ps.

[8] Alexander S. Aiken, Edward L. Wimmers, et Jens Palsberg. Optimal representations of polymorphic types with subtyping. Rapport technique CSD-96-909, University of California, Berkeley, juillet 1996. URL: http://http.cs.berkeley.edu/~aiken/ftp/quant.ps.

[9] Roberto M. Amadio et Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, septembre 1993. URL: http://research.microsoft.com/research/cambridge/luca/Papers/SRT.A4.ps.

[10] André Arnold et Maurice Nivat. The metric space of infinite trees. Algebraic and topological properties. *Fundamenta Informaticæ*, 3(4):181–205, 1980.

[11] Lars Birkedal, Nick Rothwell, Mads Tofte, et David N. Turner. The ML kit (version 1). Rapport technique DIKU 93/14, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, 1993. URL: http://www.diku.dk/research-groups/topps/activities/kit2/.

[12] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, février/mars 1988. Nouvelle version de l'article paru dans 1984 Semantics of Data Types Symposium, LNCS 173, pp. 51–66. URL: `http://research.microsoft.com/research/cambridge/luca/Papers/Inheritance.A4.ps`.

[13] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, et Gilles Kahn. A simple applicative language: Mini-ML. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pp. 13–27. ACM, ACM, août 1986.

[14] Bruno Courcelle. Fundamental properties of infinite trees. *Theoret. Comput. Sci.*, 25(2):95–169, mars 1983.

[15] Jonathan Eifrig, Scott Smith, et Valery Trifonov. Sound polymorphic type inference for objects. In *OOPSLA '95 Conference Proceedings*, *ACM SIGPLAN Notices*, volume 30(10), pp. 169–184, 1995. URL: `http://www.cs.jhu.edu/~trifonov/papers/sptio.ps.gz`.

[16] Jonathan Eifrig, Scott Smith, et Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Mathematical Foundations of Programming Semantics, New Orleans*, *Electronic Notes in Theoretical Computer Science*, volume 1. Elsevier, 1995. URL: `http://www.elsevier.nl/locate/entcs/volume1.html`.

[17] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, et Alexander S. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the 1998 Conference on Programming Languages Design and Implementation*, Montréal, juin 1998. A paraître. URL: `http://www.cs.berkeley.edu/~manuel/papers/pldi98.ps`.

[18] Cormac Flanagan. *Effective Static Debugging via Componential Set-Based Analysis.* Thèse de doctorat, Rice University, mai 1997. URL: `http://www.cs.rice.edu/CS/PLT/Publications/thesis-flanagan.ps.gz`.

[19] Cormac Flanagan et Matthias Felleisen. Modular and polymorphic set-based analysis: Theory and practice. Rapport technique TR96-266, Rice University, novembre 1996. URL: `http://www.cs.rice.edu/CS/PLT/Publications/tr96-266.ps.gz`.

[20] Cormac Flanagan et Matthias Felleisen. Componential set-based analysis. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pp. 235–248, Las Vegas, Nevada, juin 1997. URL: `http://www.cs.rice.edu/CS/PLT/Publications/pldi97-ff.ps.gz`.

[21] Alexandre Frey. Satisfying subtype inequalities in polynomial space. In Pascal Van Hentenryck (rédacteur), *Proceedings of the Forth International Symposium on Static Analysis (SAS'97)*, numéro 1302 in Lecture Notes in Computer Science, pp. 265–277, Paris, France, septembre 1997. Springer Verlag. URL: `http://www.ensmp.fr/~frey/Publications/SAS97.ps.gz`.

[22] You-Chin Fuh et Prateek Mishra. Type inference with subtypes. In H. Ganzinger (rédacteur), *Proceedings of the European Symposium on Programming*, *Lecture Notes in Computer Science*, volume 300, pp. 94–114. Springer Verlag, 1988.

[23] You-Chin Fuh et Prateek Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In J. Díaz et F. Orejas (rédacteurs), *Proceedings of the International Joint Conference on Theory and Practice of Software Development : Vol. 2*, *LNCS*, volume 352, pp. 167–183, Berlin, mars 1989. Springer.

[24] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Thèse de doctorat, Université Paris VII, juin 1972.

[25] Juan Carlos Guzmán et Ascánder Suárez. An extended type system for exceptions. In *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications*, numéro 2265 in INRIA Research Reports, pp. 127–135. INRIA, BP 105, 78153 Le Chesnay Cedex, France, juin 1994. URL: `http://www.ldc.usb.ve/~suarez/PAPERS/except.ps`.

[26] Nevin Heintze. Set based analysis of ML programs. Rapport technique CMU-CS-93-193, Carnegie Mellon University, School of Computer Science, juillet 1993. URL: `ftp://reports.adm.cs.cmu.edu/usr/anon/1993/CMU-CS-93-193.ps`.

[27] Fritz Henglein. Breaking through the n^3 barrier: Faster object type inference. In Benjamin Pierce (rédacteur), *Proc. 4th Int'l Workshop on Foundations of Object-Oriented Languages (FOOL), Paris, France*, janvier 1997. URL: `http://www.cs.indiana.edu/hyplan/pierce/fool/henglein.ps.gz`.

[28] Fritz Henglein et Jakob Rehof. Constraint automata and the complexity of recursive subtype entailment. In *25th International Colloquium on Automata, Languages, and Programming (ICALP'98)*, juillet 1998. A paraître.

[29] My Hoang et John C. Mitchell. Lower bounds on type inference with subtypes. In *Proceedings of the 22nd Symposium on Principles of Programming Languages (POPL'95)*, pp. 176–185, New York, NY, USA, janvier 1995. ACM Press.

[30] John E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi (rédacteur), *Theory of Machines and Computations*, pp. 189–196. Academic Press, NY, 1971.

[31] Gérard Huet. *Résolution d'équations dans des langages d'ordre* 1, 2, ..., ω. Thèse de doctorat, Université Paris VII, septembre 1976.

[32] Deepak Kapur et Hantao Zhang. An overview of Rewrite Rule Laboratory (RRL). *J. Comput. Appl. Math.*, 29(2):91–114, 1995. URL: `ftp://ftp.cs.albany.edu/pub/ipl/papers/overview.rrl.ps.gz`.

[33] Dexter Kozen, Jens Palsberg, et Michael I. Schwartzbach. Efficient recursive subtyping. In *Proceedings POPL '93*, pp. 419–428, 1993. URL: `ftp://ftp.daimi.aau.dk/pub/palsberg/papers/popl93.ps.Z`.

[34] Xavier Leroy. *Typage polymorphe d'un langage algorithmique*. Thèse de doctorat, Université Paris VII, juin 1992. URL: `http://pauillac.inria.fr/~xleroy/publi/these-doctorat.ps.gz`.

[35] David B. MacQueen, Gordon D. Plotkin, et Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1–2):95–130, octobre–novembre 1986.

[36] John C. Mitchell. Coercion and type inference. In *11th Annual ACM Symposium on Principles of Programming Languages*, pp. 175–185, janvier 1984.

[37] Joachim Niehren, Martin Müller, et Andreas Podelski. Inclusion constraints over non-empty sets of trees. In M. Dauchet (rédacteur), *Theory and Practice of Software Development, International Joint Conference CAAP/FASE/TOOLS*, *Lecture Notes in Computer Science*, volume 1214. Springer-Verlag, avril 1997. URL: `ftp://ftp.ps.uni-sb.de/pub/papers/ProgrammingSysLab/ines97.ps.Z`.

[38] Robert Paige et Robert E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, décembre 1987.

[39] Jens Palsberg. Efficient inference of object types. *Information and Computation*, 123(2):198–209, 1995. URL: `http://www.cs.purdue.edu/homes/palsberg/paper/ic95-p.ps.gz`.

[40] Jens Palsberg et Patrick M. O'Keefe. A type system equivalent to flow analysis. In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, San Francisco, Calif.* ACM, janvier 1995. URL: `ftp://ftp.daimi.aau.dk/pub/palsberg/papers/popl95.ps.Z`.

[41] Jens Palsberg et Scott Smith. Constrained types and their expressiveness. *ACM Transactions on Programming Languages and Systems*, 18(5):519–527, septembre 1996. URL: `http://www.cs.purdue.edu/homes/palsberg/paper/toplas96-ps.ps.gz`.

[42] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, juillet 1994. URL: `http://www.cs.indiana.edu/hyplan/pierce/ftp/fsubpopl.ps.gz`.

[43] François Pottier. Simplifying subtyping constraints. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, pp. 122–133, janvier 1996. URL: `http://pauillac.inria.fr/~fpottier/publis/ICFP96.ps.gz`.

[44] Didier Rémy. Extending ML type system with a sorted equational theory. Rapport technique 1766, INRIA, Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France, 1992. URL: ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/eq-theory-on-type%s.ps.gz.

[45] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter et John C. Mitchell (rédacteurs), *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993. URL: ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/taoop1.ps.gz.

[46] Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In Masami Hagiya et John C. Mitchell (rédacteurs), *International Symposium on Theoretical Aspects of Computer Software*, pp. 321–346, Sendai, Japan, avril 1994. Springer-Verlag. URL: ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/tacs94.ps.gz.

[47] Didier Rémy et Jérôme Vouillon. Objective ML: A simple object-oriented extension of ML. In *Proceedings of the 24th ACM Conference on Principles of Programming Languages*, pp. 40–53, Paris, France, janvier 1997. URL: ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/objective-ml!popl97.ps.gz.

[48] Jerzy Tiuryn. Subtype inequalities. In Andre Scedrov (rédacteur), *Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science*, pp. 308–317, Santa Cruz, CA, juin 1992. IEEE Computer Society Press.

[49] Valery Trifonov et Scott Smith. Subtyping constrained types. In *Proceedings of the Third International Static Analysis Symposium, LNCS*, volume 1145, pp. 349–365. SV, septembre 1996. URL: http://www.cs.jhu.edu/~trifonov/papers/subcon.ps.gz.

[50] Andrew K. Wright. Polymorphism for imperative languages without imperative types. Rapport technique 93-200, Rice University, février 1993.

[51] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, décembre 1995.

[52] Andrew K. Wright et Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, novembre 1994. URL: http://www.cs.rice.edu/CS/PLT/Publications/ic94-wf.ps.gz.

Index