N° d'ordre: 000

THÈSE DE DOCTORAT DE L'UNIVERSITÉ PARIS 6

Spécialité
Informatique

présentée par

Emmanuel Coquery

pour obtenir le grade de

Docteur de l'université de Paris 6

École doctorale : Informatique, Télécommunications et Electronique de Paris
UFR D'INFORMATIQUE
Laboratoire : INRIA-Rocquencourt
Équipe d'accueil : Contraintes

Titre de la thèse :
Typage et programmation en logique avec contraintes

Soutenue le 3 Décembre 2004 devant le jury composé de :

Me. :	Irène	Guessarian	Présidente
MM. :	Jean	Goubault-Larrecq	Rapporteurs
	Jan	Maluszynski	
	François	Pottier	
Me. :	Véronique	Donzeau-Gouge	Examinatrice
M. :	François	Fages	Directeur

Résumé

Les langages logiques avec contraintes sont des langages non typés. Les programmeurs qui utilisent ces langages bénéficient donc d'une grande souplesse de programmation, au détriment des avantages apportés par le typage, en particulier la détection d'erreurs de programmation.

Cette thèse décrit un système de types pour les langages logiques avec contraintes, de sa formalisation à sa mise en œuvre. Ce système combine polymorphisme paramétrique, sous-typage et surcharge pour obtenir la souplesse nécessaire au typage de programmes utilisant des techniques de programmation logique avec contraintes comme la méta-programmation et l'utilisation conjointe de différents domaines de contraintes. Le système est prouvé cohérent par rapport à la résolution CSLD et par rapport à un modèle d'exécution typé avec substitutions. Un algorithme de vérification des types avec inférence du type des variables et gestion efficace de la surcharge a également été développé, ainsi qu'un algorithme heuristique d'inférence de type pour les prédicats.

Cette thèse s'intéresse également au problème de la satisfiabilité des contraintes d'ordre dans les quasi-treillis, qui constituent des structures de types riches, ajoutant ainsi de la souplesse au système de type. Le problème de satisfiabilité est montré NP-complet dans le cas de quasi-treillis dont les extrema sont des constantes en nombre fini. La complexité de l'algorithme devient $O(n^3)$ dans le cas où toutes les variables du système de contraintes sont bornées. Un algorithme pour le calcul explicite de solutions est également présenté. De plus, cet algorithme permet de tester la satisfiabilité dans des quasi-treillis dont seuls les maxima sont des constantes en nombre fini. Ces résultats sur les quasi-treillis sont généraux et leur portée n'est pas limitée au typage.

Mots clé : Typage, Sous-typage, Polymorphisme paramétrique, Surcharge, Contraintes de sous-typage, Programmation logique avec contraintes.

Abstract

Constraint logic languages are untyped. The programmers that use these languages therefore benefit from a high flexibilty in programming, to the detriment of the advantages of typing, in particular the detection of programming errors.

This thesis describes a type system for constraint logic languages, from its formalization to its implementation. This system combines parametric polymorphism with subtyping and overloading to obtain the flexibility that is needed for typing programs using logic programming techniques such as metaprogramming and the simultaneous use of different constraint domains. The system is proven coonsistent with reference to the CSLD resolution and with reference to a typed execution model with substitutions. An type checking algorithm, with type inference for variables and an efficient management of overloading was developped, as well as a heuristic algorithm for infering the type of predicates.

This thesis also addresses the problem of the satisfiability of ordering constraints in quasi-lattices, which are rich type structures, thus adding more flexibility to the type system. The problem of satisfiability is proven NP-complete for the case of quasi-lattices where the set of extrema is finite and contains only constants. The complexity of this algorithm becomes $O(n^3)$ when all the variables in the constraint system are bounded. An algorithm for computing explicit solutions is also presented. Moreover, this algorithm allows for testing the satisfiability in quasi-lattices where only the maxima are constants and in finite number. These results on quasi-lattices are general and are not limited to typing.

Keywords : Typing, Subtyping, Parametric polymorphism, Overloading, Subtyping constraints, Constraint logic programming.

Remerciements

Je tiens tout particulièrement à remercier François Fages pour m'avoir guidé et soutenu durant toute la durée de ma thèse. Ses nombreux conseils, tant sur le plan purement scientifique que sur la démarche d'un chercheur m'ont été très précieux.

Jean Goubault-Larrecq, François Pottier et Jan Małuszyński ont accepté la lourde tâche de rapporteur. Je les remercie chaleureusement pour le temps et l'intérêt qu'ils ont porté à mon travail.

Je remercie également Véronique Donzeau-Gouge et Irène Guessarian qui ont bien voulu faire partie de ce jury.

Je tiens également à remercier les membres du projet Contraintes au sein duquel j'ai effectué cette thèse, pour leur bonne humeur quotidienne.

Je remercie tous les membres de ma famille et tous mes amis pour leur soutien et leurs encouragements sans faille.

Enfin, je remercie ma femme, Sandrine-Dominique Gouraud, pour son amour, son soutien indéfectible et la patience dont elle a fait preuve à mon égard. Elle a été pour moi une extraordinaire source d'inspiration et de courage.

Table des matières

Chapitre 1

Introduction

Les langages de la classe CLP($\mathcal{X}$) (langages logiques avec contraintes), introduits par Jaffar et Lassez [39], sont non typés. Un des avantages de cette caractéristique est une grande souplesse de programmation. Cette souplesse est mise à profit dans des techniques largement utilisées en programmation logique avec contraintes. Ainsi, les programmes logiques avec contraintes font souvent appel à la méta-programmation pour effectuer des manipulations de termes, ou utiliser l'ordre supérieur via le prédicat `call/1`. Une autre pratique répandue consiste à utiliser conjointement différents solveurs de contraintes, par exemple en combinant des contraintes sur les booléens avec des contraintes sur les domaines finis.

En revanche, les langages typés possèdent un certain nombre d'avantages [5]. Le premier d'entre eux est la détection d'erreurs de programmation. Celle-ci se fait en associant à chaque donnée du programme un type et en vérifiant que l'utilisation qui est faite de cette donnée est compatible avec ce type. Par exemple, on peut donner à `1` le type int, à `[]` le type list et à `+`/2 le type int $\times$ int $\rightarrow$ int, ce qui signifie que le type de ses arguments doit être compatible avec int et que l'expression résultant aura le type int. On détectera alors une erreur dans l'expression `[]+1`, le type de `[]` étant incompatible avec le type du premier argument de `+`/2. Cette vérification des types se fait par une analyse abstraite du programme, à l'aide d'un ensemble de règles logiques également appelé *système de types*. La correction d'un système de types, c'est-à-dire l'absence d'erreur de type à l'exécution, est souvent obtenue par un théorème dit d'auto-réduction. Ce théorème exprime qu'une expression correctement typée se dérive toujours en une expression correctement typée.

Le typage permet également d'avoir une vision abstraite d'un programme, dans le sens où les types représentent une certaine abstraction des valeurs pouvant être prises par les variables, être utilisées comme arguments d'un sous-programme ou résulter de l'application d'une fonction. Cela peut, par exemple, être utilisé

pour l'optimisation des programmes ou la documentation du code.

Des systèmes de types ont été créés pour les langages de la classe CLP($\mathcal{X}$), afin de leur apporter les avantages des langages typés. On peut répartir ces systèmes en deux groupes, selon la manière dont ils analysent les programmes : les systèmes *prescriptifs* et les systèmes *descriptifs*.

Les systèmes prescriptifs visent à vérifier la cohérence du programme par rapport à un ensemble de déclarations données par le programmeur. Ces systèmes sont proches des systèmes de types utilisés pour les langages fonctionnels ou impératifs. Notamment, leur correction est effectivement exprimée à travers un théorème d'auto-réduction. Un exemple de tel système est celui de Mycroft et O'Keefe [50] qui est une adaptation du système de Damas et Milner [17] pour ML.

D'un autre côté, l'intuition de base des systèmes descriptifs, donnée par Mishra [46], est qu'une formule qui échoue peut être considérée comme erronée. Ces systèmes utilisent les types pour approximer l'ensemble des succès des prédicats et des buts, c'est-à-dire l'ensemble des termes pour lesquels un prédicat peut réussir. Ainsi, Heintze et Jaffar [33, 34] utilisent des types sémantiques exprimés à l'aide de formules ensemblistes. Frühwirth, Shapiro Vardi et Yardeni [26] utilisent des programmes logiques comme types pour les programmes logiques. Plus récemment Drabent, Małuszyński et Pietrzak [19], utilisent des grammaires paramétrées pour décrire les succès des prédicats.

On oppose souvent ces deux groupes en affirmant que le typage descriptif se caractérise par l'inférence de types, contrairement au typage prescriptif caractérisé par la vérification des types [32]. Cependant il est possible d'utiliser l'inférence de types dans un système prescriptif, notamment de manière à alléger la tâche de l'utilisateur au niveau des déclarations de types, comme dans le cas de des langages fonctionnels à la ML [17], où seules les déclarations de types des structures de données sont nécessaires.

Dans cette thèse, nous nous intéresserons plus particulièrement aux systèmes prescriptifs pour la programmation en logique avec contraintes. De tels systèmes sont particulièrement adaptés pour assurer la compositionalité des programmes, c'est-à-dire adaptés au développement et à l'utilisation de bibliothèques, que ce soit à des fins de réutilisation de code, ou dans le cadre de l'écriture de programmes de taille importante. En effet, dans la plupart des langages de programmation typés, l'interface d'une bibliothèque consiste en un ensemble de déclarations décrivant les fonctions fournies par cette dernière, ainsi que les structures de données manipulées par ces fonctions. Le typage de l'implantation de la bibliothèque permet de vérifier que le code des fonctions correspond bien au type déclaré dans l'interface, alors que le typage de code utilisant cette bibliothèque assure que ces fonctions seront utilisées conformément à leur déclaration. Dans les deux cas, on vérifie la cohérence du code par rapport aux déclarations de

types de l'interface de la bibliothèque, ce qui correspond bien à une discipline de typage prescriptif, par opposition à un typage descriptif qui chercherait d'une part à approximer l'ensemble des succès des différents appels aux prédicats de cette bibliothèque, et d'autre par à vérifier que les succès des prédicats définis dans cette bibliothèque correspondent à ce qui a été déclaré.

Un système de types souple

Quel qu'il soit, l'utilisation d'un système de types impose naturellement des restrictions sur la forme des programmes qu'il est possible d'écrire. L'ajout d'un système de types à un langage CLP($\mathcal{X}$) va donc forcément réduire la souplesse de programmation de celui-ci. Il est par conséquent important de limiter le plus possible l'impact du système de types sur cette souplesse de programmation. En particulier, le système de types devrait autoriser la méta-programmation et permettre d'utiliser conjointement différents domaines de contraintes, ce qui n'est pas possible dans le système de Mycroft et O'Keefe [50]. Le *polymorphisme*, c'est à dire la possibilité pour une expression d'avoir plusieurs types, permet d'obtenir des systèmes de types plus fins, imposant moins de restrictions au programmeur. Il existe trois principales formes de polymorphisme : le *polymorphisme paramétrique* [67, 60, 17], le *sous-typage* [4] et la *surcharge*, également appelée polymorphisme "ad-hoc". Nous allons à présent voir comment chacune de ces formes de polymorphisme peut être utilisée pour garder un maximum de la souplesse de programmation des langages CLP($\mathcal{X}$).

Polymorphisme paramétrique

Le polymorphisme paramétrique a été introduit par Strachey [67] et utilisé par Damas et Milner [45, 17], dans le cadre du langage ML. Il consiste à autoriser dans les types la présence de variables, ou *paramètres*, permettant ainsi d'obtenir un type abstrait par rapport à ces paramètres. Par exemple, on peut donner au constructeur de liste `[|]`/2 le type $\forall\alpha.\alpha \times \text{list}(\alpha) \rightarrow \text{list}(\alpha)$. Ici, α correspond au type des éléments de la liste construite. Ainsi, en lui passant en argument un élément de type τ et une liste contenant des éléments du même type τ, on obtient une liste dont les arguments ont le type τ, c'est-à-dire de type list(τ). Cette forme de polymorphisme est donc particulièrement adaptée à la manipulation de structures de données abstraites telles que les paires, les listes ou les arbres.

Le système de types de Mycroft et O'Keefe [50, 44, 35], qui utilise le polymorphisme paramétrique, a été implanté pour Prolog dans des systèmes tels que Gödel [35] et Mercury [66], ou encore dans λProlog [51, 42].

La souplesse du polymorphisme paramétrique n'est cependant pas suffisante

pour traiter la méta-programmation ou l'utilisation conjointe de différents domaines de contraintes. En effet, on ne peut pas donner de type à `=../2`, qui associe à un terme la liste composée du nom de son symbole de tête et de ses différents arguments, car cette liste est hétérogène : tous ses éléments n'ont pas le même type. Une des utilisations principales de la méta-programmation est de compenser l'absence d'ordre supérieur par l'utilisation du prédicat `call/1`, qui appelle le but correspondant syntaxiquement au terme qui lui est passé en argument. Ces termes peuvent en particulier être obtenus par décomposition recomposition d'autres termes, en utilisant par exemple `=../2`. On veut donc pouvoir voir un but comme un terme, ce qui n'est pas permis ici. De même, utiliser une variable avec différents domaines de contraintes suppose que ces domaines portent sur des termes de même type, ce qui n'est généralement pas le cas. Par exemple il est possible d'utiliser simultanément des contraintes sur les booléens et sur les domaines finis, si les premiers sont codés par les valeurs `0` et `1`. Ainsi les valeurs booléennes correspondent à une partie des valeurs de domaine fini, mais pas à toute ces valeurs. Donner le type int aux contraintes booléennes simplement pour pouvoir les utiliser conjointement avec les contraintes de domaine fini ne paraît pas justifié. Une possibilité permettant d'ajouter de la souplesse à ce système consiste à y ajouter le sous-typage.

Sous-typage

Le *sous-typage* est une notion fondamentale introduite par Cardelli [4] et par Mitchell [47], comme une autre forme de polymorphisme. Elle repose sur un ordre $\leq$ sur les types et sur la règle de sous-typage, qui a en général la forme suivante :

$$\frac{t : \tau \quad \tau \leq \tau'}{t : \tau'}$$

et qui dit que si t a le type τ et que τ est un sous-type de τ', alors t a également le type τ'.

L'utilisation du sous-typage en programmation logique a comme point de départ le cadre de la logique ordo-sortée [31, 65, 30]. Il existe quelques systèmes de types prescriptifs avec sous-typage pour la programmation logique. On peut citer les systèmes de Dietrich et Hagl [18], Smolka [64], Hanus [32], Hill et Topor [36], Yardeni, Frühwirth et Shapiro [71], ou encore Beierle [1], qui combinent polymorphisme paramétrique et sous-typage. Sauf dans le cas de Dietrich et Hagl [18], qui ne donnent pas d'algorithme général, cette combinaison est réalisée en utilisant une relation de sous-typage *structurelle*, c'est-à-dire construite à partir d'une relation portant exclusivement sur des constantes, puis étendues aux types par morphisme. Par exemple si on a dans la relation de sous-typage boolean $\leq$ int, on a également list(boolean) $\leq$ list(int). Même si on ne peut pas avoir directement list(int) $\leq$ set(int) – car list(.) et set(.) ne sont pas des constantes – il est

possible d'encoder de telles relations dans le sous-typage structurel. Il suffit d'introduire un constructeur de type κ_n, d'arité $n+1$, pour chaque n correspondant à l'arité d'un constructeur de type de la structure initiale, puis de donner à tous les constructeurs de la structure initiale l'arité 0. Le premier argument des κ_n encode alors le nom du constructeur de la structure initiale. Ainsi pour tester list(boolean) $\leq$ set(int), il suffit de tester $\kappa_1(\text{list}, \text{boolean}) \leq \kappa_1(\text{set}, \text{int})$.

Toujours dans l'esprit de la logique ordo-sortée, Fages et Paltrinieri [23] utilisent le sous-typage pour traiter l'utilisation conjointe de différents domaines de contraintes, celui-ci permettant la coercition des types correspondant à ces domaines. Prenons l'exemple d'une utilisation conjointe de contraintes sur les booléens (CLP($\mathcal{B}$)) et sur les domaines finis (CLP($\mathcal{FD}$)), ce qui est possible si on représente les booléens par les entiers `0` et `1`. On donne ainsi à `0` et `1` le type boolean et aux autres entiers le type int. On donne au symbole symbole `+/2` le type int $\times$ int $\rightarrow$ int. On donne à la contrainte booléenne `#<=>/2` le type boolean $\times$ boolean $\rightarrow$ pred et à la contrainte de domaines finis `#=/2` le type int $\times$ int $\rightarrow$ pred. En considérant la relation de sous-typage boolean $\leq$ int, et en donnant le type boolean à toutes les variables, le but `A #<=> E1, B #<=> E2, C #<=> E3, A+B+C #= 2`, qui exprime qu'il y a exactement 2 valeurs vraies parmi `E1`, `E2` et `E3` est bien typé.

Le système de Fages et Paltrinieri [23] utilise également le sous-typage pour gérer la méta-programmation, en introduisant le type term comme majorant de tous les autres types. On peut ainsi donner le type term $\times$ list(term) au prédicat `=../2`, le deuxième argument étant une liste de termes, ayant des types variés, mais tous sous-types de term. Ainsi le but : `(3+2) =.. [ + , 3 , 2 ]` est bien typé avec `+/0` : atom, `3` : int et `2` : int, puisque atom et int sont des sous-types de term. Cette utilisation nécessite que la relation de sous-typage soit étendue pour permettre des inégalités plus générales, telles que list(α) $\leq$ term. Le sous-typage est alors dit *non structurel non homogène*.

La correction des systèmes de type, exprimée à travers le théorème d'auto-réduction, est un résultat bien connu pour les systèmes avec polymorphisme paramétrique sans sous-typage [50, 44]. Cependant lorsque l'on introduit le sous-typage, l'absence de flot de données fixé rend ce résultat difficile à obtenir. En effet, dans le cadre des langages à objets Palsberg, O'Keefe et Smith [53, 54], ont montré que l'inférence de types est fortement liée au flot de données du programme. Ainsi, ni Beierle [1], ni Hanus [32] ne montrent l'auto-réduction pour leur système. En général, les types sont gardés à l'exécution [32, 71] ou des modes sont introduits afin de fixer le flot de données [18, 66, 62]. Fages et Paltrinieri [23, 22] ont montré un théorème d'auto-réduction par rapport au modèle d'exécution abstrait CSLD[1] qui accumule les contraintes au long de la dérivation sans effectuer

[1]pour résolution Linéaire pour les programmes Définis avec Contraintes et atomes Sélectionnés

les substitutions.

Surcharge

Bien que la combinaison du polymorphisme paramétrique et du sous-typage offre une grande flexibilité, cette dernière n'est pas suffisante pour traiter certains symboles de fonctions utilisés de manières très différentes. Par exemple, le symbole `-/2` peut être utilisé soit comme l'opérateur arithmétique de la soustraction, soit comme un constructeur de paires (la paire $(\mathtt{a}, \mathtt{b})$ étant codée par `a-b`), comme dans le prédicat `keysort/2` qui trie une liste de paires en fonction du premier élément de chaque paire. Ainsi, dans le premier cas on peut, par exemple, donner à `-/2` le type $\text{int} \times \text{int} \rightarrow \text{int}$ alors que dans le deuxième cas, on lui donnera le type $\alpha \times \beta \rightarrow \text{pair}(\alpha, \beta)$. La surcharge consiste précisément à donner plusieurs types à un même symbole, associant ainsi à `-/2`, à la fois le type $\text{int} \times \text{int} \rightarrow \text{int}$ et le type $\alpha \times \beta \rightarrow \text{pair}(\alpha, \beta)$.

La surcharge peut interférer avec le sous-typage dans le sens où tous deux permettent de traiter certaines situations comme les expressions arithmétiques mêlant flottants et entiers. Pour gérer des expressions comme `3.2 + 1`, l'opérateur `+/2` doit accepter comme argument à la fois des entiers de type int et des flottants de type float. La solution utilisant la surcharge consiste à donner quatre types à `+/2` : $\text{int} \times \text{int} \rightarrow \text{int}$, $\text{int} \times \text{float} \rightarrow \text{float}$, $\text{float} \times \text{int} \rightarrow \text{float}$ et $\text{float} \times \text{float} \rightarrow \text{float}$, comme cela se fait dans d'autres langages comme C. Si on ajoute la relation de sous-typage $\text{int} \leq \text{float}$, il est possible de ne donner à `+/2` que le type $\alpha \times \alpha \rightarrow \alpha, \alpha \leq \text{float}$. Dans le cas de `3.2 + 1`, en choisissant float pour α, l'expression serait alors bien typée avec le type float.

Inférence de types et contraintes de sous-typage

Comme dans la plupart des systèmes de types prescriptifs, il est possible d'utiliser des des algorithmes d'inférence de types afin d'omettre certaines déclarations de types, tout en vérifiant que le programme est correctement typé. En l'absence complète d'inférence de types, il y a une déclaration de type pour chaque sous-expression du programme, ce qui est inutilisable en pratique. L'inférence de types pour les expressions est donc indispensable. Or celle-ci peut conduire à résoudre des problèmes non triviaux, comme dans l'exemple suivant : considérons l'expression `[1]`. Intuitivement cette expression a le type list(int). Cependant inférer le type de cette expression requiert de trouver une valeur à un certain nombre de variables de type : ici le constructeur de listes `[|]/2` a le type $\alpha \times \text{list}(\alpha) \rightarrow \text{list}(\alpha)$. Comme le premier argument `1` a le type int, et le deuxième `[]` a le type $\text{list}(\beta)$,on en déduit qu'une solution associe à α et β le type int. En fait, pour trouver les

valeurs de α et β, nous avons implicitement résolu le *système de contraintes de sous-typage* suivant : $\tau = \text{list}(\alpha), \text{int} \leq \alpha, \text{list}(\beta) \leq \text{list}(\alpha)$, où τ est le type de `[1]`. On voit donc à travers cet exemple que la résolution des systèmes de contraintes de sous-typage, de la forme $\tau \leq \tau'$, joue un rôle essentiel dans le système de type.

La difficulté qu'il y a à résoudre des contraintes de sous-typage dépend fortement des hypothèses sur la relation de sous-typage et sur la forme des contraintes. Dans le cas d'un sous-typage *atomique* [47, 48], c'est-à-dire ne portant que sur les constructeurs de type sans arguments, Tiuryn a montré que si l'ordre est une union disjointe de treillis, le problème de la satisfiabilité est linéaire [69]. Pratt et Tiuryn ont montré que dans le cas des "n-crowns"[2], le problème est NP-complet [59]. Le cas du sous-typage structurel, où deux types sont comparables uniquement s'ils ont le même squelette, c'est-à-dire s'ils diffèrent uniquement sur leurs feuilles, a été introduit par Fuh et Mishra [27, 28] et un algorithme de résolution efficace a été donné par Simonet [61]. Dans le cadre d'un sous-typage non structurel homogène, c'est-à-dire autorisant des inégalités entre constructeurs de même arité, Frey a montré que le problème de satisfiabilité dans les types finis est P-SPACE complet [24]. Dans le cas non structurel non homogène, c'est-à-dire autorisant des inégalités entre constructeurs d'arités différentes, si pour chaque constructeur de type, l'ensemble de ses majorants admet un maximum, et si le système est acyclique et linéaire à gauche alors le problème de la satisfiabilité est linéaire [23]. Toujours dans le cas non structurel non homogène, si l'ordre des constructeurs de type forme un treillis, mais sans restriction sur la forme des contraintes de sous-typage, Trifonov, Smith et Pottier [70, 56] ont donné un algorithme pour tester la satisfiabilité en $O(n^3)$, ainsi que des algorithmes de simplification permettant un calcul de solutions explicites.

Origine et contributions de la thèse

L'origine de cette thèse se trouve dans le système de Fages et Paltrinieri [23] et dans le développement d'une implantation permettant de le tester sur des programmes existants, non typés de taille réelle.

Ce système a l'avantage d'être très souple, grâce à la combinaison du sous-typage et du polymorphisme paramétrique. Il permet ainsi de gérer la méta-programmation, l'utilisation conjointe de différents domaines de contraintes et l'utilisation de structures de données abstraites vis-à-vis des éléments qu'elles contiennent. Cependant, cette souplesse n'est pas suffisante pour que le système puisse être utilisé sur des programmes réels tels que les bibliothèques de SICStus Prolog. Nous avons donc amélioré ce système dans trois directions.

[2]Une "n-crown" est un ensemble $\{\kappa_1, \ldots, \kappa_{2n}\}$, ordonné de la manière suivante : pour tout $1 \leq i \leq n$, $\kappa_{2i-1} \leq \kappa_{2i}$, pour tout $1 \leq i \leq n-1$, $\kappa_{2i+1} \leq \kappa_{2i}$ et $\kappa_1 \leq \kappa_{2n}$

Une première amélioration consiste à permettre l'inférence du type des variables dans les clauses et les buts. En effet, l'hypothèse de linéarité à gauche des système de contraintes empêche d'utiliser l'algorithme de Fages et Paltrinieri à cette fin. Une première expérimentation a été réalisée en utilisant la bibliothèque de résolution de contraintes de sous-typage Wallace de Pottier [56, 58]. Cependant, les contraintes sont résolues dans une structure de treillis, ce qui impose la présence d'un type minimal $\bot$, ne correspondant à aucune valeur. Ce type $\bot$ est toujours une solution pour le type des variables, de par la forme des systèmes de contraintes à résoudre, qui n'imposent que des bornes supérieures à ce type. En présence de ce type $\bot$, aucune erreur ne peut donc être détectée sur les variables. Par conséquent il serait intéressant de regarder des structures d'ordre de sous-typage plus générales que les treillis, dans lesquelles ce type $\bot$ ne serait pas obligatoire. En 1988, Smolka [63] a émis la conjecture selon laquelle la satisfiabilité des contraintes de sous-typage non structurelles non homogènes dans les structures de quasi-treillis est décidable. Les quasi-treillis sont des ordres partiels dans lesquels deux éléments ont une borne supérieure (resp. inférieure) si et seulement si ils ont un majorant (resp. minorant) commun. En particulier, le type $\bot$ n'est pas obligatoire dans ces structures.

Nous nous sommes donc intéressés à la résolution des contraintes de sous-typage dans ces structures. En nous appuyant sur le formalisme des types avec étiquettes développé pour les treillis par Pottier [57], nous étudions les structures de quasi-treillis de types. En particulier, nous donnons des conditions suffisantes pour qu'un quasi-treillis de constructeurs de types engendre un quasi-treillis de types. Nous montrons que le problème de la satisfiabilité des contraintes de sous-typage dans les quasi-treillis est NP-complet, lorsque ces derniers possèdent un nombre fini d'extrema, ces extrema étant tous constants. Nous donnons un algorithme de test de satisfiabilité dont la complexité se réduit à $O(n^3)$, où n est la taille du système de contraintes, dans le cas où le système est pré-clos, c'est-à-dire si toutes ses variables sont bornées. Nous donnons également un algorithme permettant de calculer explicitement des solutions à un système de contraintes de sous-typage. Cet algorithme ne nécessitant pas de condition sur les minima (ou bien symétriquement les maxima) du quasi-treillis, on obtient un algorithme permettant de tester la satisfiabilité dans des quasi-treillis possédant un nombre fini de maxima (resp. minima), tous constants. Cet algorithme a fait l'objet d'une implantation dans le langage CHR (Contraint Handling Rules [25]). Cette implantation s'est avérée très performante en pratique, grâce à la gestion de l'unification des variables de types.

Une deuxième amélioration réside dans l'ajout de la surcharge, ou polymorphisme "ad-hoc", au système. En effet, les expérimentations que nous avons

menées sur les langages SICStus Prolog et GNU-Prolog ont montré une grande utilisation de symboles surchargés. L'exemple du codage des paires avec le symbole -/2 est caractéristique. L'absence de gestion de la surcharge par le système de type conduirait alors à réécrire un certain nombre de bibliothèques standard, ce qui introduirait une incompatibilité avec les programmes existants. Nous introduisons la surcharge dans le système et montrons le théorème d'auto-réduction par rapport à la résolution CSLD, ainsi que par rapport à un modèle d'exécution typé dans lequel les substitutions sont effectuées. Nous avons développé un algorithme de vérification des types pour ce système, avec inférence du type des variables et désambiguïsation des symboles surchargés. Les contraintes de sous-typage sont résolues en utilisant le précédent algorithme de calcul de solution explicites dans les quasi-treillis. La désambiguïsation des symboles surchargés est réalisée en retardant au maximum les points de choix, ce qui permet d'en limiter l'explosion combinatoire.

Enfin, il est intéressant de pouvoir bénéficier de l'inférence de type pour les prédicats, en particulier lors de l'écriture de gros programmes ou lors des phases de prototypage pendant lesquelles le code change beaucoup. Bien que le type d'un prédicat constitue une bonne documentation, il peut en effet être pénible de donner un type à des prédicats intermédiaires, qui peuvent changer facilement en cas de modification d'un algorithme. Cependant, le type $\text{term} \times \ldots \times \text{term} \rightarrow \text{pred}$ est toujours un type possible pour un prédicat. Comme le type term est nécessaire au typage de la méta-programmation, on ne peut pas se contenter de résoudre les contraintes dans une structure plus générale, sans term. Pour répondre à ce problème, nous introduisons un algorithme heuristique pour l'inférence de types des prédicats non déclarés, afin d'obtenir des types plus informatifs que le type $\text{term} \times \ldots \times \text{term}$.

Les algorithmes de vérification et d'inférence de types ont été implantés dans un logiciel, appelé TCLP [10, 11]. Ce logiciel a été écrit en Prolog et CHR. Des bibliothèques de types ont été écrites pour trois dialectes Prolog : ISO Prolog, SICStus Prolog et GNU-Prolog. Des tests, notamment réalisés sur des bibliothèques SICStus Prolog, montrent l'efficacité en pratique des algorithmes de vérification et d'inférence. Par ailleurs, TCLP est capable de se typer lui-même.

Plan

Le chapitre 2 introduit des définitions sur les langages logiques avec contraintes, ainsi que le modèle d'exécution CSLD. Il présente également le langage CHR [25] qui a été utilisé pour l'implantation de TCLP. Le reste de la thèse est découpé en deux parties.

La première partie traite des contraintes de sous typage. Le chapitre 3 définit le langage des types que nous allons manipuler. Le chapitre 4 traite des quasi-treillis de types et donne notamment des conditions suffisantes pour qu'un quasi-treillis de constructeurs de types engendre un quasi-treillis de types. Le chapitre 5 traite de la satisfiabilité des contraintes de sous-typage dans les quasi-treillis de types. En particulier, nous y montrons que le problème de la satisfiabilité des contraintes dans des quasi-treillis dont les extrema sont constants et en nombre fini est NP-complet. Le chapitre 6, décrit l'algorithme de calcul de solutions explicites dans les quasi-treillis dont l'ensemble des maxima est constitué d'un nombre fini de constantes. Nous y décrivons également son implantation en CHR.

La deuxième partie est consacrée au système de types lui-même. Le chapitre 7 décrit le système de types et contient les théorèmes d'auto-réduction par rapport au modèle d'exécution CSLD et par rapport au modèle d'exécution avec substitution conservant le type des variables à l'exécution. Nous y décrivons également les choix que nous avons effectués concernant les bibliothèques de types pour ISO Prolog et SICStus Prolog. Le chapitre 8 traite de la vérification des types d'un programme, en particulier du problème de l'inférence de types pour les variables et de la désambiguïsation des symboles surchargés. Enfin, le chapitre 9 décrit l'inférence de types heuristique pour les prédicats.

Chapitre 2

Préliminaires CLP

J. Jaffar et J.-L. Lassez [38, 39] ont introduit en 1986 la classe des langages logiques avec contraintes (CLP), dont les racines se situent dans la programmation logique. Ils ont ainsi défini un cadre unifié pour décrire la sémantique de plusieurs langages de programmation par contraintes. Dans ce cadre, le domaine des contraintes, noté $\mathcal{X}$, devient un paramètre : la classe CLP($\mathcal{X}$) est la classe des langages logiques avec contraintes sur le domaine $\mathcal{X}$. Ainsi, Prolog II [7, 8] est une instance de ce cadre si on prend pour $\mathcal{X}$ le domaine des arbres rationnels. Un autre exemple est CLP($\mathcal{R}$) [40] dont le domaine est l'ensemble des nombres réels. Il est également possible de combiner plusieurs domaines de contraintes, comme dans Prolog IV [9], qui fournit des contraintes sur les arbres rationnels, les listes, les entiers, les booléens, les nombres rationnels et les réels, avec certaines coercitions automatiques.

Nous allons à présent présenter formellement les langages CLP($\mathcal{X}$), en commençant par les contraintes, puis les programmes et enfin le modèle d'exécution qu'est la résolution linéaire pour programmes définis avec contraintes et atomes sélectionnés (résolution CSLD). Les notations utilisées sont en grande partie issues du livre de Fages [21].

2.1 Programmes CLP($\mathcal{X}$)

2.1.1 Contraintes

Le langage des contraintes est un langage du premier ordre défini par :

- S_f un ensemble de constantes et de symboles de fonctions donnés avec leur arité et notés $\mathtt{f}/n, \ldots$.
- S_c un ensemble de symboles de contraintes donnés avec leur arité et notés $\mathtt{c}/n, \ldots$, contenant $\mathtt{true}/0$ et $=/2$.
- $\mathcal{W}$ un ensemble dénombrable de variables, notées `A`, `B`, `X`, `Y` ....

Une *contrainte atomique* est de la forme $\mathtt{c}(t_1, \ldots, t_n)$ où $t_1, \ldots, t_n$ sont des termes formés sur $S_f \cup \mathcal{W}$. Le *langage des contraintes* est la clôture de l'ensemble des contraintes atomiques par conjonction et quantification existentielle. On note $c, d, \ldots$ les contraintes.

Une *structure d'interprétation* $\mathcal{X}$ d'un langage de contraintes formé sur $(S_f, S_c, \mathcal{W})$ est un quadruplet $(\mathcal{D}, E, O, R)$, où :

- $\mathcal{D}$ est un ensemble appelé le *domaine* des contraintes.
- $E \subseteq \mathcal{D}$ est un ensemble d'éléments de $\mathcal{D}$ associés aux constantes de S_f.
- O est un ensemble d'opérateurs sur $\mathcal{D}$ associés aux symboles de fonction de S_f. À chaque symbole de fonction $\mathtt{f}/n \in S_f$ est ainsi associé l'opérateur $[\mathtt{f}/n] : \mathcal{D}^n \to \mathcal{D}$.
- R est un ensemble de relations sur $\mathcal{D}$ associées aux symboles de contraintes. À chaque symbole de contrainte $\mathtt{c}/n \in S_c$ est ainsi associée la relation $[\mathtt{c}/n] : \mathcal{D}^n \to \{0, 1\}$.

Exemple 2.1 : Considérons le domaine de Herbrand. Les symboles de fonction sont les chaînes de caractères et les seuls symboles de contraintes sont `=`/2 et `true`/0. Le domaine $\mathcal{D}$ est l'ensemble des termes. L'interprétation de chaque symbole de fonction $\mathtt{f}/n$ est la fonction qui associe à $t_1, \ldots, t_n$ le terme dont le symbole de tête est f et dont les sous-termes immédiats sont $t_1, \ldots, t_n$. ◇

Exemple 2.2 : Considérons les contraintes sur les réels. On se donne comme symboles de fonction `+`/2, `-`/2, `*`/2, `/`/2 et comme constantes les suites de chiffres entre `0` et `9`, avec un point optionnel. Les symboles de prédicats sont `=<`/2, `<`/2, `=`/2, `>`/2, `>=`/2, `true`/0. Le domaine d'interprétation $\mathcal{D}$ est l'ensemble des nombres réels et les symboles de fonction et de contraintes sont interprétés de manière habituelle. ◇

Exemple 2.3 : Considérons à présent le domaine booléen. Les constantes sont `t` et `f`, et on a comme symboles de fonctions `or`/2, `and`/2, `not`/1, `=>`/2 et `<=>`/2. Les symboles de contraintes sont `true`/0, `=`/2 et `sat`/1. Le domaine $\mathcal{D}$ est l'ensemble $\{0, 1\}$. `t` est interprété par 1 et `f` par 0. Les symboles de fonctions sont interprétés de manière usuelle avec leur table de vérité. L'interprétation de `sat`/1 est la relation $\{(1)\}$, c'est-à-dire `sat`(0) est faux et `sat`(1) est vrai. ◇

Exemple 2.4 : Considérons enfin les contraintes sur les domaines finis. Les constantes sont les suites de chiffres entre `0` et `9` représentant des nombres soit compris entre deux entiers *minint* et *maxint*. Les fonctions sont les opérateurs `+`/2, `-`/2, `*`/2 et `/`/2. Les contraintes sont `=<`/2, `<`/2, `=`/2, `>`/2, `>=`/2, `true`/0. Le domaine est l'intervalle des entiers relatifs compris entre *minint* et *maxint*, les symboles de fonction et de contrainte étant interprétés de manière habituelle. ◇

Une *valuation* est une fonction $\rho : \mathcal{W} \rightarrow \mathcal{D}$, associant aux variables des éléments du domaine. Les valuations sont étendues aux termes par morphisme.

Définition 2.1 *Une valuation* ρ satisfait *une contrainte* c*, noté* $\rho \models c$ *si l'interprétation de* $\rho(c)$ *dans* $\mathcal{X}$ *est vrai, c'est-à-dire si* $\mathcal{X} \models \rho(c)$*. Dans ce cas, on dit que* ρ *est une* solution *de* c*.*

On dit qu'une contrainte c *est* satisfiable *dans* $\mathcal{X}$*, noté* $\mathcal{X} \models c$ *s'il existe une valuation* ρ *telle que* $\rho \models c$*.*

Exemple 2.5 : Considérons les contraintes de domaine finis de l'exemple 2.4. Soit la contrainte $c = \texttt{X < 2}$. Soit la valuation $\rho : \texttt{X} \mapsto 1$. Comme $1 \leq 2$, on a $\rho \models c$, et donc c est satisfiable. Au contraire, la contrainte $c_2 = \texttt{X < Y} \wedge \texttt{Y < Z} \wedge \texttt{Z < X}$ n'est pas satisfiable, car il n'existe aucune valuation ρ telle que $\rho \models c_2$. $\diamond$

2.1.2 Programmes

Nous décrivons à présent les programmes logiques avec contraintes. Nous supposons que la structure $\mathcal{X}$ est fixée et que le problème de la satisfiabilité des contraintes dans $\mathcal{X}$ est décidable. On considère un ensemble de symboles de prédicats S_p donnés avec leur arité, notés $\texttt{p}/n$, et on suppose S_p disjoint de S_c. S_p représente les relations définies par des programmes. Par la suite, on appellera *atome* une proposition atomique, notée $A, B, \ldots$, exclusivement formée sur S_f, $\mathcal{W}$ et S_p. On notera $\vec{A}$ une conjonction d'atomes et $\Box$ une conjonction vide d'atomes.

Définition 2.2 *Une* clause de programme logique avec contraintes *est une clause contenant exactement un littéral positif* $\forall A \vee \neg c_1 \vee \ldots \vee \neg c_m \vee \neg A_1 \vee \ldots \vee \neg A_n$*, où les* c_i *sont des contraintes atomiques et les* A_j *sont des atomes. On note une clause de programme :*

$$A \leftarrow c_1, \ldots, c_m \mid A_1, \ldots, A_n$$

ou, en abrégé, $A \leftarrow c \mid \vec{A}$*, avec* $c = c_1 \wedge \ldots \wedge c_m$ *et* $\vec{A} = A_1 \wedge \ldots \wedge A_n$*.* A *est appelée la* tête *de la clause et* $c_1, \ldots, c_m \mid A_1, \ldots, A_n$ *est appelé le corps de la clause. Les* variables locales *de la clause sont les variables qui n'apparaissent que dans le corps de la clause.*

Définition 2.3 *Un* programme logique avec contraintes *est un ensemble fini de clauses.*

Exemple 2.6 : Considérons le programme CLP($\mathcal{R}$) suivant :

$$\begin{aligned} \texttt{pt\_droite(X,Y,A,B)} &\leftarrow \texttt{Y = A*X+B} \mid \\ \texttt{coupe(C,D,E,F)} &\leftarrow \mid \texttt{pt\_droite(X,Y,C,D), pt\_droite(X,Y,E,F)} \end{aligned}$$

Le prédicat pt_droite/4 exprime que le point de coordonnées (X, Y) appartient à la droite d'équation $y = \mathtt{A}x + \mathtt{B}$. Le prédicat coupe/4 exprime que les droites d"équation $y = \mathtt{C}x + \mathtt{D}$ et $y = \mathtt{E}x + \mathtt{F}$ se coupent. ◇

Définition 2.4 *Un* but *est une formule logique de la forme* $c_1 \wedge \ldots \wedge c_m \wedge A_1 \wedge \ldots \wedge A_n$ *également noté* $c_1, \ldots, c_m \mid A_1, \ldots, A_n$.

2.2 Résolution CSLD

Nous présentons maintenant le modèle d'exécution des programmes logiques avec contraintes introduit par Jaffar et Lassez [39]. La restriction des définitions du programme à être des clauses de Horn de la forme $A \leftarrow c_1, \ldots, c_m \mid A_1, \ldots, A_n$ permet l'utilisation d'un système de preuve très simple, réduit à une seule règle d'inférence, appelée résolution CSLD[1]. Cette règle d'inférence est présentée par une relation de réécriture sur les buts.

Définition 2.5 *Soit* P *un programme logique avec contraintes sur la structure* $\mathcal{X}$. *La relation de réécriture* $\longrightarrow_{CSLD}$ *sur les buts est définie comme la plus petite relation satisfaisant le principe de résolution CSLD suivant :*

$$\frac{\mathtt{p}(N_1, \ldots, N_k) \leftarrow c' \mid A_1, \ldots, A_n \in P\theta \quad \mathcal{X} \models \exists(c \wedge M_1 = N_1 \wedge \ldots \wedge M_k = N_k \wedge c')}{(c \mid \vec{A}, p(M_1, \ldots, M_k), \vec{A'}) \longrightarrow_{CSLD} (c, M_1 = N_1, \ldots, M_k = N_k, c' \mid \vec{A}, A_1, \ldots, A_n, \vec{A'})}$$

où θ *est un renommage de la clause du programme avec de nouvelles variables.*

L'atome $p(M_1, \ldots, M_k)$ dans le but à réduire est appelé atome *sélectionné*. Un but B' est un *résolvant CSLD* d'un but B si $B \longrightarrow_{CSLD} B'$. On note $\longrightarrow^*_{CSLD}$ la clôture transitive réflexive de $\longrightarrow_{CSLD}$.

Une *dérivation CSLD* pour un but B est une suite finie ou infinie de buts $(B_i)_{i\in\mathbb{N}}$, tels que $B_0 = B$ et pour tout $i \in \mathbb{N}$, $B_i \longrightarrow_{CSLD} B_{i+1}$.

Une *dérivation réussie* (ou *réfutation CSLD*) est une dérivation CSLD finie qui termine avec un but ne contenant que des contraintes. Une *réponse calculée* pour un but B est une contrainte c obtenue par réfutation $CSLD$ à partir de B : $B \longrightarrow^*_{CSLD} c \mid \Box$. La *réponse calculée projetée* est la contrainte $\exists x_1 \ldots \exists x_n c$ où $\{x_1, \ldots, x_n\} = V(c) \setminus V(B)$.

Exemple 2.7: Reprenons le programme de l'exemple 2.6. Soit le but $B = \mathtt{coupe}(\mathtt{2}, \mathtt{I}, \mathtt{2}, \mathtt{J})$. On a $B \longrightarrow_{CSLD} c_1 \mid A_1$, avec $A_1 \equiv \mathtt{pt\_droite}(\mathtt{Z}_1, \mathtt{T}_1, \mathtt{C}_1, \mathtt{D}_1)$, $\mathtt{pt\_droite}(\mathtt{Z}_1, \mathtt{T}_1, \mathtt{E}_1, \mathtt{F}_1)$ et $c_1 \equiv \mathtt{2=C}_1,\ \mathtt{I=D}_1,\ \mathtt{2=E}_1,\ \mathtt{J=F}_1$.
On a ensuite $c_1 \mid A_1 \longrightarrow_{CSLD} c_2 \mid A_2$ avec $c_2 \equiv c_1, \mathtt{C}_1\mathtt{=A}_2,\ \mathtt{D}_1\mathtt{=B}_2,\ \mathtt{Z}_1\mathtt{=X}_2,\ \mathtt{T}_1\mathtt{=Y}_2,$

[1] pour résolution Linéaire pour programmes Définis avec Contraintes et règle de Sélection des atomes

$\mathtt{Y_2{=}A_2{*}X_2{+}B_2}$ et $A_2 \equiv \mathtt{pt\_droite}(\mathtt{Z_1}, \mathtt{T_1}, \mathtt{E_1}, \mathtt{F_1})$.
On a enfin $c_2 \mid A_2 \longrightarrow_{CSLD} c_3 \mid \square$, avec $c_3 \equiv c_2, \mathtt{E_1{=}A_3},\ \mathtt{F_1{=}B_3},\ \mathtt{Z_1{=}X_3},\ \mathtt{T_1{=}Y_3},\ \mathtt{Y_3{=}A_3{*}X_3{+}B_3}$.

La dérivation $B, c_1 \mid A_1,\ c_2 \mid A_2,\ c_3 \mid \square$ est une dérivation réussie, dont la réponse calculée est c_3 et la réponse calculée projetée est :
$\exists \mathtt{C_1D_1E_1F_1Z_1T_1A_2B_2X_2Y_2A_3B_3X_3Y_3}.c_3$. ◇

Au cours d'une dérivation CSLD, on accumule des contraintes, tout en vérifiant leur satisfiabilité. En pratique, les solveurs de contraintes peuvent effectuer des simplifications sur l'ensemble des contraintes et des substitutions entre deux étapes de résolution CSLD. Dans l'exemple 2.7, un solveur intelligent pourra ainsi simplifier la réponse calculée projetée en $\mathtt{I{=}J}$. Des simplifications sont également utilisées dans CLP($\mathcal{H}$) lorsque l'on utilise la résolution SLD :

Exemple 2.8 : La règle de résolution SLD est définie comme suit :

$$\frac{\mathtt{p}(N_1, \ldots, N_k) \leftarrow B_1, \ldots, B_m \in P\theta}{\vec{A}, \mathtt{p}(M_1, \ldots, M_k), \vec{A'} \longrightarrow_{SLD} \rho(\vec{A}, B_1, \ldots, B_m, \vec{A'})}$$

où θ est une substitution de renommage et $\rho = \mathrm{mgu}(\mathtt{p}(N_1, \ldots, N_k), \mathtt{p}(M_1, \ldots, M_k))$.
On peut en effet voir cette transition comme une transition CSLD suivie d'une substitution puis d'une élimination de contraintes triviales :

$$\begin{array}{ll}
\vec{A}, \mathtt{p}(M_1, \ldots, M_k), \vec{A'} & \\
\longrightarrow_{CSLD} & M_1 = N_1, \ldots, M_k = N_k \mid \vec{A}, B_1, \ldots, B_m, \vec{A'} \\
\longrightarrow_{\rho} & \rho(M_1 = N_1, \ldots, M_k = N_k \mid \vec{A}, B_1, \ldots, B_m, \vec{A'}) \\
\longrightarrow_{simplification} & \rho(\vec{A}, B_1, \ldots, B_m, \vec{A'})
\end{array}$$

L'étape de simplification se justifie par le fait que comme $\rho(M_i) \equiv \rho(N_i)$, les contraintes $\rho(M_i = N_i)$ sont trivialement vérifiées. On peut également remarquer que la transition CSLD est possible si et seulement si la transition SLD l'est car les contraintes $M_1 = N_1, \ldots, M_k = N_k$ sont satisfiables si et seulement si $\mathrm{mgu}(\mathtt{p}(N_1, \ldots, N_k), \mathtt{p}(M_1, \ldots, M_k))$ est défini. ◇

De manière plus générale dans CLP($\mathcal{X}$), si on note $\longrightarrow_\sigma$ les étapes de simplification et de substitution, le diagramme de la figure 2.1 commute :

Démonstration : De part la correction du solveur de contraintes, les étapes $\longrightarrow_\sigma$ transforment un ensemble de contraintes en un ensemble de contraintes équivalent. Ainsi, si $B_1 = c_1 \mid \vec{A_1}$ alors $B'_1 = c'_1 \mid \vec{A'_1}$, avec $\vec{A'_1} = \rho_1(\vec{A_1})$ et $c_1 \Leftrightarrow (c'_1, \{\mathtt{X} = \rho_1(\mathtt{X}) \mid \mathtt{X} \in V(B_1)\})$, où ρ_1 est la substitution appliquée par le solveur de contraintes. Supposons que $\vec{A_1} = \vec{A_{11}}, \mathtt{p}(M_1, \ldots, M_k), \vec{A_{12}}$ et que $B_1 \longrightarrow_{CSLD} B_2$ en utilisant la clause $\mathtt{p}(N_1, \ldots, N_k) \leftarrow d|\vec{A_p}$. On a $B_2 = c_2 \mid \vec{A_2}$,

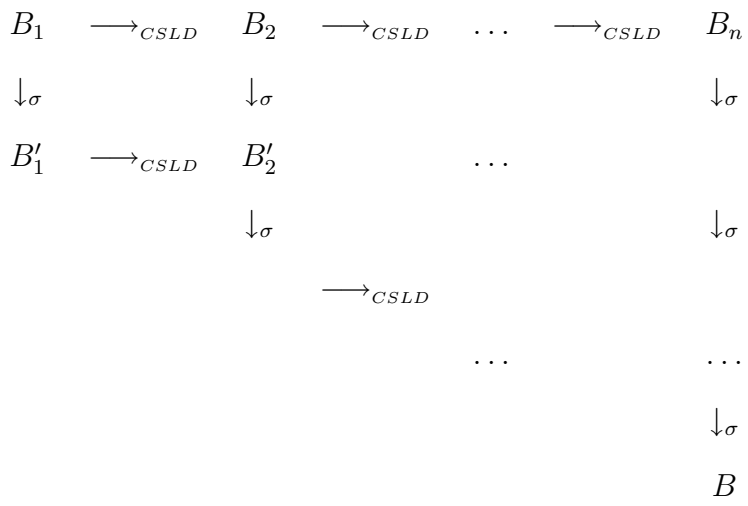


FIG. 2.1 – Réductions CSLD et étapes de simplification/substitutions

avec $\vec{A_2} = \vec{A_{11}}, \vec{A_p}, \vec{A_{12}}$ et $c_2 = (c_1, M_1 = N_1, \ldots, M_k = N_k, d)$ et c_2 est satisfiable. Comme c_2 est satisfiable et comme $c_1 \Leftrightarrow (c'_1, \{\mathtt{X} = \rho_1(\mathtt{X}) \mid \mathtt{X} \in V(B_1)\})$, la contrainte $c''_2 = c'_1, \{\mathtt{X} = \rho_1(\mathtt{X}) \mid \mathtt{X} \in V(B_1)\}, \rho_1(M_1) = N_1, \ldots \rho_1(M_k) = N_k, d$ est également satisfiable. Par hypothèse de correction du solveur, on a de plus $c'_2 = c'_1, \rho_1(M_1) = N_1, \ldots \rho_1(M_k) = N_k, d$ est satisfiable si et seulement si c''_2 l'est. Donc $B'_1 \longrightarrow_{CSLD} B'_2 = c'_2 \mid \vec{A'_2}$, avec $\vec{A'_2} = \rho_1(\vec{A_{11}}), \vec{A_p}, \rho_1(\vec{A_{12}})$.

En appliquant le raisonnement inverse, on obtient que si $B'_1 \longrightarrow_{CSLD} B'_2$, alors $B_1 \longrightarrow_{CSLD} B_2$.

Reste à montrer que $B_2 \longrightarrow_\sigma B'_2$. Par hypothèse, le solveur peut transformer c_1 en c'_1. Comme les variables de $\vec{A_p}$ sont fraîches, on a $\rho_1(\vec{A_{11}}), \vec{A_p}, \rho_1(\vec{A_{12}}) = \rho_1(\vec{A_{11}}), \vec{A_p}, \rho_1(\vec{A_{12}})$ et $\rho_1(M_1 = N_1, \ldots, M_K = N_k, d) = (\rho_1(M_1) = N_1, \ldots, \rho_1(M_K) = N_k, d)$. Donc $c_2 \longrightarrow_\sigma c'_2$ et $\rho_1(\vec{A_2}) = \vec{A'_2}$. D'où $B_2 \longrightarrow_\sigma B'_2$. □

Cela montre que le modèle d'exécution basé sur la résolution CSLD est un modèle d'exécution abstrait pour la programmation en logique avec contraintes.

2.3 Le langage CHR

Le langage des Constraint Handling Rules [25], (en abrégé CHR), est un langage de règles de réécriture destiné à l'implantation de solveurs de contraintes. Ce langage est utilisé à de nombreuses reprises pour l'implantation du logiciel TCLP, par exemple pour les contraintes de sous-typage ou pour la résolution des symboles surchargés. Les CHR sont en fait une extension de haut niveau d'un langage existant tel que Prolog ou Java, permettant d'écrire des solveurs de contraintes de

manière déclarative. Les CHR elles mêmes ne gèrent que les contraintes définies par l'utilisateur, tous les autres calculs, y compris les contraintes primitives comme l'unification, étant pris en charge par le langage hôte (dans notre cas Prolog).

Les CHR sont des règles gardées qui réécrivent des contraintes dans le but de les simplifier jusqu'à la résolution. Elles agissent selon deux principes : la *simplification* et la *propagation*. La simplification remplace certaines contraintes, tout en préservant l'équivalence logique du système. La propagation ajoute des contraintes redondantes, mais pouvant mener à d'autres simplifications.

Les CHR peuvent se présenter sous trois formes : simplification, propagation et *simpagation*. Cette dernière forme est en fait une combinaison des deux premières. Nous illustrons ces formes à travers l'exemple suivant, issu en grande partie de [25] :

Exemple 2.9 : Les règles suivantes définissent un solveur pour une contrainte `=<` dont les arguments peuvent être des variables :

```
reflexivity  @ X =< Y <=> X == Y | true.
antisymmetry @ X =< Y, Y =< X <=> X = Y.
transitivity @ X =< Y, Y =< Z ==> X =< Z.
identical    @ X =< Y \ X =< Y <=> true.
```

Le nom des règles est indiqué avant le symbole `@`. La règle `reflexivity` est une règle de simplification qui supprime la contrainte `X =< Y` (en la transformant en la contrainte `true`) lorsque la garde `X == Y` est vérifiée. La règle `antisymmetry` simplifie les deux contraintes `X =< Y` et `Y =< X` en la contrainte primitive `X = Y`. La règle `transitivity` est une règle de propagation qui ajoute la contrainte `X =< Z`, si les contraintes `X =< Y` et `Y =< Z` sont présentes dans l'ensemble de contraintes courant. Enfin la règle `identical` est une règle de simpagation : elle simplifie les contraintes comprises entre le `\` et le `<=>` et laisse inchangée les contraintes à gauche du `\`. Ici un des deux `X =< Y` disparaît (simplifié en `true`).◇

Première partie

Contraintes de sous-typage dans les quasi-treillis

Chapitre 3

Langage de types

Ce chapitre donne les hypothèses générales que nous allons considérer pour le langage des types. Nous nous plaçons dans le cadre des types infinis, les types réguliers et les types finis étant vus comme des cas particuliers de types infinis[1]. Les types sont des arbres étiquetés construits à partir d'un ensemble partiellement ordonné de symboles, appelés constructeurs de types. Ils sont ordonnés par une relation de sous-typage non structurelle non homogène basée sur un ordre partiel quelconque des constructeurs de types. Dans le chapitre 4, nous raffinerons ces hypothèses en nous concentrant sur les structures de quasi-treillis, pour lesquelles on peut décider des contraintes d'ordre.

3.1 Introduction au sous-typage

Le sous-typage est une notion fondamentale introduite par Cardelli [4] et Mitchell [47]. Elle est basée sur un ordre sur les types, appelé ordre de sous-typage. Son pouvoir réside dans la règle de subsomption (Sub). Elle exprime que si τ est un sous-type τ', c'est-à-dire plus petit dans l'ordre de sous-typage, alors une expression de type τ dans un environnement Γ peut être utilisée là où une expression de type τ' est attendue :

$$\text{(Sub)} \quad \frac{\Gamma \vdash t : \tau, \tau \leq \tau'}{\Gamma \vdash t : \tau'}$$

L'ordre sur les constructeurs de type va être utilisé pour engendrer l'ordre de sous-typage sur les types. Il peut être tout d'abord utilisé sur des types atomiques : si int $\leq$ float en tant que constructeurs de types, alors int $\leq$ float en tant que types. La relation s'étend aux types composés, par exemple list(int) $\leq$ list(float),

[1] Il faut donc comprendre la classe des types infinis comme l'ensemble des types "finis ou infinis".

ou, plus généralement, list(τ) $\leq$ list(τ') si et seulement si $\tau \leq \tau'$. De même, si list(.) $\leq$ set(.), alors list(int) $\leq$ set(int). Il se peut cependant que la relation entre certains arguments soit inversée. C'est en particulier le cas du constructeur $\rightarrow$ pour les types des fonctions. Une fonction capable de traiter des float en argument pourra être utilisée là où on attend une fonction capable de traiter des int. Inversement, une fonction renvoyant des int peut être utilisée là où on attend une fonction renvoyant des float. En résumé, on a float $\rightarrow$ int $\leq$ int $\rightarrow$ float, ou, plus généralement, $\tau_1 \rightarrow \tau_2 \leq \tau_1' \rightarrow \tau_2'$ si $\tau_1 \geq \tau_1'$ et $\tau_2 \leq \tau_2'$. On dit que le premier argument de $\rightarrow$ est *contravariant* et que sont deuxième argument est *covariant*.

On distingue différentes sortes de sous-typage, en fonction de caractéristiques sur l'ordre des constructeurs de types. Le sous-typage est dit structurel si l'ordre qui l'a engendré ne met en relation que des constructeurs n'ayant pas d'arguments. Par exemple on peut avoir int $\leq$ float, mais pas list(.) $\leq$ set(.). Le sous-typage est dit non structurel si l'ordre qui l'a engendré met en relation des constructeurs ayant des arguments. Il est dit homogène si l'ordre qui l'a engendré ne met en relation que des constructeurs ayant le même nombre d'arguments, comme par exemple list(.) $\leq$ set(.), mais pas list(.) $\leq$ term. L'utilisation du sous-typage pour typer des prédicats de méta-programmation, notamment pour les prédicats de décomposition des termes, nous conduit à utiliser un ordre sur les constructeurs incluant justement des relations de la forme list(.) $\leq$ term, c'est-à-dire engendrant un sous-typage non structurel non homogène.

3.2 Étiquettes

Dans le cas d'un sous-typage non structurel homogène, les arguments sont comparés en fonction de leur position dans le constructeur de type, c'est-à-dire $\kappa(\tau_1, \ldots, \tau_n) \leq \kappa'(\tau_1', \ldots, \tau_n')$ si et seulement si $\kappa \leq \kappa'$ et pour tout i, $1 \leq i \leq n$, $t_i \leq^i t_i'$, avec $\leq^i \equiv \leq$ si l'argument i de κ et κ' est covariant et $\leq^i \equiv \geq$ s'il est contravariant.

Exemple 3.1 : Considérons le constructeur de types $\rightarrow$. Son premier argument est contravariant et son deuxième argument est covariant. Ainsi, $\tau_1 \rightarrow \tau_2 \leq \tau_1' \rightarrow \tau_2'$ si et seulement si $\tau_1 \geq \tau_1'$ et $\tau_2 \leq \tau_2'$. $\diamond$

En revanche, dans le cas d'un sous-typage non homogène, il devient nécessaire d'exprimer explicitement la correspondance entre les arguments :

Exemple 3.2 : Supposons les constructeurs assoc à deux arguments covariants et set à un argument covariant, avec assoc $\leq$ set. A-t-on assoc(τ_1, τ_2) $\leq$ set(τ_3) si et seulement si :

– $\tau_1 \leq \tau_3$,

– ou bien $\tau_2 \leq \tau_3$? ◇

Afin d'exprimer cette correspondance, on peut imaginer introduire, pour chaque paire de constructeurs (κ, κ'), une fonction injective partielle des arguments de κ dans les arguments de κ', comme dans [22].

Exemple 3.3 : Reprenons l'exemple précédent. On peut imposer les arguments à comparer grâce à une fonction injective $\iota_{\text{assoc,set}}$, qui indique quel argument de assoc correspond à quel argument de set. Ainsi $\iota_{\text{assoc,set}}(2) = 1$ signifierait alors que $\text{assoc}(\tau_1, \tau_2) \leq \text{set}(\tau_3)$ si et seulement si $\tau_2 \leq \tau_3$. ◇

Bien que cette solution soit assez intuitive, elle introduit une grande lourdeur dans les notations et les concepts que nous serons amenés à développer. Nous allons donc lui préférer le formalisme d'étiquettes décrit par Pottier dans [57]. À chaque argument de chaque constructeur de type, on associe une étiquette, telle que pour tout constructeur, chacun de ses arguments ait une étiquette différente. Chaque argument peut ainsi être désigné par son étiquette en lieu et place de sa position. La correspondance se fait alors entre arguments ayant la même étiquette. Pour exprimer la covariance ou la contravariance des arguments, on peut annoter l'étiquette avec un signe : une étiquette l sera *positive* si elle correspond à des arguments covariants et *négative* si elle correspond à des arguments contravariants. On notera $\kappa[l_1, \ldots, l_n]$ le constructeur κ avec ses étiquettes énumérées de façon à ce que le i-ème argument du constructeur κ ait pour étiquette l_i.

Exemple 3.4 : Nous reprenons ici l'exemple précédent, cette fois ci en utilisant le formalisme des étiquettes. On a ainsi $\text{assoc}[a, d] \leq \text{set}[d]$, ce qui signifie que $\text{assoc}(\tau_1, \tau_2) \leq \text{set}(\tau_3)$ si et seulement si $\tau_2 \leq \tau_3$, puisque τ_2 et τ_3 correspondent tous deux à l'étiquette d commune à assoc et set. ◇

Habituellement, l'arité d'un symbole désigne son nombre d'arguments. Dans le formalisme des étiquettes, la notion d'arité se généralise en définissant l'arité d'un constructeur comme étant la liste des étiquettes servant à étiqueter ses arguments.

3.3 Types

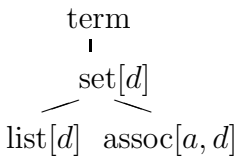
Dans cette section, nous définissons les types infinis de manière analogue à la définition des termes sous forme de fonctions [16]. Ainsi les types sont définis par rapport à une signature :

Définition 3.1 *Une* signature *est un quintuplet* $\mathcal{S} = (\mathcal{K}, \leq_{\mathcal{K}}, \mathcal{L}^+, \mathcal{L}^-, ar)$ *où :*
– $\mathcal{K}$ *est un ensemble dénombrable de* constructeurs de types.
– $\leq_{\mathcal{K}}$ *est un ordre partiel quelconque sur* $\mathcal{K}$.

- $\mathcal{L}$ *est un ensemble dénombrable d'*étiquettes *partitionné en deux sous-ensembles : l'ensemble des étiquettes positives* $\mathcal{L}^+$ *et l'ensemble des étiquettes négatives* $\mathcal{L}^-$.
- $ar : \mathcal{K} \to 2^{\mathcal{L}}$ *est la fonction qui à chaque constructeur de type associe son* arité. *Elle vérifie la condition suivante : pour tous constructeurs de types* κ_1, κ_2 *et* κ_3, *si* $\kappa_1 \leq_{\mathcal{K}} \kappa_2 \leq_{\mathcal{K}} \kappa_3$ *alors* $ar(\kappa_1) \cap ar(\kappa_3) \subseteq ar(\kappa_2)$.

On notera $ar^+(\kappa)$ l'ensemble des étiquettes positives dans l'arité de κ et $ar^-(\kappa)$ l'ensemble des étiquettes négatives dans l'arité de κ. Par la suite on supposera systématiquement que tous les constructeurs de type ont une arité finie.

Exemple 3.5 :
Soit $\mathcal{K} = \{\text{term}, \text{set}, \text{list}, \text{assoc}\}$, ordonné comme ci-contre. Soit $\mathcal{L}^+ = \{a, d\}$ et $\mathcal{L}^- = \emptyset$. Enfin soit ar est définie par : $ar(\text{term}) = \emptyset$, $ar(\text{set}) = ar(\text{list}) = \{d\}$ et $ar(\text{assoc}) = \{a, d\}$. On vérifie aisément la condition de la définition 3.1 ci-dessus.



◇

L'ordre $\leq_{\mathcal{K}}$ sur les constructeurs de types est un ordre partiel quelconque qui sera utilisé dans la section suivante pour construire l'ordre sur les types, qui est une relation de sous-typage non structurel non homogène. La condition sur l'arité exprime la cohérence des étiquettes par rapport à cette relation de sous-typage. Dans le cas où cette condition n'est pas vérifiée, la relation induite par l'ordre sur les constructeurs ne serait pas un ordre sur les types. Par exemple si $\kappa_1[l] \leq_{\mathcal{K}} \kappa_2 \leq_{\mathcal{K}} \kappa_3[l]$, et si τ et τ' sont incomparables, on a $\kappa_1(\tau) \leq \kappa_2$ et $\kappa_2 \leq \kappa_3(\tau')$ mais $\kappa_1(\tau) \not\leq \kappa_3(\tau')$.

Tout comme les termes sont définis comme des fonctions partielles sur des chaînes de nombres [16, 52], les types sont définis comme des fonctions partielles sur des chaînes d'étiquettes. Étant donné une signature $(\mathcal{K}, \leq_{\mathcal{K}}, \mathcal{L}^+, \mathcal{L}^-, ar)$, on note $\mathcal{L}^*$ l'ensemble des suites finies d'étiquettes. On notera $|w|$ la taille de la suite w, ϵ la suite vide et "." l'opération de concaténation des suites.

Un *type infini*, également appelé simplement *type*, est ainsi un terme formé sur l'ensemble de symboles $\mathcal{K}$ dans lequel les positions sont des séquences d'étiquettes, ou, plus formellement :

Définition 3.2 *Un* type infini t *construit sur une signature* $\mathcal{S} = (\mathcal{K}, \leq_{\mathcal{K}}, \mathcal{L}^+, \mathcal{L}^-, ar)$ *est une fonction partielle de* $\mathcal{L}^*$ *dans* $\mathcal{K}$ *telle que :*

1. *Son domaine est clos par préfixe : pour toutes positions* $w_1, w_2 \in \mathcal{L}^*$, *si* $w_1.w_2 \in dom(\tau)$ *alors* $w_1 \in dom(\tau)$.
2. *La chaîne vide* ϵ *est dans le domaine de* τ [2].

[2] ou de manière équivalente, $dom(\tau) \neq \emptyset$.

3. *Pour toute position* $w \in dom(\tau)$, *si* $\tau(w) = \kappa$ *alors pour toute étiquette* $l \in \mathcal{L}$, $w.l \in dom(\tau) \Leftrightarrow l \in ar(\kappa)$.

On note $\mathcal{T}(\mathcal{S})$ l'ensemble des types infinis construits sur $\mathcal{S}$. Pour toute position $w \in dom(\tau)$, on note τ/w le type $\tau' : v \mapsto \tau(w.v)$. On dit qu'un type τ' est un *sous-terme* d'un type τ s'il existe une position w telle que $\tau' = \tau/w$. Un type est dit *régulier* [16] s'il a un nombre fini de sous-termes. On notera $\mathcal{R}(\mathcal{S})$ l'ensemble des types réguliers construits sur $\mathcal{S}$. Un type est dit *fini* s'il a un domaine fini. On notera $\mathcal{F}(\mathcal{S})$ l'ensemble des types finis construits sur $\mathcal{S}$. On omettra de préciser la signature $\mathcal{S}$ lorsqu'il n'y aura pas d'ambiguïté. Par abus de notation, on notera également $\mathcal{T}(\mathcal{K})$ l'ensemble des types infinis d'une signature $\mathcal{S} = (\mathcal{K}, \leq_{\mathcal{K}}, \mathcal{L}^+, \mathcal{L}^-, ar)$.

Il est possible de définir un type τ à partir de sa valeur en ϵ et de ses sous-termes immédiats : soit κ un constructeur de type et soient les types τ_l pour toute étiquette $l \in ar(\kappa)$. Alors on peut définir τ en posant $\tau(\epsilon) = \kappa$ et pour toute position $l.w$ avec $l \in ar(\kappa)$, $\tau(l.w) = \tau_l(w)$. Il est facile de vérifier que τ est un type. Cette construction nous sera particulièrement utile dans les chapitres sur la résolution des contraintes de sous-typage.

Exemple 3.6 : Considérons la signature donnée dans l'exemple 3.5. Soit le type τ défini par : $\tau(\epsilon) = \text{assoc}$, $\tau(a) = \text{term}$, $\tau(d) = \text{list}$ et $\tau(dd) = \text{term}$. τ/a est le type $\tau' : \epsilon \mapsto \text{term}$. τ/d est le type τ'' défini par : $\tau''(\epsilon) = \text{list}$ et $\tau''(d) = \text{term}$.

Á l'inverse, on peut définir τ à partir de ses "composants", en posant $\tau(\epsilon) = \text{assoc}$, $\tau/a = \tau'$ et $\tau/d = \tau''$. ◇

On introduit également la notation suivante, plus intuitive, pour les types : si $\kappa[l_1, \ldots, l_n]$ est un constructeur de type, donné avec ses étiquettes, alors on note $\kappa(\tau_1, \ldots, \tau_n)$ le type τ défini par $\tau(\epsilon) = \kappa$ et $\tau/l_i = \tau_i$.

Exemple 3.7 : En reprenant les types donnés dans l'exemple précédent, si on donne les constructeurs assoc et list avec leur étiquettes comme suit : assoc$[a, d]$ et list$[d]$, on obtient : $\tau = \text{assoc}(\text{term}, \text{list}(\text{term}))$, $\tau' = \text{term}$ et $\tau'' = \text{list}(\text{term})$.◇

3.4 Ordre sur les types

Cette section, reprise de [57], définit l'ordre de sous-typage. Nous supposerons que la signature $(\mathcal{K}, \leq_{\mathcal{K}}, \mathcal{L}^+, \mathcal{L}^-, ar)$ est fixée. L'ordre $\leq$ sur les types infinis est défini par co-induction. On introduit à cette fin la suite $(\leq_n)_n$ de relations définie comme suit :

- $\leq_0 = \mathcal{T} \times \mathcal{T}$
- On pose $\tau \leq_{n+1} \tau'$ si $\tau(\epsilon) \leq_{\mathcal{K}} \tau'(\epsilon)$ et
  - pour toute étiquette $l \in ar^+(\tau(\epsilon)) \cap ar^+(\tau'(\epsilon))$, $\tau/l \leq_n \tau'/l$,

– pour toute étiquette $l \in ar^-(\tau(\epsilon)) \cap ar^-(\tau'(\epsilon))$, $\tau'/l \leq_n \tau/l$.

Lemme 3.3 *Pour tout $n \in \mathbb{N}$, $\leq_n$ est un pré-ordre.*

Démonstration : Par récurrence sur n. C'est vrai pour $n = 0$. Montrons le cas $n + 1$:

Transitivité. Soit $\tau_1 \leq_{n+1} \tau_2$ et $\tau_2 \leq_{n+1} \tau_3$. On pose $\kappa_i = \tau_i(\epsilon)$. On a $\kappa_1 \leq_{\mathcal{K}} \kappa_2$ et $\kappa_2 \leq_{\mathcal{K}} \kappa_3$, d'où $\kappa_1 \leq_{\mathcal{K}} \kappa_3$. Soit une étiquette $l \in ar(\kappa_1) \cap ar(\kappa_3)$. Comme $\kappa_1 \leq \kappa_2 \leq \kappa_3$, $l \in ar(\kappa_2)$. Si l est positive, par définition de $\leq_{n+1}$, $\tau_1/l \leq_n \tau_2/l$ et $\tau_2/l \leq_n \tau_3/l$. Par récurrence, on obtient $\tau_1/l \leq_n \tau_3/l$. De même, si l est négative, on obtient $\tau_3/l \leq_n \tau_1/l$. Donc $\tau_1 \leq_{+1} \tau_3$.

Réflexivité. Soit un type $\tau \in \mathcal{T}$. Par récurrence, pour toute étiquette $l \in ar(\tau(\epsilon))$, $\tau/l \leq_n \tau/l$. De plus $\tau(\epsilon) \leq_{\mathcal{K}} \tau(\epsilon)$, D'où $\tau \leq_{n+1} \tau$. □

Définition 3.4 *La relation $\leq$ sur les types est l'intersection des pré-ordres $(\leq_n)$:*

$$\leq = \bigcap_{n \in \mathbb{N}} \leq_n$$

Propriété 3.5 *$\leq$ est un ordre partiel sur $\mathcal{T}$.*

Démonstration :

Transitivité. Soit trois types $\tau_1, \tau_2, \tau_3 \in \mathcal{T}$ tels que $\tau_1 \leq \tau_2$ et $\tau_2 \leq \tau_3$. Pour tout $n \in \mathbb{N}$, $\tau_1 \leq_n \tau_2$ et $\tau_2 \leq_n \tau_3$, donc $\tau_1 \leq_n \tau_3$., D'où $\tau_1 \leq \tau_3$.

Réflexivité. Soit un type $\tau \in \mathcal{T}$. Pour tout $n \in \mathbb{N}$, $\tau \leq_n \tau$, donc $\tau \leq \tau$.

Antisymétrie. Soient deux types $\tau_1, \tau_2 \in \mathcal{T}$. On montre tout d'abord, par récurrence, que pour tout $n \in \mathbb{N}$, pour tous types $\tau_1, \tau_2 \in \mathcal{T}$, pour toute position $w \in dom(\tau_1)$, si $\tau_1 \leq_{n+1} \tau_2$, $\tau_2 \leq_{n+1} \tau_1$ et $|w| \leq n$, $w \in dom(\tau_2)$ et $\tau_1(w) = \tau_2(w)$. Considérons le cas $n = 0$, c'est-à-dire $w = \epsilon$. Par définition, $\epsilon \in dom(\tau_2)$. Comme $\tau_1 \leq_1 \tau_2$ et $\tau_2 \leq_1 \tau_1$, $\tau_1(\epsilon) \leq_{\mathcal{K}} \tau_2(\epsilon) \leq_{\mathcal{K}} \tau_1(\epsilon)$, d'où $\tau_1(\epsilon) = \tau_2(\epsilon)$. Considérons à présent le cas $n + 1$. Si $w = \epsilon$, la démonstration est similaire au cas $n = 0$. Sinon, $w = l.w'$ pour un certain w'. On pose $\kappa = \tau_1(\epsilon) = \tau_2(\epsilon)$. Par définition, $l \in ar(\kappa)$ et $\tau_1/l \leq_n \tau_2/l \leq_n \tau_1/l$. Comme $|w'| \leq n - 1$ et $w' \in dom(\tau_1/l)$, on obtient $w' \in dom(\tau_2/l)$, donc $w \in dom(\tau_2)$. De plus, $\tau_2(w) = \tau_2/l(w') = \tau_1/l(w') = \tau_1(w)$.

Soit deux types $\tau_1, \tau_2 \in \mathcal{T}$ tels que $\tau_1 \leq \tau_2 \leq \tau_1$. Si $\tau_1 \neq \tau_2$, alors il existe une position $w \in \mathcal{L}^*$ de taille minimale telle que $\tau_1(w) \neq \tau_2(w)$. Cependant $\tau_1 \leq_{|w|+1} \tau_2 \leq_{|w|+1} \tau_1$. Donc $\tau_1(w) = \tau_2(w)$, ce qui contredit $\tau_1 \neq \tau_2$. □

La propriété suivante caractérise la relation de sous-typage de manière plus intuitive :

Propriété 3.6 *Pour tous types $\tau_1, \tau_2 \in \mathcal{T}$, $\tau_1 \leq \tau_2$ si et seulement si :*
- $\tau_1(\epsilon) \leq_{\mathcal{K}} \tau_2(\epsilon)$.
- *pour toute étiquette* $l \in ar^+(\tau_1(\epsilon)) \cap ar^+(\tau_2(\epsilon))$, $\tau_1/l \leq \tau_2/l$.
- *pour toute étiquette* $l \in ar^-(\tau_1(\epsilon)) \cap ar^-(\tau_2(\epsilon))$, $\tau_2/l \leq \tau_1/l$.

Démonstration : Supposons $\tau_1 \leq \tau_2$. Comme $\tau_1 \leq_1 \tau_2$, on obtient $\tau_1(\epsilon) \leq_{\mathcal{K}} \tau_2(\epsilon)$. De plus, pour tout $n \in \mathbb{N}$, $\tau_1 \leq_{n+1} \tau_2$. Donc pour toute étiquette $l \in ar^+(\tau_1(\epsilon)) \cap ar^+(\tau_2(\epsilon))$, pour tout $n \in \mathbb{N}$, $\tau_1/l \leq_n \tau_2/l$, d'où $\tau_1/l \leq \tau_2/l$. De même, pour toute étiquette $l \in ar^-(\tau_1(\epsilon)) \cap ar^-(\tau_2(\epsilon))$, $\tau_2/l \leq \tau_1/l$.

Montrons à présent la réciproque. Pour tout $n \in \mathbb{N}$, pour toute étiquette $l \in ar^+(\tau_1(\epsilon)) \cap ar^+(\tau_2(\epsilon))$, $\tau_1/l \leq_n \tau_2/l$ et pour toute étiquette $l \in ar^-(\tau_1(\epsilon)) \cap ar^-(\tau_2(\epsilon))$, $\tau_2/l \leq_n \tau_1/l$, d'où $\tau_1 \leq_{n+1} \tau_2$. De plus, on a trivialement $\tau_1 \leq_0 \tau_2$. Donc $\tau_1 \leq \tau_2$. □

Exemple 3.8 : L'ordre de sous-typage engendré par la signature de l'exemple 3.5 donne, entre autres :
- $\text{list}(\text{term}) \leq \text{term}$ (car $\text{list} \leq_{\mathcal{K}} \text{term}$),
- $\text{set}(\text{list}(\text{term})) \leq \text{set}(\text{term})$ (car $\text{list}(\text{term}) \leq \text{term}$),
- $\text{list}(\text{term}) \leq \text{set}(\text{term})$ (car $\text{list} \leq_{\mathcal{K}} \text{set}$),
- $\text{assoc}(\text{term}, \text{list}(\text{term})) \leq \text{set}(\text{set}(\text{term}))$ (car $\text{assoc} \leq_{\mathcal{K}} \text{set}$ et $\text{list}(\text{term}) \leq \text{set}(\text{term})$).

◇

3.5 Contraintes de sous-typage

Soit $\mathcal{V}$ un ensemble dénombrable de variables de types ou *paramètres*, notés $\alpha, \beta, \ldots$. On définit la signature $\mathcal{S}_{\mathcal{V}} = (\mathcal{K} \cup \mathcal{V}, \leq_{\mathcal{K}} \cup =_{\mathcal{V}}, \mathcal{L}^+, \mathcal{L}^-, ar_{\mathcal{K}\cup\mathcal{V}})$, où id désigne l'identité et $ar_{\mathcal{K}\cup\mathcal{V}}(\kappa) = ar(\kappa)$ pour les constructeurs $\kappa \in \mathcal{K}$, $ar_{\mathcal{K}\cup\mathcal{V}}(\alpha) = \emptyset$ pour les paramètres $\alpha \in \mathcal{V}$. Par abus de notation, on notera $\tau \in \mathcal{V}$ pour $\tau(\epsilon) = \alpha$ et $\alpha \in \mathcal{V}$. On note $\mathcal{F}_{\mathcal{V}} = \mathcal{F}(\mathcal{S}_{\mathcal{V}})$ l'ensemble des types finis construits à partir de $\mathcal{S}_{\mathcal{V}}$.

Une *contrainte de sous-typage* est de la forme $\tau_1 \leq \tau_2$, où $\tau_1, \tau_2 \in \mathcal{F}_{\mathcal{V}}$ sont deux types *finis*. Si C est un ensemble de contraintes de sous-typage, encore appelé *système* de contraintes de sous-typage, on note $V(C)$ l'ensemble des paramètres apparaissant dans C.

Exemple 3.9 : Considérons la signature $\mathcal{S}_{\mathcal{V}}$ définie à partir de la signature de l'exemple 3.5. On peut définir le système de contraintes $C = \text{list}(\alpha) \leq \text{set}(\alpha), \text{assoc}(\alpha, \beta) \leq \text{set}(\beta), \beta \leq \text{term}, \alpha \leq \beta$. Dans ce cas, $V(C) = \{\alpha, \beta\}$. ◇

Une *substitution de types* ρ est une fonction partielle de $\mathcal{V}$ dans $\mathcal{T}$. Elle peut être étendue classiquement sur $\mathcal{F}(\mathcal{S}_{\mathcal{V}})$ en définissant pour tout type τ, $\rho(\tau)(w)$ comme suit :

- Si $\tau(\epsilon) \in \mathcal{V}$ alors $\rho(\tau)(w) = \rho(\tau(\epsilon))(w)$
- Si $\tau(\epsilon) \in \mathcal{K}$ alors si $w = \epsilon$, $\rho(\tau)(\epsilon) = \tau(\epsilon)$ et si $w = l.w'$, $\rho(\tau)(l.w') = \rho(\tau/l)(w')$.

Exemple 3.10 : Considérons le type $\tau = \text{assoc}(\alpha, \text{list}(\beta))$ et la substitution $\rho : \alpha \mapsto \text{term}$, $\beta \mapsto \text{set}(\text{term})$. On a $\rho(\tau) = \text{assoc}(\text{term}, \text{list}(\text{set}(\text{term})))$. ◇

Propriété 3.7 *Soient deux types $\tau_1, \tau_2 \in \mathcal{F}(\mathcal{S}_\mathcal{V})$. Soit ρ une substitution de types. Si $\tau_1 \leq \tau_2$ alors $\rho(\tau_1) \leq \rho(\tau_2)$.*

Démonstration : On montre que pour tout $n \in \mathbb{N}$, $\tau_1 \leq_n \tau_2$. Le cas $n = 0$ est trivial. Considérons le cas $n + 1$. Comme $\tau_1 \leq \tau_2$, $\tau_1(\epsilon) \leq_\mathcal{K} \tau_2(\epsilon)$. Si $\tau_1(\epsilon) = \alpha \in \mathcal{V}$ ou si $\tau_2(\epsilon) = \alpha \in \mathcal{V}$, alors on a forcément $\tau_1(\epsilon) = \tau_2(\epsilon) = \alpha$. Donc $\rho(\tau_1) = \rho(\tau_2)$, d'où $\rho(\tau_1) \leq_{n+1} \rho(\tau_2)$. Sinon, pour toute étiquette $l \in ar(\tau_1(\epsilon)) \cap ar(\tau_2(\epsilon))$, par récurrence, si $l \in \mathcal{L}^+$ alors $\rho(\tau_1)/l \leq_n \rho(\tau_2)/l$ et si $l \in \mathcal{L}^-$ alors $\rho(\tau_2)/l \leq_n \rho(\tau_1)/l$. D'où $\rho(\tau_1) \leq_{n+1} \rho(\tau_2)$. □

La définition suivante décrit la notion de satisfaction d'une contrainte de sous-typage :

Définition 3.8 *Une substitution $\rho : \mathcal{V} \to \mathcal{T}$* satisfait *la contrainte $\tau_1 \leq \tau_2$, noté $\rho \models \tau_1 \leq \tau_2$, si $\rho(\tau_1) \leq \rho(\tau_2)$. Elle satisfait un système de contraintes C, noté $\rho \models C$, si elle satisfait toutes les contraintes $c \in C$.*

Un système de contraintes C est dit satisfiable *s'il existe une substitution ρ qui satisfait C.*

Exemple 3.11 : Considérons le système de contraintes C défini dans l'exemple 3.9. La substitution $\rho : \alpha \mapsto \text{list}(\text{term})$, $\beta \mapsto \text{set}(\text{term})$ satisfait C car on a bien dans la relation de sous-typage :

$$\text{list}(\text{list}(\text{term})) \leq \text{set}(\text{list}(\text{term})),$$

$$\text{assoc}(\text{list}(\text{term}), \text{set}(\text{term})) \leq \text{set}(\text{set}(\text{term})),$$

$$\text{set}(\text{term}) \leq \text{term},$$

$$\text{list}(\text{term}) \leq \text{set}(\text{term}).$$

On peut remarquer que, de par la propriété 3.7, toutes substitutions définies sur α et β satisfont le système de contraintes $C' = \text{list}(\alpha) \leq \text{set}(\alpha), \text{assoc}(\alpha, \beta) \leq \text{set}(\beta)$. ◇

On définit également une notion plus faible de satisfaction, qui sera utilisée dans le preuves des algorithmes de test de satisfiabilité de contraintes des chapitres 5 et 6 :

Définition 3.9 *Une substitution* $\rho : \mathcal{V} \to \mathcal{T}$ satisfait au rang n *la contrainte* $\tau_1 \leq \tau_2$, *noté* $\rho \models_n \tau_1 \leq \tau_2$, *si* $\rho(\tau_1) \leq_n \rho(\tau_2)$. *Elle satisfait au rang n un système de contraintes C, noté* $\rho \models_n C$, *si elle satisfait au rang n toutes les contraintes* $c \in C$.

On peut remarquer que $\rho \models C$ si et seulement si pour tout $n \in \mathbb{N}$, $\rho \models_n C$.

Types plats

L'étude de la satisfiabilité des contraintes de sous-typage dans les quasi-treillis fait l'objet des chapitres 5 et 6. Dans ces chapitres, nous ferons l'hypothèse que les contraintes de sous-typage sont construites à partir de types plats. La notion de types, ou termes, plats est une notion classique en théorie de l'unification [37], et est utilisée dans différents travaux sur le typage comme par exemple la thèse de Pottier [56] où ils sont appelés petits termes. Un *type plat* est soit un paramètre, soit une constante, soit un type de profondeur 1 et dont toutes les feuilles sont des paramètres. Par exemple, int, list(α) et α sont des types plats, mais list(int) n'en est pas un. On peut supposer, sans perte de généralité, que les contraintes sont construites à partir de types plats : il est en effet possible de trouver à tout système de contraintes de sous-typage un système de contraintes équivalent et construit à partir de types plats. Il suffit pour cela d'introduire des variables existentiellement quantifiées pour les arguments de chaque type qui n'est pas un type plat ainsi que des contraintes d'égalités (ou des doubles inégalités) entre ces variables et les arguments correspondants.

Chapitre 4

Quasi-treillis de types

De la structure sous-jacente à la signature des constructeurs de types dépendent à la fois l'expressivité du langage des types et la complexité de la résolution des contraintes de sous-typage. Dans le chapitre précédent, nous avons supposé que l'ordre $\leq_{\mathcal{K}}$ sur les constructeurs de types était un ordre partiel quelconque. Cependant, la décidabilité de la satisfiabilité des contraintes de sous-typage non structurel non homogène basé sur un ordre partiel quelconque est un problème ouvert, ce qui nous conduit à étudier une restriction de cet ordre.

Dans le cas des types finis, munis d'un sous-typage non structurel homogène issu d'un ordre partiel quelconque, Frey [24] a montré que le problème de la satisfiabilité des contraintes de sous-typage était PSPACE-complet. Dans le cadre de types infinis, munis d'un sous-typage non structurel non homogène, et si l'on restreint la structure des constructeurs de types à un treillis, Trifonov et Smith [70] on exhibé un algorithme pour tester la satisfiabilité en $O(n^3)$, où n est la taille du système de contraintes. Cependant, aucune de ces deux solutions n'est réellement adaptée à nos besoins. En effet, le sous-typage homogène ne permet pas de relations telles que $\text{list}(\alpha) \leq \text{term}$, que nous utilisons pour le typage de la métaprogrammation. D'un autre côté, les structures de treillis imposent le type $\bot$, qui est un type vide. De plus, ce type est toujours un type possible pour les variables dans le système que nous décrivons au chapitre 7, ce qui est particulièrement problématique pour l'inférence de type de ces variables.

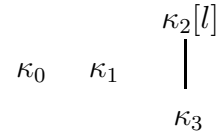
Dans cette thèse, nous nous intéressons aux structures de quasi-treillis engendrant un sous-typage non structurel non homogène. Dans ces structures, un ensemble de types a une borne supérieure (resp. inférieure) si et seulement si il est majoré. Contrairement aux treillis, ces structures ne nécessitent pas de type maximal $\top$ ou de type minimal $\bot$. Utiliser une structure de quasi-treillis évite donc d'introduire artificiellement $\bot$. De plus, comme nous le verrons dans le chapitre 6, il ne suffit pas vérifier son absence dans les types inférés pour pouvoir conclure à une erreur de type, à cause du phénomène d'"oubli" d'argument illustré dans

l'exemple 4.1.

Dans ce chapitre, nous allons étudier un certain nombre de propriétés des quasi-treillis de constructeurs de types et des structures de types infinis générées par ces derniers. Ces propriétés s'avéreront utiles pour résoudre les contraintes de sous-typage dans les quasi-treillis. En particulier, nous introduisons la notion de signature *bien formée,* qui correspond à des conditions suffisantes pour qu'un quasi-treillis de constructeurs de types engendre un quasi-treillis de types, ce qui est exprimé dans le théorème 4.10. Nous terminons ce chapitre en étudiant les structures de types réguliers et les structures de types finis issues de signatures bien formées.

Une particularité du sous-typage non structurel non homogène dans des quasi-treillis est l'"oubli" d'argument dans la borne inférieure (ou supérieure) d'un ensemble. Nous illustrons cette particularité dans l'exemple suivant :

Exemple 4.1 : Considérons la signature $(\mathcal{K}, \leq_{\mathcal{K}}, \mathcal{L}^+, \mathcal{L}^-, ar)$, avec $\mathcal{K} = \{\kappa_0, \kappa_1, \kappa_2, \kappa_3\}$, avec $\kappa_3 \leq_{\mathcal{K}} \kappa_2$, $\mathcal{L}^+ = \{l\}$, $\mathcal{L}^- = \emptyset$ et $ar(\kappa_2) = \{l\}$ correspondant au schéma ci-contre. Les types $\kappa_2(\kappa_0)$ et $\kappa_2(\kappa_1)$ ont une borne inférieure κ_3, bien que les sous-termes κ_0 et κ_1 n'en aient pas. Il ont en fait simplement disparus dans cette borne, c'est-à-dire la borne inférieure de $\kappa_2(\kappa_0)$ et $\kappa_2(\kappa_1)$ les a "oubliés". Plus précisément le constructeur de cette borne ne possède pas d'étiquette correspondant à ces arguments litigieux.



◇

La principale difficulté liée à ce phénomène d'"oubli" d'argument est que, contrairement aux cas des structures plus régulières que sont les treillis, le constructeur de tête de la borne inférieure de deux types ne dépend pas que des constructeurs de tête de ces types, mais également des sous-termes de ces types.

4.1 Préliminaires

Nous introduisons ici quelques notations et définitions concernant les quasi-treillis. Soit $(E, \leq)$ un ensemble partiellement ordonné. Soit S un sous-ensemble de E. L'ensemble des *minorants* de S, noté $\downarrow S$, est l'ensemble des éléments de E qui sont plus petits que tous les éléments de S ($\downarrow S = \{e \in E \mid \forall s \in S, e \leq s\}$). L'ensemble des majorants de S, noté $\uparrow S$, est l'ensemble des éléments de E qui sont plus grands que tous les éléments de S ($\uparrow S = \{e \in E \mid \forall s \in S, s \leq e\}$). On dit que S est minoré si $\downarrow S \neq \emptyset$ et majoré si $\uparrow S \neq \emptyset$. La *borne inférieure* de S, notée $\sqcap S$, est son plus grand minorant s'il existe et la *borne supérieure* de S, notée $\sqcup S$, est son plus petit majorant s'il existe.

Définition 4.1 *Un* sup-quasi-treillis *est un ensemble partiellement ordonné dans lequel toute paire d'éléments majorée admet une borne supérieure. Un* inf-quasi-treillis *est un ensemble partiellement ordonné dans lequel toute paire d'éléments minorée admet une borne inférieure. Un* quasi-treillis *est un sup-quasi-treillis et un inf-quasi-treillis.*

On peut remarquer qu'un sup-quasi-treillis fini est un quasi-treillis.

Nous définissons à présent les *quasi-treillis complets* au sens des *ensembles*, par opposition au sens des *chaînes*, habituellement utilisé pour les treillis [2].

Définition 4.2 *Un* sup-quasi-treillis complet *(au sens des ensembles) (resp.* inf-quasi-treillis complet*) est un ensemble partiellement ordonné $(E, \leq)$ tel que tout sous ensemble non vide $S \subseteq E$ majoré (resp. minoré) admet une borne supérieure (resp. inférieure). Un* quasi-treillis complet *est sup-quasi-treillis complet et un inf-quasi-treillis complet.*

On peut remarquer que tout quasi-treillis fini est complet.

4.2 Constructeurs de types

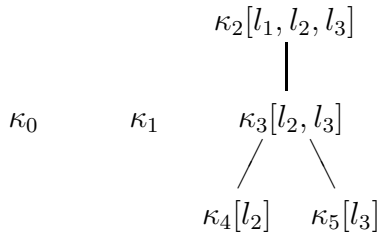
Dans cette section nous définissons la notion de signature bien formée, qui est une condition suffisante pour engendrer un quasi-treillis complet de types infinis. Nous définissons également une notion de majorants et de minorants relatifs à un ensemble d'étiquettes. Cette notion sera utile pour la construction des bornes inférieures et supérieures dans le quasi-treillis des types.

Définition 4.3 *Une signature $(\mathcal{K}, \leq_{\mathcal{K}}, \mathcal{L}^+, \mathcal{L}^-, ar)$ est dite* bien formée *si :*

1. *$(\mathcal{K}, \leq_{\mathcal{K}})$ est un quasi-treillis complet.*
2. *Pour tout sous-ensemble $S \subseteq \mathcal{K}$, si $\sqcap S$ existe alors $ar(\sqcap S) \subseteq \bigcup_{s \in S} ar(S)$, et si $\sqcup S$ existe alors $ar(\sqcup S) \subseteq \bigcup_{s \in S} ar(S)$.*
3. *Pour toute paire de constructeurs $\kappa_1 \leq_{\mathcal{K}} \kappa_2$, il existe un constructeur κ tel que $\kappa_1 \leq_{\mathcal{K}} \kappa \leq_{\mathcal{K}} \kappa_2$ et $ar(\kappa) = ar(\kappa_1) \cap ar(\kappa_2)$.*

L'exemple suivant nous servira à illustrer les différentes notions que nous introduirons dans la suite de ce chapitre :

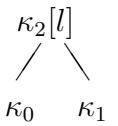
Exemple 4.2 : Soit la signature $\mathcal{S} = (\mathcal{K}, \leq_{\mathcal{K}}, \mathcal{L}^+, \mathcal{L}^-, ar)$, avec $\mathcal{L}^+ = \{l_1, l_2, l_3\}$, $\mathcal{L}^- = \emptyset$, $\mathcal{K}$, ar et $\leq_{\mathcal{K}}$ étant donnés dans la figure ci-dessous :



$\mathcal{S}$ constitue un exemple de signature bien formée. ◇

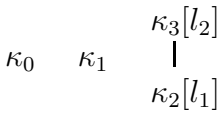
Les deux exemples ci-dessous illustrent les rôles respectivement de la deuxième et de la troisième conditions :

Exemple 4.3 :
Considérons $\mathcal{K} = \{\kappa_0, \kappa_1, \kappa_2\}$, $\kappa_0 \leq_{\mathcal{K}} \kappa_2$, $\kappa_1 \leq_{\mathcal{K}} \kappa_2$. Supposons également $\mathcal{L}^+ = \{l\}$, $\mathcal{L}^- = \emptyset$, $ar(\kappa_0) = ar(\kappa_1) = \emptyset$ et $ar(\kappa_2) = l$. La paire $\{\kappa_0, \kappa_1\}$ est majorée par $\kappa_2(\kappa_0)$ et par $\kappa_2(\kappa_1)$ mais n'admet pas de borne supérieure.



◇

Exemple 4.4 :
Considérons $\mathcal{K} = \{\kappa_0, \kappa_1, \kappa_2, \kappa_3\}$, $\kappa_2 \leq_{\mathcal{K}} \kappa_3$. Supposons également $\mathcal{L}^+ = \{l_1, l_2\}$, $\mathcal{L}^- = \emptyset$, $ar(\kappa_0) = ar(\kappa_1) = \emptyset$, $ar(\kappa_2) = \{l_1\}$, $ar(\kappa_3) = \{l_2\}$. La paire $\{\kappa_2(\kappa_0), \kappa_2(\kappa_1)\}$ est majorée par $\kappa_3(\kappa_0)$ et $\kappa_3(\kappa_1)$, mais n'admet pas de borne supérieure.



◇

Dans le cadre des treillis, Pottier [57] utilise la notion de signature "raisonnable", similaire à la notion de signature bien formée. Pour être que la signature soit raisonnable, $(\mathcal{K}, \leq_{\mathcal{K}})$ doit être un treillis complet et elle doit respecter la deuxième condition de la définition 4.3. Cette condition n'est cependant pas absolument nécessaire pour obtenir un treillis de types, mais est demandée afin de simplifier les algorithmes de résolution des contraintes de sous-typage. L'exemple 4.3 montre qu'elle est au contraire importante pour obtenir un quasi-treillis de type à partir d'un quasi-treillis de constructeurs de types. Enfin, la troisième condition est complètement spécifique aux quasi-treillis.

Dans la suite de ce chapitre, nous considérerons une signature $(\mathcal{K}, \leq_{\mathcal{K}}, \mathcal{L}^+, \mathcal{L}^-, ar)$ bien formée fixée. On introduit à présent le concept de minorants et de majorants d'un constructeur par rapport à un ensemble d'étiquettes. Ce sont des minorants et des majorants dont les étiquettes sont restreintes, plus précisément :

Définition 4.4 *L'ensemble $\downarrow_L \kappa$ (resp. $\uparrow_L \kappa$) des* minorants *(resp.* majorants*)* de κ par rapport à L *est l'ensemble des minorants (resp. majorants) κ' de κ tels que $ar(\kappa') \cap ar(\kappa) \subseteq L$.*

Exemple 4.5 : En reprenant la signature de l'exemple 4.2, on a $\downarrow_{\{l_2,l_3\}}\kappa_2 = \{\kappa_3, \kappa_4, \kappa_5\}$. Par contre $\uparrow_{\{l_3\}}\kappa_3 = \emptyset$. ◇

Le lemme suivant énonce l'existence d'un maximum pour ces minorants, et caractérise son arité. En particulier une conséquence de ce lemme est que $ar(\sqcup(\downarrow_L\kappa)) \subseteq ar(\kappa) \cap L$ (resp. $ar(\sqcap(\uparrow_L\kappa)) \subseteq ar(\kappa) \cap L$), ce qui en consituera la principale utilisation.

Lemme 4.5 *Soient $L \subseteq \mathcal{L}$ un ensemble d'étiquettes et $\kappa \in \mathcal{K}$ un constructeur de types. Si $\downarrow_L\kappa \neq \emptyset$ alors il possède un maximum $\sqcup(\downarrow_L\kappa)$ d'arité $ar(\sqcup(\downarrow_L\kappa)) = ar(\kappa) \cap \bigcup_{\kappa' \in \downarrow_L\kappa} ar(\kappa') \subseteq L$. De même, si $\uparrow_L\kappa \neq \emptyset$ alors il possède un minimum $\sqcap(\uparrow_L\kappa)$ d'arité $ar(\sqcap(\uparrow_L\kappa)) = ar(\kappa) \cap \bigcup_{\kappa' \in \uparrow_L\kappa} ar(\kappa') \subseteq L$.*

Démonstration : On montre le lemme pour $\downarrow_L\kappa$, le cas $\uparrow_L\kappa$ étant symétrique. Comme $\downarrow_L\kappa$ est un sous-ensemble des minorants de κ, il est majoré par κ et admet donc une borne supérieure[1] $\underline{\kappa}_L = \sqcup(\downarrow_L\kappa)$ et $\underline{\kappa}_L \leq_{\mathcal{K}} \kappa$. Montrons que $ar(\underline{\kappa}_L) \subseteq ar(\kappa) \cap \bigcup_{\kappa' \in \downarrow_L\kappa} ar(\kappa')$. Par la condition 3) de la définition 4.3, il existe un constructeur κ_1 tel que $\underline{\kappa}_L \leq_{\mathcal{K}} \kappa_1 \leq_{\mathcal{K}} \kappa$ et $ar(\kappa_1) = ar(\underline{\kappa}_L) \cap ar(\kappa)$. Par la condition 2) $ar(\underline{\kappa}_L) \subseteq \bigcup_{\kappa' \in \downarrow_L\kappa} ar(\kappa')$. Et on en déduit $ar(\kappa_1) \cap ar(\kappa) = ar(\underline{\kappa}_L) \cap ar(\kappa) \subseteq \bigcup_{\kappa' \in \downarrow_L\kappa} (ar(\kappa') \cap ar(\kappa)) \subseteq L$. Donc $\kappa_1 \in \downarrow_L\kappa$, d'où $\kappa_1 = \underline{\kappa}_L$. Donc $ar(\underline{\kappa}_L) \subseteq ar(\kappa)$ et $ar(\underline{\kappa}_L) \subseteq \bigcup_{\kappa' \in \downarrow_L\kappa} ar(\kappa')$.

Montrons à présent l'inclusion inverse. Soit $l \in ar(\kappa) \cap \bigcup_{\kappa' \in \downarrow_L\kappa} ar(\kappa')$. Il existe $\kappa' \in \downarrow_L\kappa$ tel que $l \in ar(\kappa')$. De plus $\kappa' \leq_{\mathcal{K}} \underline{\kappa}_L \leq_{\mathcal{K}} \kappa$. Donc par la condition sur l'arité dans la définition 3.1 d'une signature, $l \in ar(\kappa') \cap ar(\kappa) \subseteq ar(\underline{\kappa}_L)$. □

Exemple 4.6 : Toujours avec la signature de l'exemple 4.2, $\sqcap\downarrow_{\{l_2,l_3\}}\kappa_2 = \kappa_3$. ◇

4.3 Construction de bornes pour les types infinis

Dans cette section nous introduisons les définitions qui nous permettent de construire les bornes supérieures et inférieures dans l'ensemble des types infinis. Nous verrons dans la section 4.5 que cette construction est également valable pour les ensembles finis de types réguliers et pour les ensembles finis de types finis.

Plusieurs constructions sont possibles pour ces bornes. En se limitant aux types réguliers[2], il aurait été possible de les construire à l'aide d'automates

[1] On utilise ici l'hypothèse de complétude du quasi-treillis, l'ensemble $\downarrow_L\kappa$ pouvant être infini.

[2] Cette limitation aux types réguliers n'est pas très contraignante. En effet, lorsque l'on se place dans les hypothèses utilisées pour les algorithmes décrits dans les chapitres 5 et 6, la satisfiabilité des contraintes de sous-typage dans les types infinis est équivalente à la satisfiabilité des contraintes de sous-typage dans les types réguliers (corollaire 5.19).

d'arbres, comme dans la thèse de Pottier [56]. Nous avons cependant choisi une construction basée sur des approximations finies, qui nous a semblé plus intuitive.

Cette construction se déroule en trois étapes. On détermine tout d'abord ce qui sera le constructeur de tête de la borne. On calcule ensuite un ensemble d'approximations de la borne, qui lui seront identiques jusqu'à une certaine profondeur, profondeur qui croît avec les approximations. Enfin on déduit de ces approximations la borne inférieure elle-même.

Nous introduisons tout d'abord la notation suivante : si S est un ensemble de types infinis, alors on note $S/w = \{s/w \mid s \in S \wedge w \in dom(s)\}$. Nous définissons à présent l'ensemble des étiquettes utilisables sous un ensemble de types S comme l'ensemble des étiquettes l telles que S/l est minoré (ou majoré si $l \in \mathcal{L}^-$). Cet ensemble déterminera le constructeur de tête de la borne.

Définition 4.6 *Soit $S \subseteq \mathcal{T}$ un ensemble non vide de types infinis. L'ensemble des* étiquettes utilisables sous S *est l'ensemble*

$$\mathrm{UL}{\downarrow}S = \{l \in \mathcal{L}^+ \mid S/l \neq \emptyset \wedge {\downarrow}(S/l) \neq \emptyset\} \cup \{l \in \mathcal{L}^- \mid S/l \neq \emptyset \wedge {\uparrow}(S/l) \neq \emptyset\}$$

L'ensemble des étiquettes utilisables sur S *est l'ensemble*

$$\mathrm{UL}{\uparrow}S = \{l \in \mathcal{L}^+ \mid S/l \neq \emptyset \wedge {\uparrow}(S/l) \neq \emptyset\} \cup \{l \in \mathcal{L}^- \mid S/l \neq \emptyset \wedge {\downarrow}(S/l) \neq \emptyset\}$$

Exemple 4.7: Soit les types $\tau = \kappa_2(\kappa_0, \kappa_1, \kappa_4(\kappa_0))$ et $\tau' = \kappa_3(\kappa_1, \kappa_5(\kappa_1))$, construits sur la signature donnée dans l'exemple 4.2. Soit $S = \{\tau, \tau'\}$. On a $S/l_1 = \{\kappa_0\}$, $S/l_2 = \{\kappa_1\}$ et $S/l_3 = \{\kappa_4(\kappa_0), \kappa_5(\kappa_1)\}$. S/l_1 et S/l_2 sont tous deux trivialement minorés et majorés par leur unique élément. S/l_3 est majoré par $\kappa_3(\kappa_0, \kappa_1)$, mais n'est pas minoré. Par conséquent $\mathrm{UL}{\uparrow}S = \{l_1, l_2, l_3\}$ alors que $\mathrm{UL}{\downarrow}S = \{l_1, l_2\}$. Cela signifie que les sous-termes de τ et τ' correspondant à l'étiquette l_3 seront "oubliés" dans la borne inférieure de S. ◇

Nous définissons ensuite ce qui sera le constructeur de tête des bornes dans les types infinis :

Définition 4.7 *Soit $S \subseteq \mathcal{T}$ un ensemble non vide de types infinis. Le* constructeur de borne inférieure de S, *s'il existe, est le constructeur $\sqcap_\epsilon S = \sqcup{\downarrow}_{(\mathrm{UL}{\downarrow}S)}(\sqcap\{\zeta(\epsilon) \mid \zeta \in S\})$. De la même manière, le* constructeur de borne supérieure de S *est le constructeur $\sqcup_\epsilon S = \sqcap{\uparrow}_{(\mathrm{UL}{\uparrow}S)}(\sqcup\{\zeta(\epsilon) \mid \zeta \in S\})$.*

Exemple 4.8: En reprenant l'ensemble de types S défini dans l'exemple 4.7, on a $\sqcup_\epsilon S = \sqcap{\uparrow}_{(\mathrm{UL}{\uparrow}S)}(\sqcup\{\zeta(\epsilon) \mid \zeta \in S\}) = \sqcap{\uparrow}_{\{l_1,l_2,l_3\}}(\kappa_2 \sqcup \kappa_3) = \sqcap{\uparrow}_{\{l_1,l_2,l_3\}}\kappa_2 = \kappa_2$. $\sqcap_\epsilon S$ est donné par $\sqcup{\downarrow}_{(\mathrm{UL}{\downarrow}S)}(\sqcap\{\zeta(\epsilon) \mid \zeta \in S\}) = \sqcup{\downarrow}_{\{l_1,l_2\}}(\kappa_2 \sqcap \kappa_3) = \sqcup{\downarrow}_{\{l_1,l_2\}}\kappa_3 = \kappa_4$. On peut remarquer que l'arité du constructeur de borne inférieure de S, $\sqcap_\epsilon S = \kappa_4$, ne contient pas l'étiquette l_3, ce qui correspond bien à l'"oubli" des arguments de τ et τ' correspondant à l_3 dans la borne inférieure de S. ◇

Nous définissons maintenant des séquences de types qui sont des approximations des bornes jusqu'à une profondeur donnée. Pour des raisons techniques, nous supposons l'existence d'un constructeur κ_0 d'arité $ar(\kappa_0) = \emptyset$ [3].

Définition 4.8 *La* borne inférieure *(resp.* supérieure*)* de rang n *d'un ensemble non vide de types infinis* $S \subseteq \mathcal{T}$*, notée* $\sqcap_n S$ *(resp.* $\sqcup_n S$*), est définie par :*

- $\sqcap_0 S = \sqcup_0 S = \kappa_0$
- $(\sqcap_{n+1} S)(\epsilon) = \sqcap_\epsilon S$ *et pour toute étiquette* $l \in ar(\sqcap_\epsilon S)$ *:*
  - *soit* $l \in \mathcal{L}^+$ *et* $(\sqcap_{n+1} S)/l = \sqcap_n (S/l)$
  - *soit* $l \in \mathcal{L}^-$ *et* $(\sqcap_{n+1} S)/l = \sqcup_n (S/l)$
- $(\sqcup_{n+1} S)(\epsilon) = \sqcup_\epsilon S$ *et pour toute étiquette* $l \in ar(\sqcup_\epsilon S)$ *:*
  - *soit* $l \in \mathcal{L}^+$ *et* $(\sqcup_{n+1} S)/l = \sqcup_n (S/l)$
  - *soit* $l \in \mathcal{L}^-$ *et* $(\sqcup_{n+1} S)/l = \sqcap_n (S/l)$

Exemple 4.9 : Toujours avec l'ensemble de types S défini dans l'exemple 4.7, on a les bornes de rang n suivantes :

- $\sqcup_0 S = \kappa_0$, $\sqcup_1 S = \kappa_2(\kappa_0, \kappa_0, \kappa_0)$, $\sqcup_2 S = \kappa_2(\kappa_0, \kappa_1, \kappa_3(\kappa_0, \kappa_0))$ et pour tout $n \geq 3$, $\sqcup_n S = \kappa_2(\kappa_0, \kappa_1, \kappa_3(\kappa_0, \kappa_1))$.
- $\sqcap_0 S = \kappa_0$, $\sqcap_1 S = \kappa_4(\kappa_0)$ et pour tout $n \geq 2$, $\sqcap_n S = \kappa_4(\kappa_1)$. ◇

Ces approximations nous permettent de construire des candidats $\sqcup_\mathcal{T}$ et $\sqcap_\mathcal{T}$ pour les bornes supérieures et inférieures. L'idée est que, pour un ensemble de types S, tous les constructeurs se situant à profondeur n dans $\sqcap_\mathcal{T} S$ (resp. $\sqcup_\mathcal{T} S$) sont déterminés par l'approximation $\sqcap_{n+1} S$ (resp. $\sqcup_{n+1} S$).

Définition 4.9 *La fonction partielle* $\sqcap_\mathcal{T} : 2^\mathcal{T} \to (\mathcal{L}^* \to \mathcal{K})$ *(resp.* $\sqcup_\mathcal{T}$*) est définie par :*

$$(\sqcap_\mathcal{T} S)(w) = (\sqcap_{n+1} S)(w) \qquad \textit{(resp. } (\sqcup_\mathcal{T} S)(w) = (\sqcup_{n+1} S)(w) \textit{)}$$

pour tout ensemble non vide de types infinis $S \subseteq \mathcal{T}$*, pour tout* $n \in \mathbb{N}$*, pour tout* $w \in dom(\sqcap_{n+1} S)$ *(resp.* $w \in dom(\sqcup_{n+1} S)$*) tel que* $|w| = n$*.*

Le théorème suivant, au cœur de ce chapitre, justifie cette construction, indiquant que $\sqcap_\mathcal{T} S$ (resp. $\sqcup_\mathcal{T} S$) est un type et est la borne inférieure (resp. supérieure) de S dans $\mathcal{T}$:

Théorème 4.10 *[14] Soit* $\mathcal{S}$ *une signature bien formée. Alors* $\mathcal{T}(\mathcal{S})$ *est un quasi-treillis complet, dont les bornes inférieures sont désignées par* $\sqcap_\mathcal{T}$ *et les bornes supérieures par* $\sqcup_\mathcal{T}$*.*

Exemple 4.10 : La construction des bornes de l'ensemble S de l'exemple 4.7 donne deux types : $\sqcup_\mathcal{T} S = \kappa_2(\kappa_0, \kappa_1, \kappa_3(\kappa_0, \kappa_1))$ et $\sqcap_\mathcal{T} S = \kappa_4(\kappa_1)$. ◇

La preuve de ce théorème nécessite de nombreux lemmes intermédiaires et fait l'objet de la section suivante.

[3] Il est toutefois possible de lever cette restriction en remplaçant κ_0 par un type fixé quelconque dans les définitions et les preuves qui suivent.

4.4 Preuve du théorème

La preuve du théorème 4.10 se découpe en trois parties. La première partie consiste à montrer que, étant donné un ensemble de types S minoré (resp. majoré), la fonction $\sqcap_{\mathcal{T}} S$ (resp. $\sqcup_{\mathcal{T}} S$) est définie et est un type. La seconde partie consiste à montrer que ce type est un minorant (resp. un majorant) de S. Enfin la dernière partie consiste à montrer que c'est le plus grand minorant (resp. le plus petit majorant) de S. Chacune de ses parties comporte trois étapes : montrer la propriété pour $\sqcap_\epsilon$ (resp. $\sqcup_\epsilon$), puis pour $\sqcap_n$ (resp. $\sqcup_n$) et enfin pour $\sqcap_{\mathcal{T}}$ (resp. $\sqcup_{\mathcal{T}}$).

On montre donc tout d'abord que si un ensemble de type non vide S est minoré, alors $\sqcap_{\mathcal{T}} S$ est bien défini et est un type. La première étape consiste à montrer que $\sqcap_\epsilon S$ est bien défini, ce qui est une conséquence du lemme suivant :

Lemme 4.11 *Soient $S \subseteq \mathcal{T}$ un ensemble non vide de types infinis et $\tau \in \mathcal{T}$ un type. Si $\tau \in \downarrow S$, alors $\tau(\epsilon) \in \downarrow_{\mathrm{UL}\downarrow S} \sqcap \{\zeta(\epsilon) \mid \zeta \in S\}$. De même, si $\tau \in \uparrow S$, alors $\tau(\epsilon) \in \uparrow_{\mathrm{UL}\uparrow S} \sqcup \{\zeta(\epsilon) \mid \zeta \in S\}$.*

Démonstration : On note $K_S = \{\zeta(\epsilon) \mid \zeta \in S\}$ et $\kappa = \tau(\epsilon)$. Comme pour tout constructeur $\kappa_\zeta \in K_S$, $\kappa \leq_{\mathcal{K}} \kappa_\zeta$, on a $\kappa \leq_{\mathcal{K}} \sqcap K_S$. Reste à montrer que $ar(\kappa) \cap ar(\sqcap K_S) \subseteq \mathrm{UL}\downarrow S$. Soit $l \in ar(\kappa) \cap ar(\sqcap K_S)$.

1. Comme $l \in ar(\sqcap K_S)$, il existe un constructeur $\kappa_\zeta \in K_S$ tel que $l \in ar(\kappa_\zeta)$, donc $S/l \neq \emptyset$.
2. Soit $S^l = \{\zeta \in S \mid l \in ar(\zeta)\}$. Pour tout $\zeta \in S^l, \tau \leq \zeta$, et, par la propriété 3.6 p.39 :
   - Si $l \in \mathcal{L}^+$ alors pour tout $\zeta \in S^l, \tau/l \leq \zeta/l$, et donc $\forall \zeta' \in S/l, \tau/l \leq \zeta'$.
   - Si $l \in \mathcal{L}^-$ alors pour tout $\zeta \in S^l, \zeta/l \leq \tau/l$, et donc $\forall \zeta' \in S/l, \zeta' \leq \tau/l$.

En combinant 1) et 2), on obtient $l \in \mathrm{UL}\downarrow S$. On a donc $\kappa \leq_{\mathcal{K}} \sqcap K_S$ et $ar(\kappa) \cap ar(\sqcap K_S) \subseteq \mathrm{UL}\downarrow S$, d'où $\kappa \in \downarrow_{\mathrm{UL}\downarrow S} \sqcap K_S$.

La démonstration est similaire pour $\kappa' \in \downarrow_{\mathrm{UL}\uparrow S} \sqcup (\{\zeta(\epsilon) \mid \zeta \in S\})$. □

Corollaire 4.12 *Soit $S \subseteq \mathcal{T}$ un ensemble non vide de types infinis. Si S est minoré alors $\sqcap_\epsilon S$ est bien défini. Si S est majoré alors $\sqcup_\epsilon S$ est bien défini.*

Démonstration : Comme $\downarrow S \neq \emptyset$, il existe un type $\tau \in \downarrow S$ et, par le lemme 4.11, $\tau(\epsilon) \in \downarrow_{\mathrm{UL}\downarrow S} \sqcap \{\zeta(\epsilon) \mid \zeta \in S\}$. Donc $\downarrow_{\mathrm{UL}\downarrow S} \sqcap \{\zeta(\epsilon) \mid \zeta \in S\} \neq \emptyset$ et il est majoré, donc il admet une borne supérieure $\sqcup\downarrow_{\mathrm{UL}\downarrow S} \sqcap \{\zeta(\epsilon) \mid \zeta \in S\} = \sqcap_\epsilon S$. La démonstration est similaire pour $\sqcup_\epsilon S$. □

Le corollaire suivant étend ce résultat aux bornes de rang n.

Corollaire 4.13 *Soit $S \subseteq \mathcal{T}$ un ensemble non vide de types infinis. Si S est minoré alors pour tout $n \in \mathbb{N}$, $\sqcap_n S$ est bien défini. Si S est majoré alors pour tout $n \in \mathbb{N}$, $\sqcup_n S$ est bien défini.*

Démonstration : On le montre par récurrence sur n. Le cas $n = 0$ est trivial. Pour le cas $n+1$ considérons $\sqcap_{n+1}S$, le démonstration pour $\sqcup_{n+1}S$ étant similaire. Par le corollaire 4.12, $\sqcap_\epsilon S$ est bien défini. Soit l une étiquette dans $ar(\sqcap_\epsilon S)$. Par définition, on a $l \in \mathrm{UL}{\downarrow}S$, d'où $S/l \neq \emptyset$ et :

- Si $l \in \mathcal{L}^+$ alors $\downarrow(S/l) \neq \emptyset$ et donc, par récurrence, $\sqcap_n(S/l)$ est bien défini.
- Si $l \in \mathcal{L}^-$ alors $\uparrow(S/l) \neq \emptyset$ et donc, par récurrence, $\sqcup_n(S/l)$ est bien défini.

Donc $\sqcap_{n+1}S$ est bien défini. Similairement, $\sqcup_{n+1}S$ est bien défini. □

Le lemme suivant va permettre de prouver que $\sqcap_\mathcal{T} S$ est correctement formé. Il établit que, pour $m \leq n$, $\sqcap_m S$ est une approximation de $\sqcap_n S$ jusqu'à la profondeur m.

Lemme 4.14 *Soit $S \subseteq \mathcal{T}$ un ensemble non vide de types infinis. Si S est minoré (resp. majoré), alors pour tous $m, n \in \mathbb{N}$ tels que $m \leq n$, pour toute position $w \in \mathcal{L}^*$ telle que $|w| < m$, $w \in dom(\sqcap_m S)$ si et seulement si $w \in dom(\sqcap_n S)$ (resp. $w \in dom(\sqcup_m S)$ si et seulement si $w \in dom(\sqcup_n S)$), de plus $(\sqcap_m S)(w) = (\sqcap_n S)(w)$ (resp. $(\sqcup_m S)(w) = (\sqcup_n S)(w)$).*

Démonstration : On le montre par récurrence sur m. Le cas $m = 0$ est trivial car il n'existe pas de position w avec $|w| < 0$. Considérons le cas $m + 1 \leq n + 1$ avec $\sqcap_{m+1}S$ et $\sqcap_{n+1}S$.

- Si $w = \epsilon$, par définition, $\epsilon \in dom(\sqcap_{m+1}S)$ et $\epsilon \in dom(\sqcap_{n+1}S)$. De plus, $(\sqcap_{m+1}S)(\epsilon) = \sqcap_\epsilon S = (\sqcap_{n+1}S)(\epsilon)$.
- Si $w = l.w'$, $l \in \Delta ar(\sqcap_\epsilon S)$. Si $l \in \mathcal{L}^+$, $(\sqcap_{m+1}S)/l = \sqcap_m(S/l)$ et $(\sqcap_{n+1}S)/l = \sqcap_n(S/l)$. Par récurrence $w' \in dom(\sqcap_m(S/l))$ si et seulement si $w' \in dom(\sqcap_n(S/l))$, d'où $w \in dom(\sqcap_{m+1}S)$ si et seulement si $w \in dom(\sqcap_{n+1}S)$. Par récurrence on obtient également $(\sqcap_m(S/l))(w') = (\sqcap_n(S/l))(w')$, d'où $(\sqcap_{m+1}S)(w) = (\sqcap_{n+1}S)(w)$. Le cas $l \in \mathcal{L}^-$ est symétrique.

La démonstration pour $\sqcup_{m+1}S$ et $\sqcup_{n+1}S$ est similaire. □

On peut maintenant énoncer la propriété suivante de bonne définition de $\sqcap_\mathcal{T}$ (resp. $\sqcup_\mathcal{T}$) :

Propriété 4.15 *Soit $S \subseteq \mathcal{T}$ un ensemble non vide de types infinis. Si S est minoré alors $\sqcap_\mathcal{T} S$ est défini et est un type. Si S est majoré, alors $\sqcup_\mathcal{T} S$ est défini et est un type.*

Démonstration : Supposons que S soit minoré. Par le corollaire 4.13, pour tout $n \in \mathbb{N}$, $\sqcap_n S$ est défini, donc $\sqcap_\mathcal{T} S$ est défini. Montrons à présent que c'est un type :

- Soit une position $w_1.w_2 \in dom(\sqcap_\mathcal{T} S)$. Par définition, $w_1.w_2 \in dom(\sqcap_{|w_1.w_2|+1}S)$, et donc $w_1 \in dom(\sqcap_{|w_1.w_2|+1}S)$. Par le lemme 4.14, $w_1 \in dom(\sqcap_{|w1|+1}S)$, donc par définition, $w_1 \in dom(\sqcap_\mathcal{T} S)$. $dom(\sqcap_\mathcal{T} S)$ est donc clos par préfixe.

- $\epsilon \in dom(\sqcap_1 S)$, donc $\epsilon \in dom(\sqcap_{\mathcal{T}} S)$.
- Soit une position $w \in dom(\sqcap_{\mathcal{T}} S)$ et $\kappa = (\sqcap_{\mathcal{T}} S)(w) = (\sqcap_{|w|+1} S)(w)$. Par définition, $w.l \in dom(\sqcap_{\mathcal{T}} S) \Leftrightarrow w.l \in dom(\sqcap_{|w.l|+1} S)$. Par le lemme 4.14, $(\sqcap_{|w.l|+1} S)(w) = (\sqcap_{|w|+1} S)(w) = \kappa$. Donc $w.l \in dom(\sqcap_{|w.l|+1} S) \Leftrightarrow l \in ar(\kappa)$. Donc $w.l \in dom(\sqcap_{\mathcal{T}} S) \Leftrightarrow l \in ar(\kappa)$

La démonstration est similaire pour $\sqcup_{\mathcal{T}} S$. □

La propriété suivante caractérise $\sqcap_{\mathcal{T}} S$, ou, plus précisément ses sous-termes. Elle est non seulement utilisée dans la preuve lemme 4.17 sur les approximations $\sqcap_n$ et $\sqcup_n$, mais également dans celle du théorème 4.25 sur les types réguliers et dans celle du théorème 4.27 sur les types finis.

Propriété 4.16 *Soit $S \subseteq \mathcal{T}$ un ensemble non vide de types infinis. Si S est minoré, alors pour toute étiquette $l \in ar((\sqcap_{\mathcal{T}} S)(\epsilon))$:*

- *Soit $l \in \mathcal{L}^+$ et $(\sqcap_{\mathcal{T}} S)/l = \sqcap_{\mathcal{T}}(S/l)$.*
- *Soit $l \in \mathcal{L}^-$ et $(\sqcap_{\mathcal{T}} S)/l = \sqcup_{\mathcal{T}}(S/l)$.*

De la même manière, si S est majoré, alors pour toute étiquette $l \in ar((\sqcup_{\mathcal{T}} S)(\epsilon))$:

- *Soit $l \in \mathcal{L}^+$ et $(\sqcup_{\mathcal{T}} S)/l = \sqcup_{\mathcal{T}}(S/l)$*
- *Soit $l \in \mathcal{L}^-$ et $(\sqcup_{\mathcal{T}} S)/l = \sqcap_{\mathcal{T}}(S/l)$*

Démonstration : $((\sqcap_{\mathcal{T}} S)/l)(w) = (\sqcap_{\mathcal{T}} S)(l.w) = (\sqcap_{|l.w|+1} S)(l.w)$. Si $l \in \mathcal{L}^+$ alors $(\sqcap_{|l.w|+1} S)/l = \sqcap_{|w|+1}(S/l)$. Dans ce cas, $(\sqcap_{|l.w|+1} S)(l.w) = ((\sqcap_{|l.w|+1} S)/l)(w)$ $= (\sqcap_{|w|+1}(S/l))(w) = (\sqcap_{\mathcal{T}}(S/l))(w)$. Le cas $l \in \mathcal{L}^-$ est symétrique. On démontre de la même manière la proposition pour $\sqcup_{\mathcal{T}} S$. □

Le lemme suivant exprime que les $\sqcap_n S$ sont des approximations de $\sqcap_{\mathcal{T}} S$.

Lemme 4.17 *Soit $S \subseteq \mathcal{T}$ un ensemble non vide de types infinis. Si S est minoré, alors pour tout $n \in \mathbb{N}$, $\sqcap_n S \leq_n \sqcap_{\mathcal{T}} S$ et $\sqcap_{\mathcal{T}} S \leq_n \sqcap_n S$. Si S est majoré, alors pour tout $n \in \mathbb{N}$, $\sqcup_n S \leq_n \sqcup_{\mathcal{T}} S$ et $\sqcup_{\mathcal{T}} S \leq_n \sqcup_n S$.*

Démonstration : On le montre par récurrence sur n. Le cas $n = 0$ est trivial. Considérons le cas $\sqcap_{\mathcal{T}} S \leq_{n+1} \sqcap_{n+1} S$. Par lemme 4.14, $(\sqcap_{\mathcal{T}} S)(\epsilon) = (\sqcap_1 S)(\epsilon) = (\sqcap_{n+1} S)(\epsilon)$. On a donc $(\sqcap_{\mathcal{T}} S)(\epsilon) \leq_{\mathcal{K}} (\sqcap_{n+1} S)(\epsilon)$ et $(\sqcap_{n+1} S)(\epsilon) \leq_{\mathcal{K}} (\sqcap_{\mathcal{T}} S)(\epsilon)$. Soit une étiquette $l \in ar((\sqcap_{\mathcal{T}} S)(\epsilon))$. Si $l \in \mathcal{L}^+$, alors, par la propriété 4.16, $(\sqcap_{\mathcal{T}} S)/l = \sqcap_{\mathcal{T}}(S/l)$. On a également $(\sqcap_{n+1} S)/l = \sqcap_n(S/l)$. Par récurrence, $\sqcap_{\mathcal{T}}(S/l) \leq_n \sqcap_n(S/l)$ et $\sqcap_n(S/l) \leq_n \sqcap_{\mathcal{T}}(S/l)$. D'où $(\sqcap_{n+1} S)/l \leq_n (\sqcap_{\mathcal{T}} S)/l$ et $(\sqcap_{\mathcal{T}} S)/l \leq_n (\sqcap_{n+1} S)/l$. Le cas $l \in \mathcal{L}^-$ est symétrique. On a donc bien $\sqcap_{\mathcal{T}} S \leq_{n+1} \sqcap_{n+1} S$ et $\sqcap_{n+1} S \leq_{n+1} \sqcap_{\mathcal{T}} S$.

De la même manière, $\sqcup_{\mathcal{T}} S \leq_{n+1} \sqcup_{n+1} S$ et $\sqcup_{n+1} S \leq_{n+1} \sqcup_{\mathcal{T}} S$. □

On montre à présent que $\sqcap_{\mathcal{T}} S$ est un minorant de S. On commence par montrer que son constructeur de tête est un minorant des constructeurs de tête des éléments de S.

Lemme 4.18 *Soit $S \subseteq \mathcal{T}$ un ensemble non vide de types infinis. Si S est minoré, alors pour tout $\zeta \in S$, $\sqcap_\epsilon S \leq_\mathcal{K} \zeta(\epsilon)$. Si S est majoré, alors pour tout constructeur de tête $\zeta(\epsilon)$ tel que $\zeta \in S$, $\zeta(\epsilon) \leq_\mathcal{K} \sqcup_\epsilon S$.*

Démonstration : Supposons qu'il existe un type τ minorant S, alors $\sqcap_\epsilon S$ est défini. Notons $K_S = \{\zeta(\epsilon) \mid \zeta \in S\}$. Par définition, $\sqcap_\epsilon S = \sqcup(\downarrow_{\mathrm{UL}\downarrow S} \sqcap K_S)$. De plus pour tout type $\zeta \in S$, $\sqcap K_S \leq_\mathcal{K} \zeta(\epsilon)$. Comme, par définition, $\sqcap_\epsilon S \leq_\mathcal{K} \sqcap K_S$, on a pour tout type $\zeta \in S$, $\sqcap_\epsilon S \leq_\mathcal{K} \zeta(\epsilon)$. On montre de la même manière le lemme pour $\sqcup_\epsilon S$. □

Corollaire 4.19 *Soit $S \subseteq \mathcal{T}$ un ensemble non vide de types infinis. Si S est minoré, alors pour tout $n \in \mathbb{N}$, pour tout type $\zeta \in S$, $\sqcap_n S \leq_n \zeta$. De même, si S est majoré alors pour tout $n \in \mathbb{N}$, pour tout type $\zeta \in S$, $\zeta \leq_n \sqcup_n S$.*

Démonstration : On le montre par récurrence sur n. Le cas $n = 0$ est trivial. Considérons le cas $n + 1$. Soit $\zeta \in S$. On a $(\sqcap_{n+1} S)(\epsilon) = \sqcap_\epsilon S$, et par lemme 4.18, $\sqcap_\epsilon S \leq_\mathcal{K} \zeta(\epsilon)$. Soit une étiquette $l \in ar(\sqcap_\epsilon S) \cap ar(\zeta(\epsilon))$. Par le lemme 4.5, $l \in ar(\sqcup\downarrow_{(\mathrm{UL}\downarrow S)}(\sqcap\{\zeta'(\epsilon) \mid \zeta' \in S\})) \subseteq \mathrm{UL}\downarrow(S)$. Si $l \in \mathcal{L}^+$, alors $(\sqcap_{n+1} S)/l = \sqcap_n(S/l)$. Comme $\zeta/l \in S/l$, par récurrence on obtient $(\sqcap_{n+1} S)/l = \sqcap_n(S/l) \leq_n \zeta/l$. Le cas $l \in \mathcal{L}^-$ est symétrique.

De la même manière, $\zeta \leq_{n+1} \sqcup_{n+1} S$. □

Propriété 4.20 *Soit $S \subseteq \mathcal{T}$ un ensemble non vide de types infinis. Si S est minoré, alors pour tout type $\zeta \in S$, $\sqcap_\mathcal{T} S \leq \zeta$. De même, si S est majoré, alors pour tout type $\zeta \in S$, $\zeta \leq \sqcup_\mathcal{T} S$.*

Démonstration : Soit un type $\zeta \in S$. Pour tout entier $n \in \mathbb{N}$, par le lemme 4.17 $\sqcap_\mathcal{T} S \leq_n \sqcap_n S$ et par le corollaire 4.19 $\sqcap_n S \leq_n \zeta$. Donc pour tout $n \in \mathbb{N}$, $\sqcap_\mathcal{T} S \leq_n \zeta$. Donc $\sqcap_\mathcal{T} S \leq \zeta$. La démonstration est similaire pour $\sqcup_\mathcal{T}$. □

On montre enfin que $\sqcap_\mathcal{T} S$ est plus grand que tous les minorants de S.

Lemme 4.21 *Soit $S \subseteq \mathcal{T}$ un ensemble non vide de types infinis. Pour tout type $\tau \in \downarrow S$, $\tau(\epsilon) \leq_\mathcal{K} \sqcap_\epsilon S$ et pour tout type $\tau \in \uparrow S$, $\sqcup_\epsilon S \leq_\mathcal{K} \tau(\epsilon)$.*

Démonstration : Soit $\tau \in \downarrow S$.
Par le lemme 4.11, $\kappa = \tau(\epsilon) \in \downarrow_{\mathrm{UL}\downarrow S} \sqcap \{\zeta(\epsilon) \mid \zeta \in S\}$. Donc par définition de $\sqcap_\epsilon S$, $\kappa \leq_\mathcal{K} \sqcap_\epsilon S$. On montre de la même manière le lemme pour $\sqcup_\epsilon S$. □

Corollaire 4.22 *Soit $S \subseteq \mathcal{T}$ un ensemble non vide de types infinis. Pour tout type $\tau \in \downarrow S$, pour tout $n \in \mathbb{N}$, $\tau \leq_n \sqcap_n S$. Pour tout type $\tau \in \uparrow S$, pour tout $n \in \mathbb{N}$, $\sqcup_n S \leq_n \tau$.*

Démonstration : On le montre par récurrence sur n. Le cas $n = 0$ est trivial. Considérons le cas $n + 1$. Soit $\tau \in \downarrow S$. Par le lemme 4.21, $\tau(\epsilon) \leq_{\mathcal{K}} \sqcap_\epsilon S$. Soit $l \in ar(\tau(\epsilon)) \cap ar(\sqcap_\epsilon S)$. Si $l \in \mathcal{L}^+$, alors, par la propriété 3.6, $\tau/l \in \downarrow(S/l)$. Comme $ar(\sqcap_\epsilon S) \subseteq \mathrm{UL}{\downarrow}S$, $S/l \neq \emptyset$. Par récurrence, on obtient $\tau/l \leq_n \sqcap_n(S/l) = (\sqcap_{n+1} S)/l$. Le cas $l \in \mathcal{L}^-$ est symétrique. Donc $\tau \leq_{n+1} \sqcap_{n+1} S$. La preuve est similaire pour $\sqcup_{n+1} S$. □

Propriété 4.23 *Soit $S \subseteq \mathcal{T}$ un ensemble non vide de types infinis. Pour tout type $\tau \in \downarrow S$, $\tau \leq \sqcap_{\mathcal{T}} S$. Pour tout type $\tau \in \uparrow S$, $\sqcup_{\mathcal{T}} S \leq \tau$.*

Démonstration : Soit $\tau \in \downarrow S$. Pour tout $n \in \mathbb{N}$, par le corollaire 4.22, $\tau \leq_n \sqcap_n S$ et, par le lemme 4.17, $\sqcap_n S \leq_n \sqcap_{\mathcal{T}} S$. Donc pour tout $n \in \mathbb{N}$, $\tau \leq_n \sqcap_{\mathcal{T}} S$. Donc $\tau \leq \sqcap_{\mathcal{T}} S$. La preuve est similaire pour $\sqcup_{\mathcal{T}} S$. □

On peut à présent démontrer le théorème :

Démonstration (du théorème 4.10) : Soit S un ensemble non vide de types infinis. Supposons que S soit minoré. Par la propriété 4.15, $\sqcap_{\mathcal{T}} S$ est défini et est un type. Par la propriété 4.20, c'est un minorant de S et par la propriété 4.23, il est plus grand que tous les minorants de S. $\sqcap_{\mathcal{T}} S$ est donc la borne inférieure de S. De même si S est majoré, $\sqcup_{\mathcal{T}} S$ est la borne supérieure de S. □

4.5 Types réguliers et types finis

Dans cette section, nous étudions la structure des types réguliers et celle des types finis construits sur une signature bien formée. Plus précisément, nous montrons que l'ensemble ordonné des types réguliers et celui des types finis forment des quasi-treillis.

Propriété 4.24 *Soient τ et υ deux types réguliers. Si $\{\tau, \upsilon\}$ est minoré alors $\tau \sqcap_{\mathcal{T}} \upsilon$ est un type régulier. Si $\{\tau, \upsilon\}$ est majoré alors $\tau \sqcup_{\mathcal{T}} \upsilon$ est un type régulier.*

Démonstration : Soit Sub l'ensemble des sous-termes de τ et υ. Soit $Sub_\sqcap = \{\tau_1 \sqcap_{\mathcal{T}} \tau_2 \mid \tau_1, \tau_2 \in Sub\}$ et $Sub_\sqcup = \{\tau_1 \sqcup_{\mathcal{T}} \tau_2 \mid \tau_1, \tau_2 \in Sub\}$.

On montre que pour tout type $\tau' \in Sub_\sqcap \cup Sub_\sqcup$, pour toute position $w \in dom(\tau)$, $\tau'/w \in Sub_\sqcap \cup Sub_\sqcup$, par induction sur w. Si $w = \epsilon$ alors $\tau'/\epsilon = \tau' \in S_\sqcap \cup S_\sqcup$. Sinon $w = l.w'$. Supposons $\tau' = \tau_1 \sqcap_{\mathcal{T}} \tau_2$ avec $\tau_1, \tau_2 \in Sub$ (la preuve est similaire pour $\tau = \tau_1 \sqcup_{\mathcal{T}} \tau_2$). Supposons également que $l \in \mathcal{L}^+$ (la preuve est similaire pour $l \in \mathcal{L}^-$). Par la propriété 4.16, $\tau'/l = (\tau_1 \sqcap_{\mathcal{T}} \tau_2)/l = \sqcap_{\mathcal{T}}(\{\tau_1, \tau_2\}/l)$.

- Si τ_1/l est défini mais as τ_2/l alors $\tau'/l = \tau_1/l \in Sub \subseteq Sub_\sqcap \cup Sub_\sqcup$.
- Si τ_2/l est défini mais as τ_1/l alors $\tau'/l = \tau_2/l \in Sub \subseteq Sub_\sqcap \cup Sub_\sqcup$.

– Si τ_1/l et τ_2/l sont tous deux définis, alors $\tau_1/l \in Sub$ et $\tau_2/l \in Sub$, donc $\tau'/l = (\tau_1/l)\sqcap_{\mathcal{T}}(\tau_2/l) \in Sub_{\sqcap} \cup Sub_{\sqcup}$.

Donc, $\tau'/l \in Sub_{\sqcap} \cup Sub_{\sqcup}$. Par induction, $(\tau'/l)/w' \in Sub_{\sqcap} \cup Sub_{\sqcup}$, d'où $\tau'/w \in Sub_{\sqcap} \cup Sub_{\sqcup}$.

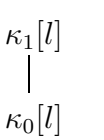
Comme τ et υ sont des types réguliers, Sub est fini et $Sub_{\sqcap} \cup Sub_{\sqcup}$ est également fini. Cependant, $\tau\sqcap_{\mathcal{T}}\upsilon \in Sub_{\sqcap}$, donc tous ses sous-termes sont dans $Sub_{\sqcap} \cup Sub_{\sqcup}$. Il n'y a donc qu'un nombre fini de ces sous-termes, et $\tau\sqcap_{\mathcal{T}}\upsilon$ est donc un type régulier. De même $\tau\sqcup_{\mathcal{T}}\upsilon$ est un type régulier. □

Théorème 4.25 *Soit $\mathcal{S}$ une signature bien formée. $(\mathcal{R}(\mathcal{S}), \leq_{\mathcal{T}(\mathcal{S})})$ est un quasi-treillis.*

Démonstration : Par le théorème 4.10, $(\mathcal{T}(\mathcal{S}), \leq_{\mathcal{T}(\mathcal{S})})$ est un quasi-treillis. Par la propriété 4.24, si $\tau_1, \tau_2 \in \mathcal{R}(\mathcal{S})$, et si $\{\tau_1, \tau_2\}$ est minoré (resp. majoré) alors $\tau_1\sqcap_{\mathcal{T}}\tau_2 \in \mathcal{R}(\mathcal{S})$ (resp. $\tau_1\sqcup_{\mathcal{T}}\tau_2 \in \mathcal{R}(\mathcal{S})$). Donc $(\mathcal{R}(\mathcal{S}), \leq_{\mathcal{T}(\mathcal{S})})$ est un quasi-treillis où $\sqcap_{\mathcal{T}}$ désigne les bornes inférieures et $\sqcup_{\mathcal{T}}$ désigne les bornes supérieures. □

On peut cependant noter que $(\mathcal{R}(\mathcal{S}), \leq_{\mathcal{T}(\mathcal{S})})$ n'est pas forcément un quasi-treillis *complet*, comme le montre l'exemple suivant :

Exemple 4.11 : Soient $\mathcal{K} = \{\kappa_0, \kappa_1\}$ avec $\kappa_0 \leq_{\mathcal{K}} \kappa_1$ et $ar(\kappa_0) = ar(\kappa_1) = \{l\}$. Soit $(\upsilon_n)_{n\geq 0}$ la suite de types définie par : $\upsilon_n(w) = \kappa_1$ si $|w| = \frac{m(m+1)}{2}$ avec $m \leq n$, et $\upsilon_n(w) = \kappa_0$ sinon. On peut vérifier que $(\upsilon_n)_{n\geq 0}$ est majorée (par exemple par $\kappa_1(\kappa_1(\kappa_1(\ldots))))$ mais n'admet pas de borne supérieure dans $\mathcal{R}(\mathcal{S})$.



◇

Les types finis ont, vis-à-vis de la structure de quasi-treillis, les mêmes propriétés que les types réguliers :

Propriété 4.26 *Soient τ et υ deux types finis. Si $\{\tau, \upsilon\}$ est minoré, alors $\tau\sqcap_{\mathcal{T}}\upsilon$ est fini. Si $\{\tau, \upsilon\}$ est majoré, alors $\tau\sqcup_{\mathcal{T}}\upsilon$ est fini.*

Démonstration : On le montre par induction sur τ. Supposons que $\{\tau, \upsilon\}$ soit minoré. Par la propriété 4.16, pour toute étiquette $l \in ar((\tau\sqcap_{\mathcal{T}}\upsilon)(\epsilon))$, si $l \in \mathcal{L}^+$, alors $(\tau\sqcap_{\mathcal{T}}\upsilon)/l = \sqcap_{\mathcal{T}}(\{\tau, \upsilon\}/l)$, sinon $(\tau\sqcap_{\mathcal{T}}\upsilon)/l = \sqcup_{\mathcal{T}}(\{\tau, \upsilon\}/l)$. On distingue trois cas :

– Soit τ/l est défini mais pas υ/l. Dans ce cas $(\tau\sqcap_{\mathcal{T}}\upsilon)/l = \tau/l$ qui est un type fini.
– Soit υ/l est défini mais pas τ/l. Dans ce cas $(\tau\sqcap_{\mathcal{T}}\upsilon)/l = \upsilon/l$ qui est un type fini.
– Soit τ/l et υ/l sont définis. Si $l \in \mathcal{L}^+$, alors $(\tau\sqcap_{\mathcal{T}}\upsilon)/l = (\tau/l)\sqcap_{\mathcal{T}}(\upsilon/l)$ qui est fini par induction. De même, si $l \in \mathcal{L}^-$, $(\tau\sqcap_{\mathcal{T}}\upsilon)/l = (\tau/l)\sqcup_{\mathcal{T}}(\upsilon/l)$ qui est fini par induction.

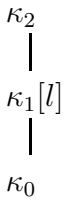
Donc pour toute étiquette $l \in ar((\tau \sqcap_{\mathcal{T}} \upsilon)(\epsilon))$, $(\tau \sqcap_{\mathcal{T}} \upsilon)/l$ est fini. Donc $\tau \sqcap_{\mathcal{T}} \upsilon$ est fini. □

Théorème 4.27 *Soit $\mathcal{S}$ une signature bien formée. $(\mathcal{F}(\mathcal{S}), \leq_{\mathcal{T}(\mathcal{S})})$ est un quasi-treillis.*

Démonstration : La preuve est similaire à celle du théorème 4.25. Par le théorème 4.10, $(\mathcal{T}(\mathcal{S}), \leq_{\mathcal{T}(\mathcal{S})})$ est un quasi-treillis. Par la propriété 4.26, si $\tau_1, \tau_2 \in \mathcal{F}(\mathcal{S})$, et si $\{\tau_1, \tau_2\}$ est minoré (resp. majoré) alors $\tau_1 \sqcap_{\mathcal{T}} \tau_2 \in \mathcal{F}(\mathcal{S})$ (resp. $\tau_1 \sqcup_{\mathcal{T}} \tau_2 \in \mathcal{F}(\mathcal{S})$). Donc $(\mathcal{F}(\mathcal{S}), \leq_{\mathcal{T}(\mathcal{S})})$ est un quasi-treillis où $\sqcap_{\mathcal{T}}$ désigne les bornes inférieures et $\sqcup_{\mathcal{T}}$ désigne les bornes supérieures. □

Tout comme $\mathcal{R}(\mathcal{S})$, $\mathcal{F}(\mathcal{S})$, n'est pas forcément un quasi-treillis *complet* :

Exemple 4.12 :
Soient $\mathcal{K} = \{\kappa_0, \kappa_1, \kappa_2\}$ avec $\kappa_0 \leq_{\mathcal{K}} \kappa_1 \leq_{\mathcal{K}} \kappa_2$, $ar(\kappa_1) = \{l\}$ et $ar(\kappa_0) = ar(\kappa_2) = \emptyset$. Soit $(\upsilon_n)_{n\geq 0}$ la suite de types définie par : $\upsilon_n(w) = \kappa_1$ si $|w| < n$, et $\upsilon_n(w) = \kappa_0$ si $|w| = n$. L'ensemble des majorants de $(\upsilon_n)_{n\geq 0}$ dans $\mathcal{F}(\mathcal{S})$ sont de la forme $\kappa_1(\kappa_1(\ldots \kappa_1(\kappa_2)\ldots))$ et peuvent être décrit par la suite $(\upsilon_n)_{n\geq 0}$ définie par $\upsilon_n(w) = \kappa_1$ si $|w| < n$, et $\upsilon_n(w) = \kappa_2$ si $|w| = n$. Cette suite est infinie et décroissante strictement. Donc $(\upsilon_n)_{n\geq 0}$ est majorée mais n'admet pas de borne supérieure dans $\mathcal{F}(\mathcal{S})$.



◇

Chapitre 5

Systèmes de contraintes de sous-typage clos

Ce chapitre, qui reprend des résultats que nous avons publiés dans [14], a pour but de définir et d'étudier les systèmes de contraintes de sous-typage *clos* ainsi que les algorithmes de clôture de tels systèmes. Cette notion de clôture est similaire à celle de Trifonov et Smith dans [70] et à la notion de clôture forte de Pottier dans [57] pour les treillis. Nous montrons qu'elle constitue une condition suffisante pour la satisfiabilité d'un système de contraintes de sous-typage dans un quasi-treillis de types. Les algorithmes de clôture permettent ainsi, moyennant des conditions sur les extremums du quasi-treillis, de tester la satisfiabilité des contraintes de sous-typage en exhibant un système clos équivalent au système de départ.

5.1 Définitions

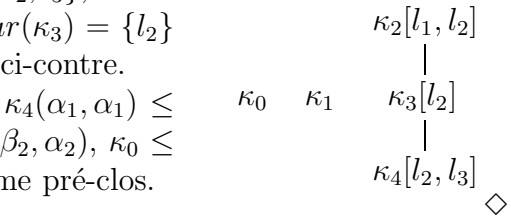
Nous introduisons tout d'abord quelques notations. Pour toute variable $\alpha \in V(C)$, on note $\Downarrow_C \alpha = \{\tau \mid \tau \leq \alpha \in C \wedge \tau \notin \mathcal{V}\}$ l'ensemble des types construits minorant α dans C et $\Uparrow_C \alpha = \{\tau \mid \alpha \leq \tau \in C \wedge \tau \notin \mathcal{V}\}$ l'ensemble des types construits majorant α dans C. Pour un ensemble de variables $A \subseteq V(C)$, on note $\Downarrow_C A = \bigcup_{\alpha \in A} \Downarrow_C \alpha$ et $\Uparrow_C A = \bigcup_{\alpha \in A} \Uparrow_C \alpha$. Lorsque C sera clair d'après le contexte, on pourra l'omettre dans les notations. On supposera que les contraintes sont interprétées dans $\mathcal{T}(\mathcal{S})$ où $\mathcal{S} = (\mathcal{K}, \leq_{\mathcal{K}}, \mathcal{L}^+, \mathcal{L}^-, ar)$ est une signature bien formée.

Nous définissons maintenant les systèmes pré-clos comme étant des systèmes de contraintes dans lesquels les variables sont bornées, plus précisément :

Définition 5.1 *Un système de contraintes de sous-typage C est dit* pré-clos supérieurement *si pour toute variable $\alpha \in V(C)$, $\Uparrow_C \alpha \neq \emptyset$. Il est dit* pré-clos inférieurement *si*

pour toute variable $\alpha \in V(C)$, $\Downarrow_C \alpha \neq \emptyset$. *Il est dit* pré-clos *si il est pré-clos supérieurement et pré-clos inférieurement.*

Exemple 5.1 : Considérons la structure $\mathcal{S} = (\mathcal{K}, <_\mathcal{K}, \mathcal{L}^+, \mathcal{L}^-, ar)$, avec $\mathcal{K} = \{\kappa_0, \kappa_1, \kappa_2, \kappa_3, \kappa_4\}$, $\kappa_4 <_\mathcal{K} \kappa_3 <_\mathcal{K} \kappa_2$, $\mathcal{L}^+ = \{l_1, l_2, l_3\}$, $\mathcal{L}^- = \emptyset$, $ar(\kappa_0) = ar(\kappa_1) = \emptyset$, $ar(\kappa_2) = \{l_1, l_2\}$, $ar(\kappa_3) = \{l_2\}$ et $ar(\kappa_4) = \{l_2, l_3\}$, résumée dans la figure ci-contre.



Le système de contraintes $C = \{\alpha_1 \leq \alpha_2,\ \kappa_4(\alpha_1, \alpha_1) \leq \alpha_1,\ \alpha_1 \leq \kappa_2(\beta_1, \alpha_1),\ \kappa_3(\alpha_2) \leq \alpha_2,\ \alpha_2 \leq \kappa_2(\beta_2, \alpha_2),\ \kappa_0 \leq \beta_1,\ \beta_1 \leq \kappa_0,\ \kappa_1 \leq \beta_2, \beta_2 \leq \kappa_1\}$ est un système pré-clos. ◇

La notion de clôture est définie par rapport à la fonction de décomposition de contraintes *dec* de Trifonov et Smith [70] et rappelée dans le tableau 5.1.

$$
\begin{array}{rcl}
dec(\alpha \leq \beta) & = & \{\alpha \leq \beta\} \\
dec(\tau \leq \alpha) & = & \{\tau \leq \alpha\} \\
dec(\alpha \leq \tau) & = & \{\alpha \leq \tau\} \\
dec(\tau_1 \leq \tau_2) & = & \displaystyle\bigcup_{l \in ar^+(\tau_1(\epsilon)) \cap ar^+(\tau_2(\epsilon))} \tau_1/l \leq \tau_2/l \cup \bigcup_{l \in ar^-(\tau_1(\epsilon)) \cap ar^-(\tau_2(\epsilon))} \tau_2/l \leq \tau_1/l \\
 & & \text{si } \tau_1(\epsilon) \leq \tau_2(\epsilon)
\end{array}
$$

TAB. 5.1 – Fonction de décomposition de contraintes

Définition 5.2 *Un système de contraintes C est dit* clos *s'il est pré-clos et si pour toute contrainte $c \in C$, $dec(c)$ est défini et inclus dans C et pour tout $\{\tau_1 \leq \alpha, \alpha \leq \tau_2\} \subseteq C$, $dec(\{\tau_1 \leq \tau_2\})$ est défini et inclus dans C.*

Exemple 5.2 : En reprenant la structure $\mathcal{S}$ et le système de contraintes C donnés dans l'exemple 5.1, le système de contrainte $C' = C \cup \{\alpha_1 \leq \kappa_2(\beta_2, \alpha_2), \kappa_4(\alpha_1, \alpha_1) \leq \alpha_2\}$ est clos. Il a été obtenu à partir de C en appliquant itérativement *dec* sur le système de contraintes courant C_n, ainsi que sur les contraintes $\tau_1 \leq \tau_2$ telles qu'il existe une variable de type α telle que $\{\tau_1 \leq \alpha, \alpha \leq \tau_2\} \subseteq C_n$, et ceci jusqu'à obtention d'un point fixe. ◇

Comme énoncé dans le chapitre 3, nous supposerons par la suite que les contraintes de sous-typage sont formées de types plats.

5.2 Satisfiabilité des systèmes clos

La notion de clôture tire son importance du fait que les systèmes clos sont satisfiables dans les quasi-treillis comme énoncé dans le théorème suivant :

Théorème 5.3 *[14] Soit $\mathcal{S}$ une signature bien formée. Tout système de contraintes clos construit sur $\mathcal{F}(\mathcal{S}_\mathcal{V})$ est satisfiable dans $\mathcal{T}(\mathcal{S})$.*

Le reste de cette section vise à démontrer ce théorème. Pour ce faire, on considère un système de contraintes clos C dont on va construire une solution ρ. A cette fin, on va construire une fonction partielle Ψ qui associe un type à deux ensembles de variables $A, B \subseteq V(C)$. Intuitivement, Ψ sera comprise entre $\rho(A)$ et $\rho(B)$. La fonction Ψ sera elle-même construite inductivement à partir de la fonction Ψ_ϵ, définie ci-après, qui en détermine le constructeur de tête.

Définition 5.4 *Étant donné un système clos C, $\Psi_\epsilon : 2^{V(C)} \times 2^{V(C)} \to \mathcal{K}$ est une fonction partielle définie par $\Psi_\epsilon(A, B) = \sqcup(\downarrow_{ar(\sqcup I)} \sqcap S)$ lorsqu'elle existe, avec $S = \{\tau(\epsilon) \mid \tau \in \Uparrow B\}$ et $I = \{\tau(\epsilon) \mid \tau \in \Downarrow A\}$.*

Intuitivement, $\Psi_\epsilon(A, B)$ est un constructeur de type compris entre les constructeurs $\sqcap S$ et $\sqcup I$, et dont l'arité est l'intersection des arités de ces deux constructeurs. L'idée ici est de ne conserver dans la construction de Ψ, et donc de ρ, que des arguments qui sont minorés et majorés, ce qui va assurer la possibilité de les construire.

Exemple 5.3 : Considérons le système C' défini dans l'exemple 5.2. On a, entre autres, $\Psi_\epsilon(\{\alpha_1\}, \{\alpha_1\}) = \sqcup(\downarrow_{ar(\sqcup\{\kappa_4\})} \sqcap \{\kappa_2\}) = \sqcup(\downarrow_{\{l_2,l_3\}} \kappa_2) = \kappa_3$. On peut constater que $ar(\kappa_3) = \{l_2\}$, c'est-à-dire que les variables β_1 et β_2, qui correspondent à l'étiquette l_1 dans les bornes de α_1 et qui sont incompatibles, n'ont pas d'argument qui leur correspond dans $\Psi_\epsilon(\{\alpha_1\}, \{\alpha_1\})$. ◇

La condition suivante sur un système de contraintes C et deux ensembles de variables A et B est une condition suffisante pour que $\Psi_\epsilon(A, B)$ soit définie (lemme 5.7). Cette condition joue un rôle particulier dans la suite de la démonstration du théorème 5.3, en définissant la portée d'un certain nombre de définitions et de lemmes intermédiaires.

Condition 5.5 *$A, B \subseteq V(C)$, $A, B \neq \emptyset$ et pour tous $\alpha \in A, \beta \in B$, $\alpha \leq \beta \in C$*

Le lemme 5.6 sera utile pour plusieurs démonstrations qui suivent. Il exprime que la transitivité des bornes est incluse dans la clôture.

Lemme 5.6 *Soient C un système clos et $A, B \subseteq V(C)$ vérifiant la condition 5.5. Pour tous types $\tau \in \Downarrow A$ et $\tau' \in \Uparrow B$, $dec(\tau \leq \tau')$ est défini et $dec(\tau \leq \tau') \subseteq C$.*

Démonstration : Comme les types qui apparaissent dans C sont plats, $(\Downarrow A)/l \subseteq V(C)$ et $(\Uparrow B)/l \subseteq V(C)$. Comme C est clos, pour toutes variables $\alpha, \beta \in V(C)$ telles que $\alpha \leq \beta \in C$, pour tous types $\tau \in \Downarrow \alpha$ et $\tau' \in \Uparrow \beta$, $dec(\tau \leq \tau')$ est défini et $dec(\tau \leq \tau') \subseteq C$. Donc pour tous types $\tau \in \Downarrow A$ et $\tau' \in \Uparrow B$, $dec(\tau \leq \tau') \subseteq C$. □

Lemme 5.7 *Étant donné un système clos C, $\Psi_\epsilon(A, B)$ est défini pour tous les ensembles $A, B \subseteq V(C)$ vérifiant la condition 5.5.*

Démonstration : Soit $\tau \in \Downarrow A$ et $\tau' \in \Uparrow B$. De par le lemme 5.6, $dec(\tau \leq \tau')$ est défini, donc $\tau(\epsilon) \leq_{\mathcal{K}} \tau'(\epsilon)$. On pose $S = \{\tau(\epsilon) \mid \tau \in \Uparrow B\}$ et $I = \{\tau(\epsilon) \mid \tau \in \Downarrow A\}$. On a donc, pour tous constructeurs $\kappa \in I$ et $\kappa' \in S$, $\kappa \leq_{\mathcal{K}} \kappa'$. Comme C est clos et A et B sont non vides, $\Downarrow A \neq \emptyset$ et $\Uparrow B \neq \emptyset$. Donc $\sqcup I$ et $\sqcap S$ sont définis et $\sqcup I \leq_{\mathcal{K}} \sqcap S$. Comme $\downarrow_{ar(\sqcup I)} \sqcap S = \{\kappa' \leq_{\mathcal{K}} \sqcap S \mid ar(\kappa') \cap ar(\sqcap S) \subseteq ar(\sqcup I)\}$ et comme $\sqcup I \leq_{\mathcal{K}} \sqcap S$ et $ar(\sqcup I) \cap ar(\sqcap S) \subseteq ar(\sqcup I)$, $\downarrow_{ar(\sqcup I)} \sqcap S$ contient $\sqcup I$. Donc $\downarrow_{ar(\sqcup I)} \sqcap S \neq \emptyset$ et $\Psi_\epsilon(A, B)$ est définie. □

Le lemme 5.8 permet de justifier la définition 5.9 de la fonction partielle Ψ_V, qui constituera la dernière brique pour construire Ψ.

Lemme 5.8 *Soient C un système clos et $A, B \subseteq V(C)$ deux ensembles de variables vérifiant la condition 5.5. Pour toute étiquette $l \in ar(\Psi_\epsilon(A, B))$, si $l \in \mathcal{L}^+$ (resp. si $l \in \mathcal{L}^-$), $((\Downarrow A)/l, (\Uparrow B)/l)$ (resp. $((\Uparrow B)/l, (\Downarrow A)/l)$) vérifie également cette condition.*

Démonstration : Soient $S = \{\tau(\epsilon) \mid \tau \in \Uparrow B\}$ et $I = \{\tau(\epsilon) \mid \tau \in \Downarrow A\}$. Par définition, et par le lemme 4.5, $ar(\Psi_\epsilon(A, B)) = ar(\sqcup(\downarrow_{ar(\sqcup I)} \sqcap S)) \subseteq ar(\sqcup I) \cap ar(\sqcap S) \subseteq \bigcup_{\kappa \in I} ar(\kappa) \cap \bigcup_{\kappa' \in S} ar(\kappa')$. Donc, pour tout $l \in ar(\Psi_\epsilon(A, B))$, il existe un type $\tau \in \Downarrow A$ tel que τ/l est défini et un type $\tau' \in \Uparrow B$ tel que τ'/l est défini. Donc $(\Downarrow A)/l \neq \emptyset$ et $(\Uparrow B)/l \neq \emptyset$.

Soit $l \in ar^+(\Psi_\epsilon(A, B))$. Soient deux variables $\alpha \in (\Downarrow A)/l$ et $\beta \in (\Uparrow B)/l$. Il existe un type $\tau_\alpha \in \Downarrow A$ tel que $\tau_\alpha/l = \alpha$ et un type $\tau_\beta \in \Uparrow B$ tel que $\tau_\beta/l = \beta$. Comme, par le lemme 5.6, $dec(\tau_\alpha \leq \tau_\beta) \subseteq C$ et comme $l \in ar(\tau_\alpha(\epsilon)) \cap ar(\tau_\beta(\epsilon))$, on obtient $\alpha \leq \beta \in C$. De la même manière, pour toute étiquette $l \in ar^-(\Psi_\epsilon(A, B))$, pour toutes variables $\beta \in (\Uparrow B)/l$ et $\alpha \in (\Downarrow A)/l$, on obtient $\beta \leq \alpha \in C$. □

Définition 5.9 *Étant donné un système clos C, la fonction partielle $\Psi_V : 2^{V(C)} \times 2^{V(C)} \times \mathcal{L} \to 2^{V(C)} \times 2^{V(C)}$ est définie inductivement comme suit :*
$\Psi_V(A, B, \epsilon) = (A, B)$, et si $\Psi_V(A, B, w) = (E, F)$, alors
- *pour toute étiquette $l \in ar^+(\Psi_\epsilon(E, F))$, $\Psi_V(A, B, w.l) = ((\Downarrow E)/l, (\Uparrow F)/l)$,*
- *pour toute étiquette $l \in ar^-(\Psi_\epsilon(E, F))$, $\Psi_V(A, B, w.l) = ((\Uparrow F)/l, (\Downarrow E)/l)$.*

On peut remarquer que Ψ_V est définie pour les couples vérifiant la condition 5.5.

Exemple 5.4 : En reprenant le système C' défini dans l'exemple 5.2, on a $\Psi_V(\{\alpha_1\}, \{\alpha_1\}, \epsilon) = (\{\alpha_1\}, \{\alpha_1\})$ et $\Psi_\epsilon(\{\alpha_1\}, \{\alpha_1\}) = \kappa_3[l_2]$. On a $\Downarrow \{\alpha_1\} = \{\kappa_4(\alpha_1, \alpha_1)\}$ et $\Uparrow \{\alpha_1\} = \{\kappa_2(\beta_1, \alpha_1), \kappa_2(\beta_2, \alpha_2)\}$. On obtient donc $\Psi_V(\{\alpha_1\}, \{\alpha_1\}, l_2) = (\Downarrow \{\alpha_1\}/l_2, \Uparrow \{\alpha_1\}/l_2) = (\{\alpha_1\}, \{\alpha_1, \alpha_2\})$.
On peut vérifier que $\Psi_\epsilon(\{\alpha_1\}, \{\alpha_1, \alpha_2\}) = \kappa_3[l_2]$, et que $\Psi_V(\{\alpha_1\}, \{\alpha_1\}, l_2.l_2) = \Psi_V(\{\alpha_1\}, \{\alpha_1, \alpha_2\}, l_2) = (\{\alpha_1\}, \{\alpha_1, \alpha_2\})$. ◇

On peut à présent définir la fonction Ψ qui va nous permettre d'obtenir la solution d'un système clos :

Définition 5.10 *Soient C un système clos et $A, B \subseteq V(C)$ deux ensembles non vides de variables. $\Psi : 2^{V(C)} \times 2^{V(C)} \rightarrow (\mathcal{L}^* \rightarrow \mathcal{K})$ est une fonction partielle définie par $\Psi(A,B)(w) = \Psi_\epsilon(\Psi_V(A,B,w))$.*

Le lemme 5.11 établit que, pour A et B vérifiant la condition 5.5, $\Psi(A,B)$ est un type.

Lemme 5.11 *Soient C un système clos et $A, B \subseteq V(C)$ deux ensembles de variables vérifiant la condition 5.5, alors $\Psi(A,B)$ est défini et est un type.*

Démonstration : Par le lemme 5.8, si $\Psi_V(A,B,w)$ vérifie la condition 5.5, et si $\Psi_V(A,B,w.l)$ est défini, alors $\Psi_V(A,B,w.l)$ vérifie la condition 5.5. Comme (A,B) vérifie la condition 5.5, on montre par récurrence que si $\Psi_V(A,B,w)$ est défini, alors elle vérifie la condition 5.5, et donc que $\Psi(A,B)(w)$ est défini.

Il suffit maintenant de montrer que $\Psi(A,B)$ vérifie les trois conditions de la définition 3.2.

1. Par définition, pour A et B fixés, $\Psi_V(A,B,.)$ est close par préfixe, donc $\Psi(A,B)$ est close par préfixe.
2. $\Psi_V(A,B,\epsilon)$ est défini, donc $\epsilon \in dom(\Psi(A,B))$.
3. Soit $w \in dom(\Psi(A,B))$. $\Psi(A,B)(w) = \Psi_\epsilon(\Psi_V(A,B,w))$. Par définition, $\Psi_V(A,B,w.l)$ est défini si et seulement si $l \in ar(\Psi_\epsilon(\Psi_V(A,B,w))$ c'est à dire si et seulement si $l \in ar(\Psi(A,B)(w))$. □

Exemple 5.5 : En reprenant le système C' de l'exemple 5.2, on a $\Psi(\{\alpha_1\},\{\alpha_1\})(\epsilon) = \kappa_3$ et pour toute position $w = l_2.\cdots.l_2$, $\Psi(\{\alpha_1\},\{\alpha_1\})(w) = \kappa_3$. D'où $\Psi(\{\alpha_1\},\{\alpha_1\}) = \kappa_3(\kappa_3(\kappa_3(\ldots)))$. ◇

Nous introduisons ici un lemme qui caractérise $\Psi(A,B)$ par rapport à ses sous-termes :

Lemme 5.12 *Soient C un système clos et $A, B \subseteq V(C)$ deux ensembles de variables vérifiant la condition 5.5. Pour toute étiquette $l \in ar(\Psi_\epsilon(A,B))$, si $l \in \mathcal{L}^+$ (resp. si $l \in \mathcal{L}^-$), alors $\Psi(A,B)/l = \Psi((\Downarrow A)/l, (\Uparrow B)/l)$ (resp. $\Psi(A,B)/l = \Psi((\Uparrow B)/l, (\Downarrow A)/l)$).*

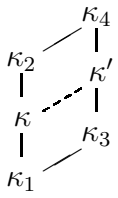
Démonstration : Il suffit de montrer par induction sur les positions $w \in \mathcal{L}^*$ que pour toutes les paires (A,B) vérifiant la condition 5.5, pour toutes les étiquettes $l \in ar(\Psi_\epsilon(A,B))$, $\Psi_V(A,B,l.w) = \Psi_V((\Downarrow A)/l, (\Uparrow B)/l, w)$ si l est positif et $\Psi_V(A,B,l.w) = \Psi((\Uparrow B)/l, (\Downarrow A)/l, w)$ si l est négatif.

– Considérons $w = \epsilon$.
  Si $l \in \mathcal{L}^+$ alors $\Psi_V(A, B, l) = ((\Downarrow A)/l, (\Uparrow B)/l) = \Psi_V((\Downarrow A)/l, (\Uparrow B)/l, \epsilon)$.
  Si $l \in \mathcal{L}^-$ alors $\Psi_V(A, B, l) = ((\Uparrow B)/l, (\Downarrow A)/l) = \Psi_V((\Uparrow B)/l, (\Downarrow A)/l, \epsilon)$.
– Considérons maintenant $w = w'.l'$. Supposons $l \in \mathcal{L}^+$.
  Par récurrence $\Psi_V(A, B, l.w') = \Psi_V((\Downarrow A)/l, (\Uparrow B)/l, w')$. En appliquant la définition 5.9, on obtient $\Psi_V(A, B, l.w'.l') = \Psi_V((\Downarrow A)/l, (\Uparrow B)/l, w'.l')$.
  De même, si $l \in \mathcal{L}^-$, $\Psi_V(A, B, l.w'.l') = \Psi_V((\Uparrow B)/l, (\Downarrow A)/l, w'.l')$. □

Le lemme suivant et ses corollaires vont nous permettre de démontrer que la valuation $\rho : \alpha \mapsto \Psi(\{\alpha\}, \{\alpha\})$ est une solution de C. Ils établissent des conditions sous lesquelles $\Psi(A, B) \leq \Psi(E, F)$.

Lemme 5.13 *Soient $\kappa_1, \kappa_2, \kappa_3, \kappa_4 \in \mathcal{K}$ quatre constructeurs de type tels que $\kappa_1 \leq_{\mathcal{K}} \kappa_2$, $\kappa_1 \leq_{\mathcal{K}} \kappa_3$, $\kappa_2 \leq_{\mathcal{K}} \kappa_4$ et $\kappa_3 \leq_{\mathcal{K}} \kappa_4$. Alors $\sqcup(\downarrow_{ar(\kappa_1)} \kappa_2) \leq_{\mathcal{K}} \sqcup(\downarrow_{ar(\kappa_3)} \kappa_4)$.*

Démonstration :
Notons $\kappa = \sqcup(\downarrow_{ar(\kappa_1)} \kappa_2)$ et $\kappa' = \sqcup(\downarrow_{ar(\kappa_3)} \kappa_4)$. De par le lemme 4.5, $ar(\kappa) = ar(\kappa_2) \cap \bigcup_{\kappa_a \in \downarrow_{ar(\kappa_1)} \kappa_2} ar(\kappa_a) \subseteq ar(\kappa_1) \cap ar(\kappa_2)$. Comme $\kappa_1 \leq_{\mathcal{K}} \kappa_3 \leq_{\mathcal{K}} \kappa_4$, $ar(\kappa_1) \cap ar(\kappa_4) \subseteq ar(\kappa_3)$. Donc $ar(\kappa) \cap ar(\kappa_4) \subseteq ar(\kappa_2) \cap ar(\kappa_1) \cap ar(\kappa_4) \subseteq ar(\kappa_2) \cap ar(\kappa_3) \cap ar(\kappa_4) \subseteq ar(\kappa_3)$. De plus $\kappa \leq_{\mathcal{K}} \kappa_2 \leq_{\mathcal{K}} \kappa_4$, donc $\kappa \in \downarrow_{ar(\kappa_3)} \kappa_4$, d'où $\kappa \leq_{\mathcal{K}} \kappa'$.



□

Corollaire 5.14 *Soit C un système clos et soient (A, B) et (E, F) deux paires d'ensembles de variables de C vérifiant la condition 5.5. Si $\Downarrow A \subseteq \Downarrow E$ et $\Uparrow F \subseteq \Uparrow B$ alors $\Psi_\epsilon(A, B) \leq_{\mathcal{K}} \Psi_\epsilon(E, F)$.*

Démonstration : Notons $I_A = \{\tau(\epsilon) \mid \tau \in \Downarrow A\}$, $S_B = \{\tau(\epsilon) \mid \tau \in \Uparrow B\}$, $I_E = \{\tau(\epsilon) \mid \tau \in \Downarrow E\}$ et $S_F = \{\tau(\epsilon) \mid \tau \in \Uparrow F\}$. On a $I_A \subseteq I_E$ et $S_F \subseteq S_B$. Donc $\sqcup I_A \leq_{\mathcal{K}} \sqcup I_E$ et $\sqcap S_B \leq_{\mathcal{K}} \sqcap S_F$. Par le lemme 5.6, pour tous types $\tau \in \Downarrow A$ et $\tau' \in \Uparrow B$, $dec(\tau \leq \tau') \subseteq C$, donc $\tau(\epsilon) \leq_{\mathcal{K}} \tau'(\epsilon)$. Donc $\sqcup I_A \leq_{\mathcal{K}} \sqcap S_B$. De même, $\sqcup I_E \leq_{\mathcal{K}} \sqcap S_F$. Donc, par le lemme 5.13, $\Psi_\epsilon(A, B) = \sqcup(\downarrow_{ar(\sqcup I_A)} \sqcap S_B) \leq_{\mathcal{K}} \sqcup(\downarrow_{ar(\sqcup I_E)} \sqcap S_F) = \Psi_\epsilon(E, F)$. □

Corollaire 5.15 *Soit C un système clos et soient (A, B) et (E, F) deux paires d'ensembles de variables de C vérifiant la condition 5.5. Si $\Downarrow A \subseteq \Downarrow E$ et $\Uparrow F \subseteq \Uparrow B$ alors $\Psi(A, B) \leq \Psi(E, F)$.*

Démonstration : On montre par récurrence que pour tout $n \in \mathbb{N}$, $\Psi(A, B) \leq_n \Psi(E, F)$. Le cas $n = 0$ est trivial. Montrons le cas $n + 1$. Par le corollaire 5.14, $\Psi_\epsilon(A, B) \leq_{\mathcal{K}} \Psi_\epsilon(E, F)$, c'est-à-dire $\Psi(A, B)(\epsilon) \leq_{\mathcal{K}} \Psi(E, F)(\epsilon)$.

Soit une étiquette $l \in ar^+(\Psi_\epsilon(A, B)) \cap ar^+(\Psi_\epsilon(E, F))$. Par le lemme 5.8, $(\Downarrow A)/l$, $(\Uparrow B)/l$, $(\Downarrow E)/l$ et $(\Uparrow F)/l$ sont non vides. Comme $\Downarrow A \subseteq \Downarrow E$,

$(\Downarrow A)/l \subseteq (\Downarrow E)/l$ et $\Downarrow((\Downarrow A)/l) \subseteq \Downarrow((\Downarrow E)/l)$. De même, $\Uparrow((\Uparrow F)/l) \subseteq \Uparrow((\Uparrow B)/l)$. Par récurrence, on obtient donc $\Psi((\Downarrow A)/l, (\Uparrow B)/l) \leq_n \Psi((\Downarrow E)/l, (\Uparrow F)/l)$, c'est-à-dire $\Psi(A,B)/l \leq_n \Psi(E,F)/l$.

Soit une étiquette $l \in ar^-(\Psi_\epsilon(A,B)) \cap ar^-(\Psi_\epsilon(E,F))$. Par le lemme 5.8, $(\Downarrow A)/l$, $(\Uparrow B)/l$, $(\Downarrow E)/l$ et $(\Uparrow F)/l$ sont non vides. Comme $\Downarrow A \subseteq \Downarrow E$, $(\Downarrow A)/l \subseteq (\Downarrow E)/l$ et $\Uparrow((\Downarrow A)/l) \subseteq \Uparrow((\Downarrow E)/l)$. De même, $\Downarrow((\Uparrow F)/l) \subseteq \Downarrow((\Uparrow B)/l)$. Par récurrence, on obtient donc $\Psi((\Uparrow F)/l, (\Downarrow E)/l) \leq_n \Psi((\Uparrow B)/l, (\Downarrow A)/l)$, c'est-à-dire $\Psi(E,F)/l \leq_n \Psi(A,B)/l$.

Donc, $\Psi(A,B) \leq_{n+1} \Psi(E,F)$.

Donc pour tout $n \in \mathbb{N}$, $\Psi(A,B) \leq_n \Psi(E,F)$, d'où $\Psi(A,B) \leq \Psi(E,F)$. □

On peut à présent démontrer le théorème 5.3 :

Démonstration (du théorème 5.3) : Soit C un système clos. On peut supposer sans perte de généralité que $\alpha \leq \alpha \in C$ pour toutes les variables α de C. Considérons la valuation $\rho : \alpha \mapsto \Psi(\{\alpha\}, \{\alpha\})$. Considérons la contrainte $c = \tau_1 \leq \tau_2 \in C$. On distingue quatre cas :

- $\tau_1 = \alpha \in \mathcal{V}$ et $\tau_2 = \beta \in \mathcal{V}$. Comme C est clos, si $\tau \leq \alpha \in C$ alors $\tau \leq \beta \in C$ et si $\beta \leq \tau \in C$ alors $\alpha \leq \tau \in C$. Donc $\Downarrow \alpha \subseteq \Downarrow \beta$ et $\Uparrow \beta \subseteq \Uparrow \alpha$. On peut donc appliquer le corollaire 5.15 et obtenir $\rho(\alpha) = \Psi(\{\alpha\}, \{\alpha\}) \leq \Psi(\{\beta\}, \{\beta\}) = \rho(\beta)$.
- $\tau_1 = \alpha \in \mathcal{V}$ et $\tau_2 \notin \mathcal{V}$. Comme $\tau_2 \in \Uparrow \alpha$, $\rho(\alpha)(\epsilon) = \Psi_\epsilon(\{\alpha\}, \{\alpha\}) \leq_{\mathcal{K}} \sqcap(\{\tau(\epsilon) \mid \tau \in \Uparrow \alpha\} \leq_{\mathcal{K}} \tau_2(\epsilon)$.
  Soit une étiquette $l \in ar^+(\rho(\alpha)(\epsilon)) \cap ar^+(\tau_2(\epsilon))$. Comme τ_2 est un type plat, τ_2/l est une certaine variable β, et $\rho(\tau_2)/l = \Psi(\{\beta\}, \{\beta\})$. Par ailleurs, $\rho(\alpha)/l = \Psi((\Downarrow \alpha)/l, (\Uparrow \alpha)/l)$. Comme $\beta \in (\Uparrow \alpha)/l$, $\Uparrow \beta \subseteq \Uparrow((\Uparrow \alpha)/l)$. Comme C est clos, pour tout type $\tau \in \Downarrow \alpha$, $dec(\tau \leq \tau_2) \subseteq C$, donc pour tout $\alpha' \in (\Downarrow \alpha)/l$, $\alpha' \leq \beta \in C$. On en déduit que $\Downarrow((\Downarrow \alpha)/l) \subseteq \Downarrow \beta$. Par le corollaire 5.15, on obtient $\Psi((\Downarrow \alpha)/l, (\Uparrow \alpha)/l) \leq \Psi(\{\beta\}, \{\beta\})$, c'est-à-dire $\rho(\alpha)/l \leq \rho(\tau_2)/l$.
  On montre de la même manière que si $l \in ar^-(\rho(\alpha)(\epsilon)) \cap ar^-(\tau_2(\epsilon))$, alors $\rho(\tau_2)/l \leq \rho(\alpha)/l$.
  Donc par la propriété 3.6 $\rho(\alpha) \leq \rho(\tau_2)$.
- $\tau_1 \notin \mathcal{V}$ et $\tau_2 = \alpha \in \mathcal{V}$. On note $I = \{\tau(\epsilon) \mid \tau \in \Downarrow \alpha\}$ et $S = \{\tau(\epsilon) \mid \tau \in \Uparrow \alpha\}$. Comme $\tau_1 \in \Downarrow \alpha$, $\tau_1(\epsilon) \leq_{\mathcal{K}} \sqcup I$. Comme $\sqcup I \leq_{\mathcal{K}} \sqcap S$, $\sqcup I \leq_{\mathcal{K}} \sqcup(\downarrow_{ar(\sqcup I)} \sqcap S) = \Psi_\epsilon(\{\alpha, \alpha\})$. Donc $\rho(\tau_1)(\epsilon) \leq_{\mathcal{K}} \rho(\alpha)(\epsilon)$. De la même manière que dans le cas précédent, on montre que pour toute étiquette $l \in ar(\rho(\alpha)(\epsilon)) \cap ar(\tau_1(\epsilon))$, si $l \in \mathcal{L}^+$, $\rho(\tau_1)/l \leq \rho(\alpha)/l$ et si $l \in \mathcal{L}^-$, $\rho(\alpha)/l \leq \rho(\tau_1)/l$. On en déduit donc que $\rho(\tau_1) \leq \rho(\alpha)$.
- $\tau_1 \notin \mathcal{V}$ et $\tau_2 \notin \mathcal{V}$. Comme C est clos, $\tau_1(\epsilon) \leq_{\mathcal{K}} \tau_2(\epsilon)$. Soit une étiquette $l \in ar^+(\tau_1(\epsilon)) \cap ar^+(\tau_2(\epsilon))$. Comme C est clos, on a également $\tau_1/l \leq \tau_2/l \in C$.

Comme τ_1 et τ_2 sont des types plats, $\tau_1/l, \tau_2/l \in \mathcal{V}$, donc $\rho(\tau_1/l) \leq \rho(\tau_2/l)$.
De même, pour toute étiquette $l \in ar^-(\tau_1(\epsilon)) \cap ar^-(\tau_2(\epsilon))$, $\rho(\tau_2/l) \leq \rho(\tau_1/l)$.
Donc, par la propriété 3.6, $\rho(\tau_1/l) \leq \rho(\tau_2/l)$.

On en déduit donc que $\rho \models C$, c'est-à-dire que C est satisfiable. □

Exemple 5.6 : En reprenant le système clos C' défini dans l'exemple 5.2, on a : $\rho(\alpha_1) = \kappa_3(\kappa_3(\kappa_3(\ldots)))$, $\rho(\alpha_2) = \kappa_3(\kappa_3(\kappa_3(\ldots)))$, $\rho(\beta_1) = \kappa_0$ et $\rho(\beta_2) = \kappa_1$. ◇

Cette démonstration du théorème 5.3 nous permet de déduire le corollaire suivant :

Corollaire 5.16 *Soit $\mathcal{S}$ une signature bien formée. Tout système de contraintes clos construit sur $\mathcal{F}(\mathcal{S}_\mathcal{V})$ est satisfiable dans $\mathcal{R}(\mathcal{S})$.*

Démonstration : Soit C un système clos. On montre que pour tous A, B vérifiant la condition 5.5, $\Psi(A, B)$ est un type régulier. Par le lemme 5.12, $\Psi(A, B)/l = \Psi(E, F)$, où E, F vérifie la condition 5.5. En itérant ce raisonnement, on obtient pour toute position $w \in dom(\Psi(A, B))$, $\Psi(A, B)/w = \Psi(E, F)$ où E, F vérifie la condition 5.5. Or il n'y a qu'un nombre fini d'ensembles E, F inclus dans $V(C)$. Donc $\Psi(A, B)$ n'a qu'un nombre fini de sous-termes. C'est donc un type régulier. Comme pour tout $\alpha \in V(C), \rho(\alpha) = \Psi(\{\alpha\}, \{\alpha\})$, c'est un type régulier. □

5.3 Algorithmes de clôture

Nous présentons à présent un algorithme pour tester la satisfiabilité d'un système de contraintes de sous-typage C par calcul de clôture. Le test est divisé en deux parties. On calcule tout d'abord un ensemble de systèmes pré-clos, appelés *pré-clôtures*, dont les ensembles de solutions recouvrent exactement l'ensemble des solutions de C. Puis pour chaque système pré-clos C' ainsi calculé, on tente de trouver un système clos C'', appelé *clôture* équivalent à C'. Un échec signifie alors que C' n'est pas satisfiable.

Pré-clôtures

Le théorème 5.17 ci-dessous donne des conditions suffisantes sur une signature $\mathcal{S} = (\mathcal{K}, \leq_\mathcal{K}, \mathcal{L}^+, \mathcal{L}^-, ar)$ pour calculer un ensemble de pré-clôtures équivalent à un système de contraintes C. On note $\overline{\mathcal{K}}$ l'ensemble des éléments maximaux de $\mathcal{K}$ et $\underline{\mathcal{K}}$ l'ensemble de ses éléments minimaux.

Théorème 5.17 *[14] Si $\mathcal{K}$ vérifie les conditions suivantes :*

1. *Pour tout constructeur $\kappa \in \overline{\mathcal{K}} \cup \underline{\mathcal{K}}$, $ar(\kappa) = \emptyset$.*

2. *Pour tout constructeur $\kappa \in \mathcal{K}$, il existe un constructeur $\kappa_1 \in \underline{\mathcal{K}}$ et un constructeur $\kappa_2 \in \overline{\mathcal{K}}$ tels que $\kappa_1 \leq_{\mathcal{K}} \kappa \leq_{\mathcal{K}} \kappa_2$.*

Pour tout système de contraintes C, soit $pc(C)$ l'ensemble des pré-clôtures de C défini par :

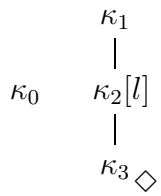
$$pc(C) = \left\{ C \cup \bigcup_{\alpha \in V(C)} \{\tau_\alpha \leq \alpha, \alpha \leq \tau'_\alpha\} \mid \tau_\alpha(\epsilon) \in \underline{\mathcal{K}}, \tau'_\alpha(\epsilon) \in \overline{\mathcal{K}} \right\}.$$

Tous les éléments de $pc(C)$ sont pré-clos et l'union de leurs ensembles de solutions est égal à l'ensemble des solutions de C.

Démonstration : Comme, pour tout $\kappa \in \overline{\mathcal{K}} \cup \underline{\mathcal{K}}$, $ar(\kappa) = \emptyset$, on a que pour tout système $C' \in pc(C)$, $V(C') = V(C)$. Donc, par construction les éléments de $pc(C)$ sont pré-clos. Pour tous les systèmes $C' \in pc(C)$, $C \subseteq C'$, donc $\rho \models C' \Rightarrow \rho \models C$. Montrons à présent que si $\rho \models C$ alors il existe un système $C' \in pc(C)$ tel que $\rho \models C'$. De par la condition 2), pour toute variable $\alpha \in V(C)$, on peut trouver $\kappa_\alpha \in \underline{\mathcal{K}}$ et $\kappa'_\alpha \in \overline{\mathcal{K}}$ tels que $\kappa_\alpha \leq_{\mathcal{K}} \rho(\alpha)(\epsilon) \leq_{\mathcal{K}} \kappa'_\alpha$. Par la condition 1), on obtient $\rho \models \tau_\alpha \leq \alpha, \alpha \leq \tau'_\alpha$, avec $\tau_\alpha(\epsilon) = \kappa_\alpha$ et $\tau'_\alpha(\epsilon) = \kappa'_\alpha$. En itérant ce procédé pour toutes les variables de C, on obtient un système $C' \in pc(C)$ tel que $\rho \models C'$. Donc l'union des ensembles de solutions des éléments de $pc(C)$ est égal à l'ensemble des solutions de C. □

Exemple 5.7 :
Soit le système $C = \kappa_2(\delta) \leq \alpha, \alpha \leq \kappa_2(\beta)$ défini par rapport à la signature correspondant au schéma ci-contre. On a

$$\begin{aligned} pc(C) \quad = \{ \quad & C \cup \{\tau_1 \leq \beta, \beta \leq \tau_2, \tau_3 \leq \delta, \delta \leq \tau_4\} \\ & \mid \tau_1, \tau_3 \in \{\kappa_0, \kappa_3\} \wedge \tau_2, \tau_4 \in \{\kappa_0, \kappa_1\} \quad \} \end{aligned}$$



◇

Clôture d'un système pré-clos

Étant donné un système pré-clos C, il est possible d'en calculer la clôture, comme dans l'algorithme de Trifonov et Smith [70] de complexité en temps $O(n^3)$, où n est la taille du système de contraintes. L'algorithme procède en calculant la suite $C, C^1, C^2, \ldots$ définie par :

$$C^{n+1} = C^n \cup \bigcup_{c \in C^n} dec(c) \cup \bigcup_{\{\tau \leq \alpha, \alpha \leq \tau'\} \subseteq C^n} dec(\tau \leq \tau')$$

On vérifie facilement que C^{n+1} est équivalent à C^n lorsque C^{n+1} est défini. Si C^{n+1} n'est pas défini, alors C^n n'est pas satisfiable, et donc C ne l'est pas non plus.

En effet cela correspond au cas où on essaie d'appliquer *dec* sur une contrainte $\tau \leq \tau'$ alors que $\tau(\epsilon) \not\leq_{\mathcal{K}} \tau'(\epsilon)$, ce qui signifie que $\tau \leq \tau'$ n'est pas satisfiable. Si tous les C^n sont définis, alors la suite atteint un point fixe, noté C^∞, car on passe de C^n à C^{n+1} en ajoutant à C^n des contraintes formées à partir des sous-termes des types présents dans C, et qui eux-mêmes sont en nombre fini. Comme on n'introduit pas de nouvelles variables, C^∞ est pré-clos. Par construction, il vérifie les conditions de la définition 5.2. C^∞ est donc clos, c'est à dire satisfiable de par le théorème 5.3. Comme C est équivalent à C^∞, il est satisfiable.

Exemple 5.8 : Si on reprend le système C de l'exemple 5.7, seuls deux systèmes peuvent être clos : $C_1 = C \cup \{\kappa_0 \leq \beta, \beta \leq \kappa_0, \kappa_0 \leq \delta, \delta \leq \kappa_0\}$ et $C_2 = C \cup \{\kappa_3 \leq \beta, \beta \leq \kappa_1, \kappa_3 \leq \delta, \delta \leq \kappa_1\}$. Leurs clôtures sont respectivement $C'_1 = C_1 \cup \{\delta \leq \beta\}$ et $C'_2 = C_2 \cup \{\delta \leq \beta\}$. En revanche $C_3 = C \cup \{\kappa_0 \leq \beta, \beta \leq \kappa_0, \kappa_3 \leq \delta, \delta \leq \kappa_1\}$ ne peut être clos, car $dec(\{\kappa_2(\delta) \leq \alpha, \alpha \leq \kappa_2(\beta)\} = \{\delta \leq \beta\}$, et car $dec(\{\delta \leq \beta\}) = \kappa_0 \leq \kappa_1$ et car cette dernière inégalité ne être satisfaite. ◇

Test de satisfiabilité

Si $\mathcal{K}$ vérifie les conditions du théorème 5.17 et si $\overline{\mathcal{K}}$ et $\underline{\mathcal{K}}$ sont finis, il est alors possible d'énumérer les éléments de $pc(C)$. Comme ces éléments sont pré-clos, on peut tester leur satisfiabilité en utilisant l'algorithme de clôture décrit ci-dessus. On obtient ainsi un algorithme pour tester la satisfiabilité des systèmes de contraintes non pré-clos dans les quasi-treillis avec un nombre fini d'extrema, chacun d'arité vide. La complexité en temps de cet algorithme est $O(n^3 m^v M^v)$ où n est la taille du système de contraintes, v est le nombre de variables non bornées dans C, m la taille de $\underline{\mathcal{K}}$ et M la taille de $\overline{\mathcal{K}}$. On peut remarquer que si $(\mathcal{K}, \leq_{\mathcal{K}})$ forme un treillis avec un maximum $\top$ et un minimum $\bot$, il n'existe qu'une seule pré-clôture pour un système C. Le calcul de pré-clôture n'est alors pas nécessaire et l'algorithme devient identique à celui de Trifonov et Smith [70], avec la même complexité $O(n^3)$.

Exemple 5.9 : Le système C défini dans l'exemple 5.7 est satisfiable, car l'un des éléments de sa pré-clôture est satisfiable (c'est par exemple le cas du système C_1 défini dans l'exemple 5.8, puisque sa clôture C'_1 est définie). ◇

L'algorithme décrit ci-dessus nous permet d'énoncer le résultat suivant :

Théorème 5.18 *[14] Le problème de la satisfiabilité des contraintes de sous-typage dans les quasi-treillis avec un nombre fini d'extrema, chacun d'arité vide, et vérifiant la conition 2 du théorème 5.17, est NP-complet.*

Démonstration : La satisfiabilité des système pré-clos peut-être testée en temps polynomial. Par le théorème 5.17, l'ensemble des pré-clôtures d'un système peut-être trouvé en énumérant, parmi un ensemble fini, les bornes possibles pour les

variables non bornées. Le problème de satisfiabilité est donc dans NP. Afin de prouver la NP-complétude, nous utilisons le résultat de Pratt et Tiuryn [59] qui énonce que la satisfiabilité des contraintes de sous-typage dans les couronnes, en anglais "n-crowns", est NP-complète. Une couronne est un ensemble partiellement ordonné à $2n$ éléments $\kappa_0, \ldots, \kappa_{2n-1}$, chacun d'arité vide et ordonné de telle manière à ce que les seules comparaisons soient $\kappa_{2i} \leq_{\mathcal{K}} \kappa_{2i\pm1}$ et $\kappa_0 \leq_{\mathcal{K}} \kappa_{2n-1}$. Il est clair que pour $n \geq 3$, les couronnes sont des quasi-treillis avec un nombre finis d'extrema, chacun d'arité vide. Le problème de satisfiabilité dans les quasi-treillis est donc NP-complet. □

Corollaire 5.19 *Soit une signature $\mathcal{S}$ vérifiant les conditions du théorème 5.18. Un système de contraintes C construit sur $\mathcal{F}(\mathcal{S}_\mathcal{V})$ est satisfiable sur l'ensemble $\mathcal{T}(\mathcal{S})$ des types infinis si et seulement si il l'est sur l'ensemble des types réguliers $\mathcal{R}(\mathcal{S})$.*

Démonstration : De par le corollaire 5.16 tout système de contrainte clos est satisfiable sur $\mathcal{R}(\mathcal{S})$. Comme tout système pré-clos est satisfiable si et seulement si sa clôture est définie, on obtient que tout système pré-clos est satisfiable dans les types infinis si et seulement si il l'est dans les types réguliers. En appliquant un raisonnement similaire à la démonstration du théorème précédent, on obtient que tout système de contraintes de sous-typage est satisfiable dans les types infinis si et seulement si il l'est dans les types réguliers. □

La première condition du théorème 5.17 impose que les extrema du quasi-treillis aient une arité vide. On peut noter que sans cette condition, l'introduction d'une nouvelle contrainte $\tau_\alpha \leq \alpha$ (ou bien $\alpha \leq \tau'_\alpha$) peut également introduire de nouvelles variables non bornées apparaissant dans τ_α. Ces variables doivent alors être elles-mêmes bornées en introduisant de nouvelle contraintes. L'itération de ce phénomène peut alors produire une infinité de variables. On ne peut donc pas appliquer dans ce cas l'algorithme décrit ci-dessus. Ce résultat laisse donc ouverte la question de la décidabilité de la satisfiabilité des contraintes de sous-typage non structurelles dans les quasi-treillis dans lesquels des extrema ont une arité non vide. Le chapitre suivant porte sur le calcul explicite de solutions de systèmes de contraintes de sous-typage dans les quasi-treillis et apportera quelques éléments de réponse à ce problème, notamment en relâchant l'hypothèse d'arité sur une partie des extrema.

Chapitre 6

Calcul de bornes pour les contraintes de sous-typage

Dans ce chapitre, nous nous intéressons à calculer des solutions explicites à un système de contraintes. Afin d'obtenir ces solutions, nous améliorons l'algorithme précédent en introduisant un calcul de bornes optimales pour les variables des systèmes de contraintes de sous-typage dans les quasi-treillis. Ce calcul de bornes est effectué en simplifiant les systèmes de contraintes, ce qui correspond à l'opération de canonisation que l'on peut trouver dans les travaux de Pottier [57] et de Trifonov et Smith [70]. Il permet, dans le cas où tous les constructeurs de types sont covariants, d'obtenir des solutions maximales et minimales pour un système de contraintes. Cela nous permettra d'inférer des types pour les variables et les prédicats, comme expliqué dans les chapitres 8 et 9. Enfin, ce calcul de bornes va nous permettre d'apporter un début de réponse au problème d'arité des extrema rencontré au chapitre précédent, en retirant la condition exprimée dans le théorème 5.17 sur les minima ou sur les maxima du quasi-treillis des constructeurs de types.

L'algorithme de calcul de bornes, que nous avons déjà décrit en partie dans [14], est présenté sous forme d'un système de réécriture sur les contraintes de sous-typage. Il correspond, pour partie à la mise en forme de règles de réécriture de l'algorithme de canonisation incrémentale de Pottier pour les treillis [56, 22, 12], et pour partie à des règles spécifiques au traitement des quasi-treillis [14]. Nous montrons la terminaison et la correction du système de réécriture ainsi obtenu, puis nous décrivons l'implantation dans les Constraint Handling Rules [25] (ou CHR) de ce système de réécriture, en particulier au niveau de la stratégie d'application des règles qui a été utilisée. Enfin nous comparons l'efficacité du solveur CHR à celle du solveur Wallace de Pottier [58], et évaluons le coût en pratique du passage d'une structure de treillis à une structure de quasi-treillis.

Nous supposerons par la suite que l'ensemble des types est basé sur une si-

gnature bien formée $\mathcal{S} = (\mathcal{K}, \leq_{\mathcal{K}}, \mathcal{L}^+, \mathcal{L}^-, ar)$. Afin d'effectuer le calcul de bornes, on va plonger $\mathcal{K}$ dans un treillis $\mathcal{K}^{\perp,\top}$, en le complétant avec un élément maximal $\top$ et un élément minimal $\perp$, tous deux d'arité vide. Ces deux éléments vont nous permettre d'interpréter l'absence de borne pour une variable, alors majorée par $\top$ et/ou minorée par $\perp$. Ils vont également permettre d'indiquer qu'il n'existe pas de solution dans le quasi-treillis pour certaines variables, en les majorant par $\perp$ ou en les minorant par $\top$.

Nous supposerons également, comme expliqué à la fin du chapitre 3, que les systèmes de contraintes de sous-typage sont formés de types plats et qu'au moins un des membres de chacune de leur inégalités est une variable.

6.1 Système de réécriture pour le calcul de bornes

Afin d'effectuer le calcul de bornes, il sera parfois nécessaire de "factoriser" des inégalités entre variables. Pour cela, on introduit de nouvelles variables. Ces variables seront dites *introduites* et les autres variables seront dites *originelles*. On note $V^o(C)$ l'ensemble des variables originelles d'un système C et $V^i(C)$ l'ensemble de ses variables introduites. À chaque ensemble A de variables originelles correspond deux variables introduites γ_A et λ_A. Si l'ensemble A se réduit à un singleton $\{\alpha\}$, alors $\alpha = \gamma_{\{\alpha\}} = \lambda_{\{\alpha\}}$. γ_A correspond intuitivement à la borne inférieure de A et λ_A à sa borne supérieure. On ajoute également les contraintes suivantes pour chaque ensemble A de variables originelles : $\{\gamma_A \leq \alpha \mid \alpha \in A\}$ et $\{\alpha \leq \lambda_A \mid \alpha \in A\}$. Le système ainsi obtenu est appelé le *complété* de C, ou *système complété* et est noté $Compl(C)$ lorsque l'on fait référence au système dont il est issu.

Exemple 6.1 : Considérons le système $C = \alpha \leq \beta$.
On a $Compl(C) = \alpha \leq \beta,\ \gamma_{\{\alpha,\beta\}} \leq \alpha,\ \gamma_{\{\alpha,\beta\}} \leq \beta,\ \alpha \leq \lambda_{\{\alpha,\beta\}},\ \beta \leq \lambda_{\{\alpha,\beta\}}$. ◇

On introduit deux fonctions $^\uparrow$ et $^\downarrow$ qui donnent l'ensemble des variables originelles correspondant à une variable (originelle ou introduite) donnée. La fonction $^\uparrow$ est définie par $\alpha^\uparrow = \{\alpha\}$ si α est une variable originelle et ${\gamma_A}^\uparrow = A$ pour les variables introduites. Elle est indéfinie pour les variables de la forme λ_A. Symétriquement, $^\downarrow$ est définie par $\alpha^\downarrow = \{\alpha\}$ si α est une variable originelle et ${\lambda_A}^\downarrow = A$ pour les variables introduites et indéfinie pour les variables de la forme γ_A.

Le tableau 6.1 définit comment factoriser des bornes. $\sqcap_C(\alpha \leq \tau_1, \alpha \leq \tau_2)$ est un ensemble de contraintes équivalent à $\{\alpha \leq \tau_1, \alpha \leq \tau_2\}$ dans lequel on remplace τ_1 et τ_2 par leur borne inférieure. $\sqcup_C$ est symétrique. Leur définition est relative à un système complété C. C'est ici que l'on utilise les variables introduites γ_A et λ_A. On voit également, dans le calcul des contraintes C_l, l'utilisation du fait que

les types qui apparaissent dans les contraintes sont plats. On peut noter que si $\kappa \sqcap \kappa'$ n'est pas toujours définie dans $\mathcal{K}$, elle l'est toujours dans $\mathcal{K}^{\perp,\top}$. En revanche, $\sqcap_C$ n'est définie que si les conditions suivantes sont vérifiées :

- toute étiquette $l \in ar^+(\tau_1(\epsilon)) \cap ar^+(\tau_2(\epsilon))$, chacune des variables τ_1/l et τ_2/l sont soit originelles, soit de la forme γ_A.
- toute étiquette $l \in ar^-(\tau_1(\epsilon)) \cap ar^-(\tau_2(\epsilon))$, chacune des variables τ_1/l et τ_2/l sont soit originelles, soit de la forme λ_A.

La bonne définition de $\sqcup_C$ dépend de conditions symétriques. $\sqcap_C$ et $\sqcup_C$ servent à définir les règles (Glb), (Lub), (Glb Trans) et (Lub Trans).

$$\sqcap_C(\alpha \leq \tau_1, \alpha \leq \tau_2) = \alpha \leq \tau \cup \bigcup_{l \in ar(\tau(\epsilon))} C_l$$

avec $\tau(\epsilon) = \tau_1(\epsilon) \sqcap \tau_2(\epsilon)$
et pour toute étiquette $l \in ar(\tau(\epsilon))$,

si $l \in ar^+(\tau_1(\epsilon)) \cap ar^+(\tau_2(\epsilon))$,	alors $\tau/l = \gamma_{(\tau_1/l)^\uparrow \cup (\tau_2/l)^\uparrow}$ et $C_l = \{\tau/l \leq \tau_1/l, \tau/l \leq \tau_2/l\}$
si $l \in ar^-(\tau_1(\epsilon)) \cap ar^-(\tau_2(\epsilon))$,	alors $\tau/l = \lambda_{(\tau_1/l)^\downarrow \cup (\tau_2/l)^\downarrow}$ et $C_l = \{\tau_1/l \leq \tau/l, \tau_2/l \leq \tau/l\}$
si $l \in ar(\tau_1(\epsilon)) \setminus ar(\tau_2(\epsilon))$,	alors $\tau/l = \tau_1/l$ et $C_l = \emptyset$
si $l \in ar(\tau_2(\epsilon)) \setminus ar(\tau_1(\epsilon))$,	alors $\tau/l = \tau_2/l$ et $C_l = \emptyset$

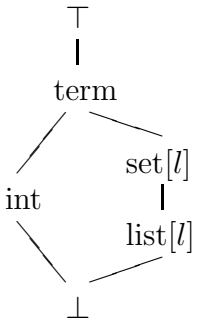
$$\sqcup_C(\tau_1 \leq \alpha, \tau_2 \leq \alpha) = \tau \leq \alpha \cup \bigcup_{l \in ar(\tau(\epsilon))} C_l$$

si $\tau(\epsilon) = \tau_1(\epsilon) \sqcup \tau_2(\epsilon)$
et pour toute étiquette $l \in ar(\tau(\epsilon))$,

si $l \in ar^+(\tau_1(\epsilon)) \cap ar^+(\tau_2(\epsilon))$,	alors $\tau/l = \lambda_{(\tau_1/l)^\downarrow \cup (\tau_2/l)^\downarrow}$ et $C_l = \{\tau_1/l \leq \tau/l, \tau_2/l \leq \tau/l\}$
si $l \in ar^-(\tau_1(\epsilon)) \cap ar^-(\tau_2(\epsilon))$,	alors $\tau/l = \gamma_{(\tau_1/l)^\uparrow \cup (\tau_2/l)^\uparrow}$ et $C_l = \{\tau/l \leq \tau_1/l, \tau/l \leq \tau_2/l\}$
si $l \in ar(\tau_1(\epsilon)) \setminus ar(\tau_2(\epsilon))$,	alors $\tau/l = \tau_1/l$ et $C_l = \emptyset$
si $l \in ar(\tau_2(\epsilon)) \setminus ar(\tau_1(\epsilon))$,	alors $\tau/l = \tau_2/l$ et $C_l = \emptyset$

TAB. 6.1 – Factorisation de bornes

Exemple 6.2 : Considérons la structure correspondant au schéma ci-dessous. On a :



- $\sqcap_C(\alpha \leq \text{list}(\beta), \alpha \leq \text{set}(\delta)) =$
  $\alpha \leq \text{list}(\gamma_{\{\beta,\delta\}}), \gamma_{\{\beta,\delta\}} \leq \beta, \gamma_{\{\beta,\delta\}} \leq \delta$
- $\sqcup_C(\text{list}(\alpha_2) \leq \alpha_1, \text{set}(\lambda_{\{\alpha_3,\alpha_4\}}) \leq \alpha_1) =$
  $\text{set}(\lambda_{\{\alpha_2,\alpha_3,\alpha_4\}}) \leq \alpha_1,\ \alpha_2 \leq \lambda_{\{\alpha_2,\alpha_3,\alpha_4\}},\ \lambda_{\{\alpha_3,\alpha_4\}} \leq \lambda_{\{\alpha_2,\alpha_3,\alpha_4\}}$
- $\sqcap_C(\alpha \leq \text{list}(\beta), \alpha \leq \text{int}) = \alpha \leq \bot$

Dans ce dernier exemple, la variable α ne pourra trouver de solution dans $\mathcal{T}(\mathcal{K})$.

◇

Les tableaux 6.2 et 6.3 présentent un système de réécriture non deterministe pour les systèmes de contraintes de sous-typage permettant d'effectuer le calcul de bornes. Ce système produit un ensemble de formes résolues permettant de décider de la satisfiabilité des contraintes.

$\alpha, \beta, \ldots$ représentent des variables et $\tau, \tau_1, \ldots$ représentent des types qui ne sont pas des variables. Le système de réécriture limité aux règles de la table 6.2 sera par la suite noté $\rightarrow$. Le système $\rightarrow$ auquel on ajoute la règle (Intro $\uparrow$) (resp. (Intro $\downarrow$)) sera noté $\rightarrow^{\uparrow}$ (resp. $\rightarrow^{\downarrow}$). Enfin, le système formé par l'ensemble des règles sera noté $\rightarrow^{\uparrow\downarrow}$. On suppose qu'une règle ne peut se déclencher que si son exécution apporte quelque chose, c'est-à-dire que si $C \rightarrow^{\uparrow\downarrow} C'$ alors $C \neq C'$. Par exemple, si $C = \alpha \leq \beta, \beta \leq \delta, \alpha \leq \delta$, alors la règle (Trans) pourrait se déclencher sur $\alpha \leq \beta, \beta \leq \delta$, mais comme $\alpha \leq \delta \in C$, le déclenchement ne se produit pas.

La règle (Trans) exprime la transitivité de la relation de sous-typage sur les variables.

La règle (Dec) décompose une contrainte à la manière de la fonction *dec* définie dans le tableau 5.1 (p. 58). La règle (Clash) correspond à l'échec de l'application de la règle (Dec).

La règle (Glb) calcule une nouvelle borne supérieure pour α, en factorisant deux de ses précédentes bornes. La règle (Lub) est symétrique. La règle (Glb Trans) met à jour une borne de α en utilisant une borne de β si $\alpha \leq \beta$. L'ensemble de ces cinq premières règles suffisent à calculer les bornes dans le treillis $\mathcal{T}(\mathcal{K}^{\bot,\top})$. Les règles suivantes sont utilisées pour le calcul des bornes dans le quasi-treillis $\mathcal{T}(\mathcal{K})$.

Les règles (Fail $\top$) et (Fail $\bot$) correspondent au cas où il n'y a pas de solution pour α dans $\mathcal{T}(\mathcal{K})$. On autorise cependant les variables introduites à prendre les valeurs $\top$ ou $\bot$. Cela correspond au cas où certains ensembles de variables originelles n'ont pas de borne inférieure (si $\gamma_A \leq \bot$) ou de borne supérieure (si $\top \leq \lambda_A$). Dans ce cas, ces variables ne doivent pas apparaître dans les bornes

(Trans) $C, \alpha \leq \beta, \beta \leq \delta \quad \rightarrow \quad C, \alpha \leq \beta, \beta \leq \delta, \alpha \leq \delta$

(Dec) $C, \tau_1 \leq \alpha, \alpha \leq \tau_2 \quad \rightarrow$
$C, \quad \tau_1 \leq \alpha, \alpha \leq \tau_2$
$\cup\{\tau_1/l \leq \tau_2/l \mid l \in ar^+(\tau_1(\epsilon)) \cap ar^+(\tau_2(\epsilon))\}$
$\cup\{\tau_2/l \leq \tau_1/l \mid l \in ar^-(\tau_1(\epsilon)) \cap ar^-(\tau_2(\epsilon))\}$
si $\tau_1(\epsilon) \leq_{\mathcal{K}} \tau_2(\epsilon)$

(Clash) $C, \tau_1 \leq \alpha, \alpha \leq \tau_2 \quad \rightarrow \quad$ faux
si $\tau_1(\epsilon) \not\leq_{\mathcal{K}} \tau_2(\epsilon)$

(Glb) $C, \alpha \leq \tau_1, \alpha \leq \tau_2 \quad \rightarrow \quad C, \sqcap_C(\alpha \leq \tau_1, \alpha \leq \tau_2)$

(Lub) $C, \tau_1 \leq \alpha, \tau_2 \leq \alpha \quad \rightarrow \quad C, \sqcup_C(\tau_1 \leq \alpha, \tau_2 \leq \alpha)$

(Glb Trans) $C, \alpha \leq \tau_1, \beta \leq \tau_2, \alpha \leq \beta \quad \rightarrow \quad C, \sqcap_C(\alpha \leq \tau_1, \alpha \leq \tau_2), \beta \leq \tau_2, \alpha \leq \beta$

(Lub Trans) $C, \tau_1 \leq \alpha, \tau_2 \leq \beta, \alpha \leq \beta \quad \rightarrow \quad C, \sqcup_C(\tau_1 \leq \beta, \tau_2 \leq \beta), \tau_1 \leq \alpha, \alpha \leq \beta$

(Fail $\bot$) $C, \alpha \leq \bot \quad \rightarrow \quad$ faux
si α est originelle

(Fail $\top$) $C, \top \leq \alpha \quad \rightarrow \quad$ faux
si α est originelle

(Down $\bot$) $C, \beta \leq \bot, \alpha \leq \tau \quad \rightarrow \quad C, \beta \leq \bot, \alpha \leq \tau'$
si $\tau/l = \beta$, $l \in \mathcal{L}^+$, $\tau'(\epsilon) = \sqcup(\downarrow_{ar(\tau(\epsilon))\setminus\{l\}} \tau(\epsilon))$ et $\forall l \in ar(\tau'(\epsilon)), \tau'/l = \tau/l$

(Down $\top$) $C, \top \leq \beta, \alpha \leq \tau \quad \rightarrow \quad C, \top \leq \beta, \alpha \leq \tau'$
si $\tau/l = \beta$, $l \in \mathcal{L}^-$, $\tau'(\epsilon) = \sqcup(\downarrow_{ar(\tau(\epsilon))\setminus\{l\}} \tau(\epsilon))$ et $\forall l \in ar(\tau'(\epsilon)), \tau'/l = \tau/l$

(Up $\top$) $C, \top \leq \beta, \tau \leq \alpha \quad \rightarrow \quad C, \top \leq \beta, \tau' \leq \alpha$
si $\tau/l = \beta$, $l \in \mathcal{L}^+$, $\tau'(\epsilon) = \sqcap(\uparrow_{ar(\tau(\epsilon))\setminus\{l\}} \tau(\epsilon))$ et $\forall l \in ar(\tau'(\epsilon)), \tau'/l = \tau/l$

(Up $\bot$) $C, \beta \leq \bot, \tau \leq \alpha \quad \rightarrow \quad C, \beta \leq \bot, \tau' \leq \alpha$
si $\tau/l = \beta$, $l \in \mathcal{L}^-$, $\tau'(\epsilon) = \sqcap(\uparrow_{ar(\tau(\epsilon))\setminus\{l\}} \tau(\epsilon))$ et $\forall l \in ar(\tau'(\epsilon)), \tau'/l = \tau/l$

TAB. 6.2 – Règles pour le calcul de bornes

(Intro ↑) $C \quad \rightarrow^{\uparrow} \quad C, \alpha \leq \tau \quad$ ou bien $\quad C \quad \rightarrow^{\uparrow} \quad C, \top \leq \alpha$
si il n'existe pas de type τ' tel que $\alpha \leq \tau' \in C$,
si $\top \leq \alpha \notin C$ et si $\tau(\epsilon) \in \overline{\mathcal{K}}$

(Intro ↓) $C \quad \rightarrow^{\downarrow} \quad C, \tau \leq \alpha \quad$ ou bien $\quad C \quad \rightarrow^{\downarrow} \quad C, \alpha \leq \bot$
si il n'existe pas de type τ' tel que $\tau' \leq \alpha \in C$,
si $\alpha \leq \bot \notin C$ et si $\tau(\epsilon) \in \underline{\mathcal{K}}$

TAB. 6.3 – Règles de pré-clôture

obtenues par l'algorithme. Leur élimination correspond au phénomène d'oubli d'argument qui est traité par les règles (Down $\bot$), (Down $\top$), (Up $\top$) et (Up $\bot$).

La règle (Down $\bot$) est appliquée lorsque l'un des arguments d'une borne de α, correspondant à l'étiquette l, est majoré par $\bot$. Pour les solutions donnant à α une valeur τ_α dans $\mathcal{T}(\mathcal{K})$, $\tau_\alpha(\epsilon)$ ne possédera pas d'argument correspondant à l'étiquette l. La règle (Down $\bot$) met ainsi à jour la borne en faisant disparaître l'argument l. Cela est fait en remplaçant le constructeur de tête $\tau(\epsilon)$ par le plus grand constructeur minorant $\tau(\epsilon)$ dans lequel l n'apparaît pas. Tout comme pour $\sqcap_C$, si ce constructeur n'est pas forcément défini dans $\mathcal{K}$, il l'est dans $\mathcal{K}^{\bot,\top}$. La règle (Down $\top$) traite le cas ou l est négatif. Les règles (Up $\top$) et (Up $\bot$) sont les règles symétriques de (Down $\bot$) et (Down $\top$).

Enfin les règles (Intro ↑) et (Intro ↓) correspondent au calcul de pré-clôture. $\overline{\mathcal{K}}$ est l'ensemble des maxima de $\mathcal{K}$ et $\underline{\mathcal{K}}$ l'ensemble de ses minima. On remarque que ces règles sont non déterministes dès que ces ensembles possèdent plusieurs éléments.

Par la suite, nous supposerons que la règle (Intro ↑) ne sera utilisée que si $\overline{\mathcal{K}}$ est fini et si tous ses éléments sont d'arité vide. Nous supposerons de plus, comme dans le théorème 5.17 que tout constructeur de type est majoré par un élément maximal. De même, nous supposerons que la règle (Intro ↓) ne sera utilisée que si $\underline{\mathcal{K}}$ est fini et si tous ses éléments sont d'arité vide, et que si tout constructeur de type est minoré par un élément minimum. Comme nous le verrons, il est possible de se passer soit de (Intro ↑), soit de (Intro ↓) lorsque l'on ne s'intéresse qu'à la satisfiabilité du système de contraintes. C'est ainsi que l'on relâche, soit sur $\overline{\mathcal{K}}$, soit sur $\underline{\mathcal{K}}$, la condition exprimée dans le théorème 5.17.

6.2 Propriétés du système de réécriture

6.2.1 Terminaison

On dit qu'un système est *terminal* si il ne peut pas se réécrire par les règles du système $\rightarrow^{\uparrow\downarrow}$. On note $D \not\rightarrow^{\uparrow\downarrow}$ le fait que D est terminal. On note $D \not\rightarrow$

(resp. $D \not\rightarrow^{\uparrow}$ et $D \not\rightarrow^{\downarrow}$) le fait que D ne peut se réécrire dans le système $\rightarrow$ (resp. $\rightarrow^{\uparrow}$ et $\rightarrow^{\downarrow}$).

Propriété 6.1 *Les systèmes de réécriture définis par* $\rightarrow^{\uparrow\downarrow}$, $\rightarrow^{\uparrow}$ *et* $\rightarrow^{\downarrow}$ *terminent.*

La première étape de la preuve de terminaison du système consiste à montrer que seul un nombre fini de constructeurs de types différents peuvent apparaître dans l'ensemble des dérivations issues d'un système donné (lemme 6.3). Pour cela, on commence par montrer le lemme suivant, qui exprime la commutativité des opérateurs $\sqcup\downarrow_L$ et celle des opérateurs $\sqcap\uparrow_L$.

Lemme 6.2 *Soit* $\kappa \in \mathcal{K}$ *un constructeur de type. Soient* $L_1, L_2 \subseteq 2^{\mathcal{L}}$ *deux ensembles d'étiquettes. Si* $\downarrow_{L_1} \sqcup (\downarrow_{L_2}\kappa) \neq \emptyset$ *alors* $\sqcup\downarrow_{L_1} \sqcup (\downarrow_{L_2}\kappa) = \sqcup\downarrow_{L_1\cap L_2}\kappa$. *Si* $\uparrow_{L_1} \sqcap (\uparrow_{L_2}\kappa) \neq \emptyset$ *alors* $\sqcap\uparrow_{L_1} \sqcap (\uparrow_{L_2}\kappa) = \sqcap\uparrow_{L_1\cap L_2}\kappa$.

Démonstration : On montre la double inégalité pour $\sqcup\downarrow_{L_1\cap L_2}\kappa$. La démonstration pour $\sqcap\uparrow_{L_1\cap L_2}\kappa$ est similaire.

On a $\sqcup(\downarrow_{L_1} \sqcup (\downarrow_{L_2}\kappa)) \leq \sqcup\downarrow_{L_2}\kappa \leq \kappa$ et, de par le lemme 4.5, $ar(\sqcup(\downarrow_{L_1} \sqcup (\downarrow_{L_2}\kappa))) \subseteq ar(\sqcup(\downarrow_{L_2}\kappa)) \cap L_1 \subseteq ar(\kappa) \cap L_1 \cap L_2$. Donc $\sqcup(\downarrow_{L_1} \sqcup (\downarrow_{L_2}\kappa)) \in \downarrow_{L_1\cap L_2}\kappa$. Comme $\downarrow_{L_1\cap L_2}\kappa \neq \emptyset$, et de par lemme 4.5, $\sqcup\downarrow_{L_1\cap L_2}\kappa$ existe, d'où $\sqcup(\downarrow_{L_1} \sqcup (\downarrow_{L_2}\kappa)) \leq \sqcup\downarrow_{L_1\cap L_2}\kappa$.

On a $\sqcup\downarrow_{L_1\cap L_2}\kappa \leq \kappa$ et $ar(\sqcup\downarrow_{L_1\cap L_2}\kappa) \subseteq L_1 \cap L_2 \cap ar(\kappa)$. Donc $\sqcup\downarrow_{L_1\cap L_2}\kappa \in \downarrow_{L_2}\kappa$. Donc $\sqcup\downarrow_{L_1\cap L_2}\kappa \leq \sqcup\downarrow_{L_2}\kappa$. Comme $ar(\sqcup\downarrow_{L_1\cap L_2}\kappa) \subseteq L_1$, $\sqcup\downarrow_{L_1\cap L_2}\kappa \in \downarrow_{L_1} \sqcup (\downarrow_{L_2}\kappa)$. Donc $\sqcup\downarrow_{L_1\cap L_2}\kappa \leq \sqcup\downarrow_{L_1} \sqcup (\downarrow_{L_2}\kappa)$. □

Lemme 6.3 *Soit* C *un système de contraintes complété. Si* $\overline{\mathcal{K}}$ *(resp.* $\underline{\mathcal{K}}$*) est fini alors le nombre de constructeurs de types apparaissant un des systèmes* C' *tels que* $C \rightarrow^{\uparrow *} C'$ *(resp.* $\rightarrow^{\downarrow *}$ *) est fini. Si* $\overline{\mathcal{K}}$ *et* $\underline{\mathcal{K}}$ *sont finis, le nombre de constructeurs de types apparaissant un des systèmes* C' *tels que* $C \rightarrow^{\uparrow\downarrow *} C'$ *est fini.*

Démonstration : Nous montrons ici la propriété pour le système de réécriture défini par $\rightarrow^{\uparrow\downarrow}$, la démonstration étant similaire pour les système définis par $\rightarrow^{\uparrow}$ et $\rightarrow^{\downarrow}$

Soit $K(C)$ l'ensemble des constructeurs de types apparaissant dans C. On défini maintenant un ensemble de constructeurs K_{base} en fonction du système considéré :

- pour $\rightarrow^{\uparrow}$, on pose $K_{base} = K(C) \cup \overline{\mathcal{K}}$
- pour $\rightarrow^{\downarrow}$, on pose $K_{base} = K(C) \cup \underline{\mathcal{K}}$
- pour $\rightarrow^{\uparrow\downarrow}$, on pose $K_{base} = K(C) \cup \overline{\mathcal{K}} \cup \underline{\mathcal{K}}$

Soit $K^{\downarrow} = \{\sqcup\downarrow_L\kappa \mid \kappa \in K_{base} \wedge L \subseteq \mathcal{L}\}$. Soit $K^{\uparrow} = \{\sqcap\uparrow_L\kappa \mid \kappa \in K_{base} \wedge L \subseteq \mathcal{L}\}$. Soient enfin $K_{\sqcap} = \{\sqcap K \mid K \neq \emptyset \subseteq K^{\downarrow}\}$ et $K_{\sqcup} = \{\sqcup K \mid K \neq \emptyset \subseteq K^{\uparrow}\}$.

$K(C)$ est fini, donc K_{base} l'est également. On peut remarquer que pour tout constructeur κ et pour tout ensemble d'étiquettes L, $\downarrow_L \kappa = \downarrow_{L \cap ar(\kappa)} \kappa$. Donc pour tout κ, $\{\sqcup\downarrow_L \kappa \mid L \subseteq \mathcal{L}\} = \{\sqcup\downarrow_L \kappa \mid L \subseteq ar(\kappa)\}$ qui est fini. Il en est de même pour $\{\sqcap\uparrow_L \kappa \mid L \subseteq \mathcal{L}\}$. Donc $K^\uparrow$ et $K^\downarrow$ sont finis. Donc $K_\sqcup$ et $K_\sqcap$ sont finis.

Soient les constructeurs $\kappa \in K_\sqcap$. On montre que $\sqcup\downarrow_L \kappa \in K_\sqcap$. Comme $\kappa \in K_\sqcap$, $\kappa = \sqcap K$ avec $K \neq \emptyset \subseteq K^\downarrow$. Soit $\kappa' \in K$. On a $\kappa \leq_\mathcal{K} \kappa'$ et $\sqcup\downarrow_L \kappa \leq \kappa$. Donc $\sqcup\downarrow_L \kappa \leq \kappa \leq \kappa'$. De plus, $ar(\sqcup\downarrow_L \kappa) \subseteq L$, donc $ar(\sqcup\downarrow_L \kappa) \cap ar(\kappa') \subseteq L$. Donc $\sqcup\downarrow_L \kappa \in \downarrow_L \kappa'$. Donc $\sqcup\downarrow_L \kappa \leq \sqcup\downarrow_L \kappa'$. On obtient donc que $\sqcup\downarrow_L \kappa \leq \sqcap\{\sqcup\downarrow_L \kappa' \mid \kappa' \in K\}$. D'un autre côté, comme pour tout $\kappa' \in K$, $\sqcup\downarrow_L \kappa' \leq \kappa'$, $\sqcap\{\sqcup\downarrow_L \kappa' \mid \kappa' \in K\} \leq \sqcap K = \kappa$. De plus $ar(\sqcap\{\sqcup\downarrow_L \kappa' \mid \kappa' \in K\}) \subseteq \bigcup_{\kappa' \in K} ar(\sqcup\downarrow_L \kappa') \subseteq L$. Donc $\sqcap\{\sqcup\downarrow_L \kappa' \mid \kappa' \in K\} \leq \sqcup\downarrow_L \kappa$. Pour tout $\kappa' \in K$, $\kappa' = \sqcup\downarrow_{L'} \kappa''$ avec $\kappa'' \in K_{base}$. Par le lemme 6.2, $\sqcup\downarrow_L \kappa' = \sqcup\downarrow_{L \cap L'} \kappa'' \in K^\downarrow$. Donc $\sqcup\downarrow_L \kappa$ est la borne inférieure d'un ensemble non vide de constructeurs tous dans $K^\downarrow$. Donc $\sqcup\downarrow_L \kappa \in K_\sqcap$. De la même manière, on montre que pour tout constructeur $\kappa \in K_\sqcup$, $\sqcap\uparrow_L \kappa \in K_\sqcup$.

Par induction sur la dérivation, on montre que si $C \rightarrow^{\uparrow *} C'$ (resp. $\rightarrow^{\downarrow *}$, ou bien $\rightarrow^{\uparrow\downarrow *}$), alors pour toute contrainte $\tau \leq \tau' \in C'$ ou $\tau \leq \alpha \in C'$ $\tau(\epsilon) \in K_\sqcup$ et pour toute contrainte $\tau' \leq \tau \in C'$ ou $\alpha \leq \tau \in C'$, $\tau(\epsilon) \in K_\sqcap$. C'est vérifié pour C, puisque $K(C) \subseteq K_{base} \subseteq K_\sqcap \cap K_\sqcup$. On vérifie ensuite aisément que l'application de chaque règle conserve l'invariant. □

Démonstration (de la propriété 6.1) : Soit C un système de contraintes de sous-typage complété.

Soit $\prec_V$ la relation définie sur les variables de C par :
– $\forall A \subseteq V^o(C), \forall \alpha \in A, \gamma_A \prec_V \alpha, \lambda_A \prec_V \alpha$
– et $\forall A \subseteq V^o(C), \forall B \subseteq A, \gamma_A \prec_V \gamma_B, \lambda_A \prec_V \lambda_B$.

On définit l'ordre $\leq_V$ sur les variables de C engendré par la clôture transitive réflexive de $\prec_V$. Comme $V(C)$ est fini, il est clair que $\leq_V$ ne possède pas de chaîne infinie descendante.

L'ordre $\leq^\downarrow$ sur les types plats qui ne sont pas des variables, dont le constructeur de tête peut apparaître dans une dérivation de C, est défini par $\tau_1 \leq^\downarrow \tau_2$ si $\tau_1(\epsilon) \leq_\mathcal{K} \tau_2(\epsilon)$ et pour toute étiquette $l \in ar(\tau_1(\epsilon)) \cap ar(\tau_2(\epsilon))$, $\tau_1/l \leq_V \tau_2/l$. On définit de manière similaire $\leq^\uparrow$: $\tau_1 \leq^\uparrow \tau_2$ si $\tau_2(\epsilon) \leq_\mathcal{K} \tau_1(\epsilon)$ et pour toute étiquette $l \in ar(\tau_1(\epsilon)) \cap ar(\tau_2(\epsilon))$, $\tau_1/l \leq_V \tau_2/l$. Comme nous considérons le système de réécriture défini par $\rightarrow^{\uparrow\downarrow}$, nous assumons[1] que $\underline{\mathcal{K}}$ et $\overline{\mathcal{K}}$ sont finis et ne contiennent que des constructeurs d'arité vide. Par le lemme 6.3, le nombre de constructeurs de tête pouvant apparaître dans une dérivation de C est donc fini. De plus $\leq_V$ ne possède pas de chaîne infinie descendante, donc $\leq^\downarrow$ et $\leq^\uparrow$ ne possèdent pas non plus de chaîne infinie descendante.

Soit $\leq_c$ l'ordre sur les contraintes engendré par : $c \leq_c c'$ si :
– soit $c = \alpha \leq \tau$, $c' = \alpha \leq \tau'$ et $\tau \leq^\downarrow \tau'$,

[1] *c.f.* p. 74

– soit $c = \tau \leq \alpha$, $c' = \tau' \leq \alpha$ et $\tau \leq^{\uparrow} \tau'$.

Soit $\prec_{\mathcal{C}}$ la relation définie sur les systèmes de contraintes complétés par $C \prec_{\mathcal{C}} C'$ si il existe des contraintes $c \in C, c' \in C'$ telles que $C \setminus \{c\} = C' \setminus \{c'\}$ et $c \leq_c c'$. L'ordre $\leq_{\mathcal{C}}$ sur les systèmes de contraintes est défini comme la clôture transitive réflexive de $\prec_{\mathcal{C}}$. Clairement, comme $\leq^{\downarrow}$ et $\leq^{\uparrow}$ n'admettent pas de chaîne infinie descendante, c'est également le cas pour $\leq_c$. On en déduit que $\leq_{\mathcal{C}}$ est un ordre bien fondé.

Considérons enfin, pour tout système C' dérivé de d'un système C complété, le quadruplet $(n_{ineq}, n_{nb}, n_c, C')$, dont les différentes composantes sont définies comme suit. n_{ineq} est le nombre de contraintes c de la forme $\alpha \leq \beta$ telles que $c \notin C'$ et $\alpha, \beta \in V(C)$. $n_{nb} = n_{nb}^{\uparrow} + n_{nb}^{\downarrow} + n_{\bot} + n_{\top}$, $n_{nb}^{\uparrow}$ étant le nombre de variables non bornées supérieurement, $n_{nb}^{\downarrow}$ le nombre de variables non bornées inférieurement, $n_{\bot}$ le nombre de variables non majorées par $\bot$ et $n_{\top}$ le nombre de variables non minorées par $\top$. n_c est le nombre de contraintes dans C'. On ordonne ces quadruplets suivant l'ordre lexicographique en utilisant $\leq_{\mathcal{C}}$ pour comparer les systèmes de contraintes.

On montre à présent que l'application de chaque règle fait diminuer les quadruplets ainsi définis :

– Les règles (Trans) et (Dec) ajoutent des inégalités entre variables et font donc diminuer n_{ineq}.
– Les règles (Glb) et (Lub) peuvent ajouter des inégalités entre variables et font donc diminuer n_{ineq}[2].
  Si aucune inégalité entre variable n'est ajoutée, alors l'application de ces règles remplace deux inégalités du système par une seule. Considérons le cas de (Glb), où $\alpha \leq \tau_1$ et $\alpha \leq \tau_2$ sont remplacées par $\alpha \leq \tau$: si $\tau = \bot$ alors n_{nb} diminue, sinon n_{nb} reste inchangé et n_c diminue. Le cas de la règle (Lub) est similaire.
– La règle (Glb Trans) peut ajouter des inégalités entre variables et dans ce cas, n_{ineq} diminue[2].
  Sinon elle remplace $\alpha \leq \tau_1$ par $\alpha \leq \tau$. Si $\tau = \bot$ alors n_{nb} diminue. Dans le cas contraire, n_{nb} et n_c restent inchangés et on peut vérifier que si $\tau_1 \neq \tau$ alors $\tau <^{\downarrow} \tau_1$, le reste du système demeurant inchangé, c'est-à-dire que la dernière composante du quadruplet diminue.
– La règle (Lub Trans) est similaire à (Glb Trans).
– La règle (Down $\bot$) ne change pas le nombre d'inégalités entre variables, donc n_{ineq} ne change pas. Si $\tau' = \bot$ alors n_{nb} diminue. Sinon, sa valeur reste inchangée. Comme on remplace une contrainte par une autre, n_c reste inchangé. En revanche, $\tau' <^{\downarrow} \tau$ ce qui signifie que la dernière composante

[2] On peut remarquer que comme le système C est complété, les variables de la forme γ_A et λ_A sont déjà présentes dans C, c'est-à-dire que (Glb), (Lub), (Glb Trans) et (Lub Trans) n'introduisent pas de nouvelles variables.

du quadruplet diminue.
- Les règles (Down $\top$), (Up $\top$) et (Up $\bot$) sont similaires à (Down $\bot$).
- Les règles (Intro $\uparrow$) et (Intro $\downarrow$) ne changent pas le nombre d'inégalités entre variables, donc n_{ineq} ne change pas. Elle font en revanche diminuer n_{nb}. □

6.2.2 Correction

Dans cette section, nous énonçons un ensemble de lemmes et corollaires visant à montrer la correction des systèmes de réécriture définis dans les tables 6.2 et 6.3. Cette notion de correction comporte principalement deux points. Le premier est que toute solution d'un système obtenu par réécriture d'un système complété C est une solution de ce système (lemme 6.4). Le deuxième point est que toute solution du système initial dans le quasi-treillis de types $\mathcal{T}(\mathcal{K})$ est conservée dans le sens où, pour chacune de ces solutions, si un système peut se réécrire, il est toujours possible de lui appliquer une règle qui conserve cette solution (lemmes 6.6 et 6.8).

Lemme 6.4 *Soit D et D' deux systèmes de contraintes et ρ' une solution de D'. Si $D \rightarrow^{\uparrow\downarrow} D'$ alors $\rho' \models D$.*

Démonstration : Par cas sur la règle utilisée pour réécrire de D en D' :

(Trans), (Dec), (Intro $\uparrow$) **ou** (Intro $\downarrow$)**.** Ces règles ne font qu'ajouter des contraintes, donc $D \subseteq D'$, donc $\rho' \models D$.

(Clash), (Fail $\top$) **ou** (Fail $\bot$)**.** Dans ce cas, D' = faux et n'a donc pas de solution.

(Glb)**.** On a $D = C, \alpha \leq \tau_1, \alpha \leq \tau_2$ et $D' = C, \sqcap_C(\alpha \leq \tau_1, \alpha \leq \tau_2)$. Soit $\alpha \leq \tau \in \sqcap_C(\alpha \leq \tau_1, \alpha \leq \tau_2)$. On a $\rho'(\alpha) \leq \rho'(\tau)$. Il suffit de montrer que pour $i \in \{1, 2\}$, $\rho'(\tau) \leq \rho'(\tau_i)$. Par construction, $\tau(\epsilon) \leq_{\mathcal{K}} \tau_i(\epsilon)$. Pour tout $l \in ar^+(\tau) \cap ar^+(\tau_i)$, $\tau/l \leq \tau_i/l \in D'$, donc $\rho'(\tau)/l \leq \rho'(\tau_i)/l$. De même, pour tout $l \in ar^-(\tau) \cap ar^-(\tau_i)$, $\tau_i/l \leq \tau/l \in D'$, donc $\rho'(\tau)/l \leq \rho'(\tau_i)/l$. Donc $\rho'(\tau) \leq \rho'(\tau_i)$. Donc $\rho' \models D$.

(Lub), (Glb Trans) **ou** (Lub Trans)**.** La démonstration est similaire à (Glb).

(Down $\bot$)**.** On a $D' = C, \alpha \leq \tau'$ et $D = C, \alpha \leq \tau$. Par définition de τ', pour toute étiquette $l \in ar(\tau')$, $\rho'(\tau')/l = \rho'(\tau)/l$. De plus $\tau'(\epsilon) \leq_{\mathcal{K}} \tau(\epsilon)$. Donc $\rho'(\tau') \leq \rho'(\tau)$. De plus $\rho'(\alpha) \leq \rho'(\tau')$. Donc $\rho' \models \alpha \leq \tau$.

(Down $\top$), (Up $\bot$) **ou** (Up $\bot$)**.** La démonstration est similaire à (Down $\bot$). □

Définition 6.5 *Soit un système de contraintes C et une valuation ρ des variables de C dans $\mathcal{T}(\mathcal{K})$. La valuation ρ^C des variables de $Compl(C)$ dans $\mathcal{K}^{\bot,\top}$ est définie de la manière suivante :*

- *pour toute variable* $\alpha \in V(C)$, $\rho^C(\alpha) = \rho(\alpha)$.
- *pour toute variable* $\gamma_A \in V^i(Compl(C))$, *si* $\rho(A)$ *est minoré alors* $\rho^C(\gamma_A) = \sqcap_{\mathcal{T}(\mathcal{K})}\rho(A)$, *sinon* $\rho^C(\gamma_A) = \bot$.
- *pour toute variable* $\lambda_A \in V^i(Compl(C))$, *si* $\rho(A)$ *est majoré alors* $\rho^C(\gamma_A) = \sqcup_{\mathcal{T}(\mathcal{K})}\rho(A)$, *sinon* $\rho^C(\gamma_A) = \top$.

Clairement, si $\rho \models C$ alors $\rho^C \models Compl(C)$.

Lemme 6.6 *Soient* C *et* D *deux systèmes tels que* $Compl(C) \rightarrow^{\uparrow\downarrow *} D$. *Soit* ρ *une solution de* C. *Si* $\rho^C \models D$ *et* $D \rightarrow D'$, *alors* $\rho^C \models D'$.

Démonstration : Comme $D \rightarrow D'$, la règle utilisée est une de celle de la table 6.2. Montrons que $\rho^C \models D'$ par cas sur la règle utilisée pour obtenir D' :

Règle (Trans). Comme $\rho^C \models D$, $\rho^C(\alpha) \leq \rho^C(\beta)$ et $\rho^C(\beta) \leq \rho^C(\delta)$. Par transitivité, on obtient $\rho^C(\alpha) \leq \rho^C(\delta)$.

Règle (Dec). Comme $\rho^C \models D$, $\rho(\tau_1) \leq \rho(\alpha) \leq \rho(\tau_2)$. Donc, par la propriété 3.6 (p. 39), pour toute étiquette $l \in ar^+(\tau_1(\epsilon)) \cap ar^+(\tau_2(\epsilon))$, $\rho(\tau_1/l) = \rho(\tau_1)/l \leq \rho(\tau_2)/l = \rho(\tau_2/l)$ et pour toute étiquette $l \in ar^-(\tau_1(\epsilon)) \cap ar^-(\tau_2(\epsilon))$, $\rho(\tau_2/l) = \rho(\tau_2)/l \leq \rho(\tau_1)/l = \rho(\tau_1/l)$. Donc $\rho^C \models D'$.

Règle (Clash). Si cette règle est applicable, D n'a clairement pas de solution.

Règle (Glb). Soit une contrainte $c \in \sqcap_C(\alpha \leq \tau_1, \alpha \leq \tau_2)$. c a la forme $\alpha \leq \tau$ ou bien la forme $\beta \leq \delta$ avec $\beta, \delta \in V(D)$. Dans ce dernier cas, soit pour une étiquette $l \in \mathcal{L}^+$, $\beta = \tau/l$, soit pour une étiquette $l \in \mathcal{L}^-$, $\delta = \tau/l$.
Supposons $l \in \mathcal{L}^+$, $\tau_1/l = \beta_1$ et $\tau_2/l = \beta_2$. On a $\beta = \gamma_{\beta_1^\uparrow \cup \beta_2^\uparrow}$. Si $\rho(\beta_1^\uparrow \cup \beta_2^\uparrow)$ est minoré alors, $\rho^C(\beta) = \sqcap_{\mathcal{T}(\mathcal{K})}\rho(\beta_1^\uparrow \cup \beta_2^\uparrow)$. De plus $\delta = \beta_1$ ou bien $\delta = \beta_2$. Comme $\rho(\beta_1^\uparrow \cup \beta_2^\uparrow)$ est minoré, $\rho(\beta_i^\uparrow)$ l'est, donc $\rho^C(\beta_i) = \sqcap_{\mathcal{T}(\mathcal{K})}\rho(\beta_i^\uparrow)$, d'où $\rho^C(\beta) \leq \rho^C(\beta_i)$. Si $\rho(\beta_1^\uparrow \cup \beta_2^\uparrow)$ n'est pas minoré, alors $\rho^C(\beta) = \bot$, donc $\rho^C(\beta) \leq \rho^C(\delta)$. De manière similaire, pour $l \in \mathcal{L}^-$, $\rho^C(\beta) \leq \rho^C(\delta)$.
Reste à montrer que $\rho^C(\alpha) \leq \rho^C(\tau)$. Comme $\rho^C \models D$, $\rho^C(\alpha) \leq \rho^C(\tau_1)$ et $\rho^C(\alpha) \leq \rho^C(\tau_2)$. Donc $\rho^C(\alpha) \leq \rho^C(\tau_1) \sqcap_{\mathcal{T}(\mathcal{K}^{\bot,\top})} \rho^C(\tau_2)$, donc $\rho^C(\alpha)(\epsilon) \leq_{\mathcal{K}^{\bot,\top}} \tau_1(\epsilon) \sqcap \tau_2(\epsilon)$. Soit une étiquette $l \in ar(\rho^C(\alpha)(\epsilon)) \cap ar(\tau(\epsilon))$. Supposons $l \in \mathcal{L}^+$. Si $l \notin ar(\tau_2(\epsilon))$ alors $\tau/l = \tau_1/l$ et donc $\rho^C(\alpha)/l \leq \rho^C(\tau)/l$. De même, si $l \notin ar(\tau_1(\epsilon))$, $\rho^C(\alpha)/l \leq \rho^C(\tau)/l$. Si $l \in ar(\tau_1(\epsilon)) \cap ar(\tau_2(\epsilon))$, alors, en posant pour $i \in \{1, 2\}$, $\tau_i/l = \beta_i$, $\tau/l = \gamma_{\beta_1^\uparrow \cup \beta_2^\uparrow}$. Comme $\rho^C(\alpha) \leq \rho^C(\tau_i)$, $\rho^C(\alpha)/l \leq \rho^C(\beta_i)$. D'où $\rho^C(\beta_1^\uparrow \cup \beta_2^\uparrow)$ est minoré. Donc $\rho^C(\gamma_{\beta_1^\uparrow \cup \beta_2^\uparrow}) = \rho^C(\beta_1) \sqcap_{\mathcal{T}(\mathcal{K})} \rho^C(\beta_2)$. D'où $\rho^C(\alpha)/l \leq \rho^C(\tau)/l$. De même, si $l \in \mathcal{L}^-$, $\rho^C(\tau)/l \leq \rho^C(\alpha)/l$. Donc $\rho^C(\alpha) \leq \rho^C(\tau)$.

Règle (Lub). La démonstration est similaire à la règle (GLB).

Règle (Glb Trans). On a $\rho^C(\alpha) \leq \rho^C(\beta) \leq \rho^C(\tau_2)$ et $\rho^C(\alpha) \leq \rho^C(\tau_1)$. De manière similaire à la règle (Glb), on démontre que $\rho^C \models \sqcap_C(\alpha \leq \tau_1, \alpha \leq \tau_2)$.

Règle (Lub Trans). La démonstration est similaire à la règle (GLB Trans).
Règle (Fail $\bot$). Il est clair que pour toute substitution ρ telle que $\rho(\alpha) \in \mathcal{T}(\mathcal{K})$, $\rho \not\models \alpha \leq \bot$.
Règle (Fail $\top$). Il est clair que pour toute substitution ρ telle que $\rho(\alpha) \in \mathcal{T}(\mathcal{K})$, $\rho \not\models \top \leq \alpha$.
Règle (Down $\bot$). Si $\rho^C(\alpha) = \bot$, on a trivialement $\rho^C(\alpha) \leq \rho^C(\tau')$. Dans le cas contraire, $\rho^C(\alpha) \in \mathcal{T}(\mathcal{K})$. De plus $\rho^C(\alpha) \leq \rho^C(\tau)$. Comme $\tau/l = \beta$ et $\beta \leq \bot \in D$, $\rho^C(\tau)/l = \bot$. Si l était dans $ar(\rho^C(\alpha)(\epsilon))$, alors on aurait forcément $\rho^C(\alpha)/l = \bot$ ce qui contredit $\rho^C(\alpha) \in \mathcal{T}(\mathcal{K})$. Donc $\rho^C(\alpha)(\epsilon) \in \downarrow_{ar(\tau(\epsilon))\setminus\{l\}}\tau(\epsilon)$, donc $\rho^C(\alpha)(\epsilon) \leq_{\mathcal{K}} \tau'(\epsilon)$. De plus, pour toute étiquette $l' \in ar(\tau'(\epsilon))$, $\rho^C(\tau')/l' = \rho^C(\tau)/l'$. Donc pour toute étiquette $l' \in ar(\tau'(\epsilon)) \cap ar(\rho^C(alpha)(\epsilon))$, $\rho^C(\alpha)/l' \leq \rho^C(\tau')/l$. D'où $\rho^C(\alpha) \leq \rho^C(\tau')$.
Règles (Down $\top$), (Up $\top$) et (Up $\bot$). La démonstration est similaire à la règle (Down $\bot$). □

Corollaire 6.7 *Soit C un système de contraintes et ρ une solution de C. Soient D et D' deux systèmes tels que $Compl(C) \rightarrow^{\uparrow\downarrow *} D \rightarrow D'$. $\rho^C \models D$ si et seulement si $\rho^C \models D'$.*

Démonstration : Par lemme 6.6, si $\rho^C \models D$ alors $\rho^C \models D'$. Par le lemme 6.4, si $\rho^C \models D'$ alors $\rho^C \models D$. □

Le lemme suivant exprime que lors de l'application des la règles (Intro $\uparrow$) et (Intro $\downarrow$) il est toujours possible de faire un choix conservant une solution donnée du système de départ.

Lemme 6.8 *Soient C et D deux systèmes tels que $Compl(C) \rightarrow^{\uparrow\downarrow *} D$. Soit ρ une solution de C et ρ^C la solution de $Compl(C)$ correspondante. Si $\rho^C \models D$ et si la règle* (Intro $\uparrow$) *(resp.* (Intro $\downarrow$)*) peut s'appliquer à D sur une variable α alors il existe D' tel que $D \rightarrow^{\uparrow} D'$ par* (Intro $\uparrow$) *(resp. $D \rightarrow^{\downarrow} D'$ par* (Intro $\downarrow$)*) sur α tel que $\rho^C \models D'$.*

Démonstration : Soit $\tau \in \mathcal{T}(\mathcal{K}^{\bot,\top})$. D'après la construction de $\mathcal{K}^{\bot,\top}$, soit $\tau(\epsilon) \leq_{\mathcal{K}} \kappa$ avec $\kappa \in \overline{\mathcal{K}}$, soit $\tau(\epsilon) = \top$. Si on prend $\tau = \rho^C(\alpha)$, on obtient que soit $\rho^C \models \alpha \leq \tau_\kappa$, avec $\tau_\kappa(\epsilon) = \kappa \in \overline{\mathcal{K}}$, soit $\rho^C(\alpha) = \top$, c'est-à-dire $\rho^C \models \top \leq \alpha$. Donc il existe bien D' tel que $D \rightarrow D'$ par (Intro $\uparrow$) sur α tel que $\rho^C \models D'$.

La démonstration est similaire pour la règle (Intro $\downarrow$). □

Corollaire 6.9 *Soient C et D deux systèmes tels que $Compl(C) \rightarrow^{\uparrow\downarrow *} D$. Soit ρ une solution de C et ρ^C la solution de $Compl(C)$ correspondant. Si la règle* (Intro $\uparrow$) *(resp.* (Intro $\downarrow$)*) peut s'appliquer à D sur une variable α alors $\rho^C \models D$ si et seulement si il existe D' tel que $D \rightarrow^{\uparrow} D'$ par* (Intro $\uparrow$) *(resp. $D \rightarrow^{\downarrow} D'$ par* (Intro $\downarrow$)*) sur α tel que $\rho^C \models D'$.*

Démonstration : Si $\rho^C \models D$, alors par le lemme 6.8, il existe D' tel que $D \rightarrow^{\uparrow} D'$ par (Intro $\uparrow$) sur α tel que $\rho^C \models D'$.
Si $\rho^C \models D'$ alors, par le lemme 6.4, $\rho^C \models D$. □

6.2.3 Satisfiabilité d'un système de contraintes

Cette section vise à montrer que les systèmes de réécriture $\rightarrow^{\uparrow}$ et $\rightarrow^{\downarrow}$ peuvent être utilisés pour tester, de façon non déterministe, la satisfiabilité d'un système de contraintes. Cela peut être exprimé par la propriété suivante :

Propriété 6.10 *Soit C un système de contraintes. C est satisfiable dans $\mathcal{T}(\mathcal{K})$ si et seulement si il existe un système $D \neq$ faux tel que $Compl(C) \rightarrow^{\uparrow *} D \not\rightarrow^{\uparrow}$ (resp. $\rightarrow^{\downarrow}$).*

Le reste de cette section est constitué de la preuve de cette propriété. Le lemme suivant exprime quelque propriétés sur la forme des types construits apparaissant dans les systèmes lors de la réécriture.

Lemme 6.11 *Soit deux systèmes C et D tels que $Compl(C) \rightarrow^{\uparrow\downarrow *} D$.*
- *Si $\alpha \leq \tau \in D$ alors pour toute étiquette $l \in ar(\tau(\epsilon))$, soit $\tau/l \in V(C)$, soit $l \in \mathcal{L}^+$ et $\tau/l = \gamma_A$ pour un certain A, soit $l \in \mathcal{L}^-$ et $\tau/l = \lambda_A$ pour un certain A. De plus $\tau \neq \top$.*
- *Si $\tau \leq \alpha \in D$ alors pour toute étiquette $l \in ar(\tau(\epsilon))$, soit $\tau/l \in V(C)$, soit $l \in \mathcal{L}^+$ et $\tau/l = \lambda_A$ pour un certain A, soit $l \in \mathcal{L}^-$ et $\tau/l = \gamma_A$ pour un certain A. De plus $\tau \neq \bot$.*

Démonstration : Par définition $Compl(C)$ vérifie l'invariant. Il suffit ensuite de vérifier que chaque règle préserve l'invariant. La définition de $\sqcap_C$ et de $\sqcup_C$ implique la préservation de l'invariant par les règles (Glb), (Lub), (Glb Trans) et (Lub Trans). On vérifie aisément que les règles (Down $\bot$), (Down $\top$), (Up $\top$) et (Up $\bot$) conservent l'invariant puisque les variables apparaissant dans les nouvelles bornes calculées étaient déjà dans les bornes précédentes avec la même étiquette. Les règles (Intro $\uparrow$) et (Intro $\downarrow$) préservent l'invariant car les bornes introduites n'ont pas d'argument. Enfin les autres règles n'introduisent pas d'inégalité entre variables et types construits, donc préservent l'invariant. □

Le lemme 6.12 indique que si un système de contraintes ne peut pas se réécrire par $\rightarrow^{\uparrow}$ (resp. $\rightarrow^{\downarrow}$), alors ou bien ce système est faux, ou bien il a une solution, qui envoie les variables du système de départ dans le quasi-treillis de types.

Lemme 6.12 *Soit C et D deux systèmes tels que $Compl(C) \rightarrow^{\uparrow *} D$ (resp $\rightarrow^{\downarrow *}$). Si D ne peut se réécrire par $\rightarrow^{\uparrow}$ (resp. par $\rightarrow^{\downarrow}$), alors soit $D =$ faux, soit C*

admet pour solution $\rho^D_{\not\rightarrow\uparrow}$ *(resp.* $\rho^D_{\not\rightarrow\downarrow}$ *), définie par le système d'équations suivant :*

$$\{\alpha = \top \mid \top \leq \alpha \in D\} \cup \{\alpha = \tau \mid \tau \notin \mathcal{V} \wedge \alpha \leq \tau \in D\}$$

(resp. $\{\alpha = \bot \mid \alpha \leq \bot \in D\} \cup \{\alpha = \tau \mid \tau \notin \mathcal{V} \wedge \tau \leq \alpha \in D\}$*),*
et telle que pour toute variable $\alpha \in V(C)$, $\rho(\alpha) \in \mathcal{T}(\mathcal{K})$.

Démonstration : Supposons que D soit différent de faux. Comme (Intro $\uparrow$) ne peut s'appliquer, pour chaque variable α de D, soit $\top \leq \alpha \in D$, soit $\exists \tau \notin \mathcal{V}, \alpha \leq \tau \in D$. De plus, comme (Glb) ne peut pas s'appliquer, il existe au plus un type τ tel que $\alpha \leq \tau \in D$. On construit alors le système d'équations suivant : $\{\alpha = \top \mid \top \leq \alpha \in D\} \cup \{\alpha = \tau \mid \tau \notin \mathcal{V} \wedge \alpha \leq \tau \in D\}$. Ce système d'équation associe à chaque variable son majorant unique dans D lorsqu'il existe, ou $\top$ le cas échéant[3]. Chaque variable apparaissant exactement une fois à gauche du signe égal dans ce système, il a une solution $\rho^D_{\not\rightarrow\uparrow}$, abrégée par la suite ρ. Pour toute variable $\alpha \in V(D)$, si $\alpha \leq \tau \in D$ avec $\tau \notin \mathcal{V}$, alors $\rho(\alpha)(\epsilon) = \tau(\epsilon)$ et pour toute étiquette $l \in ar(\tau(\epsilon))$, $\rho(\alpha)/l = \rho(\tau/l)$. Sinon, $\top \leq \alpha \in D$ et $\rho(\alpha) = \top$. On montre à présent que $\rho \models D$. Pour cela, on montre par récurrence que pour tout $n \in \mathbb{N}$, $\rho \models_n D$. Le cas $n = 0$ est trivial. Pour le cas $n + 1$, on procède par cas sur les contraintes de D :

- $\alpha \leq \beta$: Si $\top \leq \alpha \in D$, alors, comme (Lub Trans) ne peut s'appliquer, $\top \leq \beta \in D$. Si $\top \leq \beta$, alors quelque soit la valeur de $\rho(\alpha)$, on a bien $\rho(\alpha) \leq_{n+1} \rho(\beta) = \top$. Sinon, β et α sont majorés dans D, c'est-à-dire $\exists \tau_\alpha, \tau_\beta \notin \mathcal{V}$ t.q. $\alpha \leq \tau_\alpha, \beta \leq \tau_\beta \in D$. Pour montrer que $\rho(\alpha) \leq_{n+1} \rho(\beta)$, il suffit donc de montrer que $\rho(\tau_\alpha) \leq_{n+1} \rho(\tau_\beta)$. Comme (Glb Trans) ne s'applique pas, $\sqcap_C(\alpha \leq \tau_\alpha, \alpha \leq \tau_\beta) \subseteq D$. Comme (Glb) ne s'applique pas, la seule possibilité est que $\tau_\alpha(\epsilon) \sqcap \tau_\beta(\epsilon) = \tau_\alpha(\epsilon)$, ce qui implique $\tau_\alpha(\epsilon) \leq_{\mathcal{K}} \tau_\beta(\epsilon)$. Soit une étiquette $l \in ar^+(\tau_\alpha) \cap ar^+(\tau_\beta)$. Par définition de $\sqcap_C$, $\tau_\alpha/l \leq \tau_\beta/l \in \sqcap_C(\alpha \leq \tau_\alpha, \alpha \leq \tau_\beta) \subseteq D$. Donc, par récurrence, $\rho(\tau_\alpha/l) \leq_n \rho(\tau_\beta/l)$. De même si $l \in ar^-(\tau_\alpha) \cap ar^-(\tau_\beta)$, $\rho(\tau_\beta/l) \leq_n \rho(\tau_\alpha/l)$. Donc $\rho(\tau_\alpha) \leq_{n+1} \rho(\tau_\beta)$, d'où $\rho(\alpha) \leq_{n+1} \rho(\beta)$.
- $\alpha \leq \tau$, $\tau \notin \mathcal{V}$: De par le lemme 6.11, $\tau \neq \top$. Donc $\top \leq \alpha \notin D$ (sinon (Clash) s'appliquerait). Donc $\rho(\alpha) = \rho(\tau)$ ce qui implique $\rho(\alpha) \leq_{n+1} \rho(\tau)$.
- $\tau \leq \alpha$, $\tau \notin \mathcal{V}$: Si $\tau = \top$ alors $\rho(\alpha) = \top$ d'où $\tau \leq_{n+1} \rho(\alpha)$. Sinon, on a $\alpha \leq \tau'$ pour un certain $\tau' \notin \mathcal{V}$. Comme (Clash) ne peut s'appliquer, on a $\tau(\epsilon) \leq_{\mathcal{K}} \tau'(\epsilon)$. De plus, comme (Dec) ne peut s'appliquer, pour toute étiquette $l \in ar^+(\tau) \cap ar^+(\tau')$, $\tau/l \leq \tau'/l \in D$, ce qui implique par récurrence que $\rho(\tau)/l \leq_n \rho(\tau')/l$. De même, pour toute étiquette $l \in ar^-(\tau) \cap ar^-(\tau')$, $\rho(\tau')/l \leq_n \rho(\tau)/l$. Donc $\rho(\tau) \leq_{n+1} \rho(\tau')$.

Donc pour tout $n \in \mathbb{N}$, $\rho \models_n D$, donc $\rho \models D$. En itérant le lemme 6.4 sur la réécriture $Compl(C) \rightarrow^{\uparrow *} D$, on obtient que $\rho \models Compl(C)$, et donc $\rho \models C$.

[3]Dans ce cas, on a $\top \leq \alpha \in D$, donc toutes les solutions ρ de D associent $\top$ à α.

Reste à montrer que pour toute variable originelle $\alpha \in V(C)$, $\rho(\alpha) \in \mathcal{T}(\mathcal{K})$.

Pour cela, on montre d'abord que si α est une variable originelle, $\rho(\alpha) \notin \{\top, \bot\}$: comme (Fail $\top$) ne peut s'appliquer, $\top \leq \alpha \notin D$, et donc il existe $\tau \notin \mathcal{V}$, $\alpha \leq \tau \in D$. De part le lemme 6.11, $\tau \neq \top$. Donc $\rho(\alpha) = \rho(\tau) \neq \top$. De plus, si τ était égal à $\bot$, la règle (Fail $\bot$) pourrait s'appliquer, donc $\rho(\alpha) = \rho(\tau) \neq \bot$.

On montre ensuite que pour toute inégalité $\alpha \leq \tau \in D$, pour toute variable β telle que $\beta = \tau/l$ avec $l \in ar(\tau(\epsilon))$, $\rho(\beta) \neq \top$ et $\rho(\beta) \neq \bot$. On distingue des cas selon le type de variable qu'est β :

- β est une variable originelle : d'après ce qui précède, $\rho(\beta) \neq \top$ et $\rho(\beta) \neq \bot$.
- $\beta = \gamma_A$: Par définition de $Compl(C)$, et comme aucune règle n'enlève d'inégalité entre variables, on a $\beta \leq \alpha' \in D$ pour tout $\alpha' \in A$. Si $\top \leq \beta \in D$, alors, par (Lub Trans) on a $\top \leq \alpha'$. Or α' est originelle, ce qui signifie que (Fail $\top$) peut s'appliquer, ce qui contredit l'hypothèse. Donc $\top \leq \beta \notin D$. Donc $\beta \leq \tau' \in D$ pour un certain $\tau' \notin \mathcal{V}$, et donc $\rho(\beta) = \rho(\tau')$. Par le lemme 6.11, $\tau' \neq \top$, donc $\rho(\beta) \neq \top$. Par le lemme 6.11, l'étiquette l est positive. Comme la règle (Down $\bot$) ne peut s'appliquer, on obtient que $\tau' \neq \bot$, donc $\rho(\beta) \neq \bot$.
- $\beta = \lambda_A$: Par le lemme 6.11, l'étiquette l est négative, et, comme la règle (Down $\top$) ne peut s'appliquer, $\top \leq \beta \notin D$. Donc $\beta \leq \tau' \in D$ pour un certain $\tau' \notin \mathcal{V}$ et $\rho(\beta) = \rho(\tau')$. Par définition de $Compl(C)$, et comme aucune règle n'enlève d'inégalité entre variables, on a $\alpha' \leq \beta \in D$ pour tout $\alpha' \in A$. Si $\beta \leq \bot \in D$, alors, par (Glb Trans) on a $\alpha' \leq \bot$. Or α' est originelle, ce qui signifie que (Fail $\bot$) peut s'appliquer, ce qui contredit l'hypothèse. Donc $\alpha \leq \bot \notin D$, c'est-à-dire $\tau' \neq \bot$. De plus, par le lemme 6.11, $\tau' \neq \top$. Donc $\rho(\beta) \neq \bot$ et $\rho(\beta) \neq \top$.

Par récurrence sur w, on montre à présent que si $\rho(\alpha)(\epsilon) \notin \{\top, \bot\}$, alors, pour tout $w \in dom(\rho(\alpha))$, $\rho(\alpha)(w) \notin \{\top, \bot\}$. Le cas $w = \epsilon$ est trivial. Considérons $w = l.w'$. Comme $\rho(\alpha) \notin \{\top, \bot\}$, il existe une inégalité $\alpha \leq \tau \in D$ pour un certain $\tau \notin \mathcal{V}$. Donc $\rho(\alpha)/l = \rho(\tau)/l = \rho(\tau/l)$. Or, par ce qui précède, $\rho(\tau/l) \notin \{\top, \bot\}$. Donc, par récurrence, $\rho(\alpha)(l.w') = \rho(\tau)(l.w') = \rho(\tau/l)(w') \notin \{\top, \bot\}$. Donc si $\rho(\alpha)(\epsilon) \notin \{\top, \bot\}$, alors $\rho(\alpha) \in \mathcal{T}(\mathcal{K})$.

D'après ce qui précède, si α est originelle, alors $\rho(\alpha)(\epsilon) \notin \{\top, \bot\}$, ce qui implique que $\rho(\alpha) \in \mathcal{T}(\mathcal{K})$.

La démonstration est similaire pour $\rightarrow^{\downarrow}$. □

Démonstration (de la propriété 6.10) : Si C a une solution ρ, alors $\rho^C \models Compl(C)$. En utilisant les lemmes 6.6 et 6.8, on peut construire une suite de longueur maximale $D_0 \rightarrow^{\uparrow} D_1 \rightarrow^{\uparrow} \ldots$, telle que $D_i \rightarrow^{\uparrow} D_{i+1}$ et pour tout i, $\rho^C \models D_i$. D'après la propriété 6.1, cette suite est finie. Soit D son dernier élément. On a $D \not\rightarrow^{\uparrow}$, et, comme $\rho^C \models D$, $D \neq$ faux.

Si $Compl(C) \rightarrow^{\uparrow *} D \not\rightarrow^{\uparrow}$, avec $D \neq$ faux, alors, d'après le lemme 6.12 il existe une solution ρ de D telle que ρ est une solution de C dans $\mathcal{T}(\mathcal{K})$.

La démonstration est similaire pour $\rightarrow^{\downarrow}$. □

6.2.4 Solutions explicites optimales

Les solutions décrites dans le lemme 6.12 montrent que les bornes des variables dans les systèmes terminaux sont optimales, dans le sens où, si on considère deux systèmes C et D tels que $Compl(C) \rightarrow^{\uparrow *} D \not\rightarrow^{\uparrow}$, et si $\alpha \leq \tau \in D$ alors le système D', obtenu en remplaçant τ par τ' tel que $\tau'(\epsilon) <_{\mathcal{K}} \tau(\epsilon)$, a forcément strictement moins de solutions que D. En d'autres termes, il n'est pas possible d'améliorer les bornes de D.

Cas covariant et solutions maximales

Dans le cas covariant, c'est-à-dire lorsqu'aucun constructeur n'a d'étiquette négative ($\mathcal{L}^- = \emptyset$), on obtient alors une solution maximale ou minimale du système D :

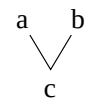
Théorème 6.13 *Si l'ensemble $\mathcal{L}^-$ des étiquettes négatives est vide alors pour tout système $D \not\rightarrow^{\uparrow}$ (resp. $D \not\rightarrow^{\downarrow}$) différent de* faux*, la solution $\rho^D_{\not\rightarrow^{\uparrow}}$ (resp. $\rho^D_{\not\rightarrow^{\downarrow}}$) définie dans le lemme 6.12 est maximale (resp. minimale) parmi toutes les solutions de D.*

Démonstration : Supposons $D \not\rightarrow^{\uparrow}$. Soit ρ une solution de D. Il suffit de montrer que que pour tout $n \in \mathbb{N}$, pour toute variable $\alpha \in V(D)$, $\rho(\alpha) \leq_n \rho^D_{\not\rightarrow^{\uparrow}}(\alpha)$. C'est trivialement vrai pour $n = 0$. Considérons le cas $n + 1$. Comme expliqué dans la preuve du lemme 6.12, soit $\top \leq \alpha \in D$, soit $\alpha \leq \tau \in D$. Dans le premier cas on a $\rho(\alpha) = \top = \rho^D_{\not\rightarrow^{\uparrow}}(\alpha)$. Dans le deuxième cas, comme ρ est solution de D, $\rho(\alpha) \leq \rho(\tau)$. Par récurrence, comme $\mathcal{L}^- = \emptyset$, pour toute étiquette $l \in ar(\tau(\epsilon))$, $\rho(\tau/l) \leq_n \rho^D_{\not\rightarrow^{\uparrow}}(\tau/l)$. Donc $\rho(\tau) \leq_{n+1} \rho^D_{\not\rightarrow^{\uparrow}}(\tau)$, et par transitivité, $\rho(\alpha) \leq_{n+1} \rho^D_{\not\rightarrow^{\uparrow}}(\tau)$. □

Ceci est particulièrement intéressant dans le cadre du système de types pour CLP($\mathcal{X}$) de la deuxième partie, puisque l'hypothèse de covariance est respectée dans ce cas.

En collectant pour chaque état terminal atteint la solution maximale qui lui correspond, on peut obtenir un ensemble S de solutions du système de départ C. L'ensemble des solutions maximales de C est un sous-ensemble de S, mais tous les éléments de S ne sont cependant pas maximaux, comme le montre l'exemple suivant :

Exemple 6.3 : Considérons la structure de constructeurs de types décrite par la figure ci-contre. Considérons le système de contraintes $C = \{\alpha \leq \beta, \beta \leq b\}$. Il y a trois possibilités d'appliquer (Intro ↑) sur α :



1. $\top \leq \alpha$. Comme α est originelle, ce système se réécrit en faux.
2. $\alpha \leq a$. Ce système se réécrit par (Glb Trans) en $\{\alpha \leq c, \beta \leq b, \alpha \leq \beta\}$, qui a pour solution maximale $\alpha \mapsto c, \beta \mapsto b$.
3. $\alpha \leq b$. qui a pour solution maximale $\alpha \mapsto b, \beta \mapsto b$.

Il est clair que la solution maximale du deuxième système n'est pas une solution maximale de C, puisqu'elle est plus petite que la solution du troisième système. ◇

6.2.5 Indépendance vis-à-vis du choix des règles

Le système de réécriture $\rightarrow$ n'est pas confluent, comme le montre l'exemple suivant :

Exemple 6.4 : Considérons une signature contenant un constructeur de type list avec un argument. Soit $C = \alpha \leq \text{list}(\beta_1), \alpha \leq \text{list}(\beta_2), \alpha \leq \text{list}(\beta_3)$. Supposons $Compl(C) = C, E$. On a :

$$\begin{array}{lll} C, E & \rightarrow & \alpha \leq \text{list}(\gamma_{\{\beta_1,\beta_2\}}), \alpha \leq \text{list}(\beta_3), E \\ & \rightarrow & \alpha \leq \text{list}(\gamma_{\{\beta_1,\beta_2,\beta_3\}}), \gamma_{\{\beta_1,\beta_2,\beta_3\}} \leq \gamma_{\{\beta_1,\beta_2\}}, E \end{array}$$

et :

$$\begin{array}{lll} C, E & \rightarrow & \alpha \leq \text{list}(\gamma_{\{\beta_2,\beta_3\}}), \alpha \leq \text{list}(\beta_1), E \\ & \rightarrow & \alpha \leq \text{list}(\gamma_{\{\beta_1,\beta_2,\beta_3\}}), \gamma_{\{\beta_1,\beta_2,\beta_3\}} \leq \gamma_{\{\beta_2,\beta_3\}}, E \end{array}$$

En utilisant toujours la règle (Glb). Même si les deux systèmes finaux ont les mêmes solutions, il n'est pas possible d'obtenir un même système en appliquant des règles de $\rightarrow$ sur chacun de ces systèmes. ◇

L'absence de cette propriété de confluence pourrait s'avérer problématique pour un algorithme de test de satisfiabilité basé sur $\rightarrow^{\uparrow\downarrow}$, en particulier si celui-ci était obligé d'énumérer toutes les dérivations possibles. Cependant, à défaut de confluence, le système de réécriture possède une propriété proche, qui peut se résumer ainsi : on peut choisir la règle à appliquer lors du prochain pas de réécriture sans perdre de solution. En fait, de part le corollaire 6.7, la perte de solution ne peut résulter que de l'application des règles (Intro ↑) et (Intro ↓), ce qui sera exprimé par le lemme 6.15.

Un pas lors d'une dérivation $\rightarrow^{\uparrow\downarrow}$ peut être caractérisé par la règle utilisée, les contraintes sur lesquelles elle agit et le résultat de son application (c'est à

dire quelles contraintes ont été générées). La définition suivante introduit la notion d'*occurrence d'application d'une règle* qui regroupe les deux premières caractéristiques. Cette notion est similaire à la notion d'atome sélectionné dans le cadre de la résolution CSLD.

Définition 6.14 *Soit un système de contraintes C. Une* occurrence d'application de règle *dans C est un couple (R, S) où R est une règle du système $\rightarrow^{\uparrow\downarrow}$ et S est soit un sous-ensemble minimal de C sur lequel R peut s'appliquer, si R est une règle du tableau 6.2, soit la variable à laquelle est ajoutée une borne si R est une règle du tableau 6.3.*

Exemple 6.5 : Soit le système $C = \{\alpha \leq \tau, \alpha \leq \beta, \beta \leq \tau', \beta \leq \delta\}$.

$((\text{Lub Trans}), \{\alpha \leq \tau, \alpha \leq \beta, \beta \leq \tau'\})$ et $((\text{Trans}), \{\alpha \leq \beta, \beta \leq \delta\})$ sont des occurrences d'application de règle. En revanche, $((\text{Trans}), \{\alpha \leq \beta, \beta \leq \delta, \alpha \leq \tau\})$ n'est pas une, car le sous-ensemble de contraintes n'est pas minimal. De même, $((\text{Lub Trans}), \{\alpha \leq \tau, \alpha \leq \beta\})$ n'en est pas une non plus car la règle ne peut s'appliquer sur le sous-ensemble de contraintes. ◇

On dit que $C \rightarrow^{\uparrow\downarrow} D$ par (R, S) si $C \rightarrow^{\uparrow\downarrow} D$ en appliquant R sur S.

Lemme 6.15 *Soit deux systèmes de contraintes C et D tels que $Compl(C) \rightarrow^{\uparrow\downarrow *} D$. Soit (R_1, S_1) et $(R_2, S_2) \neq (R_1, S_1)$ deux occurrences de règles de D. Pour tout D_1 tel que $D \rightarrow^{\uparrow\downarrow} D_1$ par (R_1, S_1) et pour toute valuation ρ de C telle que $\rho^C \models D_1$, il existe D_2 tel que $D \rightarrow^{\uparrow\downarrow} D_2$ par (R_2, S_2) et $\rho^C \models D_2$.*

Démonstration : De par le lemme 6.4, $\rho^C \models D$.

Si R_2 est une règle du système $\rightarrow$, alors, par le lemme 6.6, $\rho^C \models D_2$.

Sinon, R_2 est soit (Intro $\uparrow$), soit (Intro $\downarrow$), et, par le lemme 6.8, il existe D_2 tel que $D \rightarrow^{\uparrow\downarrow} D_2$ par (R_2, S_2) et $\rho^C \models D_2$. □

Ce lemme exprime que c'est le choix effectué lors de l'application des règles (Intro $\uparrow$) et (Intro $\downarrow$) qui restreint l'ensemble des solutions et non le choix de l'occurrence de règle choisie pour effectuer le pas suivant dans la dérivation. On peut donc choisir arbitrairement l'occurrence de règle pour effectuer le pas suivant dans la dérivation.

6.3 Algorithme

Nous présentons ici un algorithme pour tester la satisfiabilité utilisant le système de règles $\rightarrow^{\uparrow}$. Il est également possible d'utiliser le système de règles $\rightarrow^{\downarrow}$ avec un algorithme symétrique.

Si on suppose que l'ensemble des maxima $\overline{\mathcal{K}}$ de $\mathcal{K}$ est un ensemble fini de constantes, le nombre de réécritures possibles d'un système $Compl(C)$ est fini :

à chaque étape, il n'existe qu'un nombre fini de possibilités d'appliquer une règle de $\rightarrow^{\uparrow}$, et, de par la propriété 6.1, il n'existe pas de réécriture infinie. On peut donc savoir s'il existe un système terminal issu de $Compl(C)$ et différent de faux, ce qui donne un algorithme pour tester la satisfiabilité de C.

L'algorithme donné tel quel procède en énumérant tous les systèmes terminaux possibles issus de $Compl(C)$. Il est cependant possible de ne considérer qu'un nombre restreint de tels systèmes : d'après le lemme 6.15, il est possible de choisir l'occurrence d'application de règle correspondant au pas suivant, et ceci de manière arbitraire, sans perdre de solution. Autrement dit, il n'est pas nécessaire d'énumérer à chaque étape les différents choix d'occurrence d'application de règle, mais simplement les choix effectués par la règle (Intro $\uparrow$) (qui est la seule règle non déterministe du système de règles $\rightarrow^{\uparrow}$). Ceci permet de limiter fortement le nombre de réécritures à effectuer. Cela permet également d'implanter une stratégie particulière en choisissant l'occurrence d'application de règle à appliquer au pas de réécriture suivant.

Incrémentalité partielle

Afin de pouvoir localiser les erreurs le plus tôt possible lors de la vérification et de l'inférence des types, il est intéressant de pouvoir tester la satisfiabilité des systèmes de contraintes de sous-typage de manière incrémentale, c'est-à-dire, de tester la satisfiabilité d'un système C, puis d'ajouter un ensemble de contraintes C' à ce système et de tester la satisfiabilité du nouveau système $C \cup C'$ sans recommencer tout le travail accompli pour C.

Le lemme 6.15 indique que l'on peut choisir l'occurrence de règle à utiliser et le lemme 6.6 indique que les règles de la table 6.2 conservent toutes les solutions. On peut donc appliquer les occurrences des règles de la table 6.2 n'apparaissant que dans C, obtenant ainsi C_1. On teste ensuite la satisfiabilité de C_1 en utilisant l'algorithme décrit précédemment. On peut ensuite tester la satisfiabilité de $C \cup C'$ en testant la satisfiabilité de $C_1 \cup C'$, puisque C_1 est équivalent à C. On aura ainsi évité de recommencer les opérations correspondant au calcul de C_1. En itérant ce schéma, on obtient alors un algorithme partiellement incrémental.

6.4 Implantation en CHR

Nous avons implanté, dans TCLP et en utilisant le langage CHR, un solveur de contraintes de sous-typage utilisant ces règles. Ce solveur met à disposition de l'utilisateur la contrainte `:<`/2 et les prédicats `fresh`/1 et `tclp__enumerate_bounds`/0. Le prédicat `fresh`/1 sert à associer à une variable une structure de donnée, en faisant ainsi une variable de type. La contrainte `:<`/2 (représentant $\leq$) est utilisable sur des variables de type et des termes représentant

des types. Enfin, le prédicat `tclp__enumerate_bounds`/0 déclenche l'utilisation de la règle (Intro ↑) sur les variables qui ne sont pas encore bornées supérieurement.

6.4.1 Structures de données

Au niveau de l'implantation, quatre données sont associées à chaque variable de type `X` au travers de la contrainte CHR `tclp__parameter`/5. Ces données sont d'une part un majorant et un minorant de `X` qui ne sont pas des variables, d'autre part deux ensembles (codés sous forme de listes) de variables reliées à `X` : $\{\alpha \mid \alpha \leq \texttt{X} \in C\}$ et $\{\alpha \mid \texttt{X} \leq \alpha \in C\}$. L'absence de borne supérieure est codée en donnant $\top$ (`top`) comme borne à la variable. De même, l'absence de borne inférieure est codée avec $\bot$ (`bot`).

Exemple 6.6 : Considérons le système de contraintes suivant : $C = \alpha \leq \beta, \beta \leq$ int. Si `X` représente α et `Y` représente β, C est représenté par :

```
tclp__parameter(X,top,[Y],[],bot),
tclp__parameter(Y,int,[],[X],bot)
```
◇

On peut remarquer que cette représentation n'autorise qu'un seul minorant et un seul majorant construits pour chaque variable. Cependant, les règles (Glb) et (Lub) indiquent qu'il est possible d'avoir plusieurs tels minorants/majorants. En appliquant la stratégie consistant à appliquer immédiatement les règles (Glb) et (Lub), on peut cependant se contenter d'un seul majorant et d'un seul minorant.

Les règles du système $\rightarrow^{\uparrow\downarrow}$ ne peuvent s'appliquer que sur des systèmes de contraintes complétés. Les variables introduites des systèmes complétés sont créées de manière paresseuse, ce qui est nécessaire pour des raisons de performances : en pratique le nombre de variable introduites reste très faible par rapport au nombre maximal de telles variables, à savoir 2^n, si n est le nombre de variables originelles. La question de connaître le nombre de variables introduites utiles au calcul de bornes reste d'ailleurs un problème ouvert, qui se posait déjà pour l'algorithme incrémental de calcul de bornes dans les treillis de Pottier [56]. Ces variables sont représentées par des variables avec la contrainte CHR `tclp__parameter`/5 et une contrainte `tclp__original_up`/2 pour les variables γ_A (`tclp__original_down`/2 pour les variables λ_A), qui associe à chaque variable introduite l'ensemble des variables qui lui correspond :

Exemple 6.7 : En reprenant l'exemple 6.6, si `Z` représente $\gamma_{\alpha,\beta}$, elle se verra attribuer les contraintes suivantes :

```
tclp__parameter(Z,top,[X,Y],[],bot),
tclp__original_up(Z,[X,Y]).
```
◇

6.4.2 Stratégie utilisée

L'ajout de contraintes de sous-typage se fait à l'aide de `:<`/2. Son comportement sera différent selon que ses arguments seront des variables ou non. Soit C le système de contraintes avant l'application des règles déclenchées par `:<`/2.

- Si les deux arguments sont des types construits, on transforme ces deux types en types plats, puis on applique une règle similaire à (Dec) avec pour seule différence que sa prémisse est de la forme $\tau_1 \leq \tau_2$ et non pas $\tau_1 \leq \alpha, \alpha \leq \tau_2$.
- Si son premier argument est une variable α et son deuxième argument un type τ construit ($\alpha \leq \tau$), on commence par transformer ce dernier en type plat τ'. Le solveur applique ensuite la règle (Glb Trans) sur toutes les variables β telles que $\beta \leq \alpha \in C$ avec comme deuxième prémisse $\alpha \leq \tau'$. Si une de ces variables n'était pas bornée supérieurement, la contrainte $\beta \leq \tau'$ est simplement ajoutée, ce qui correspond à l'application de (Intro $\uparrow$) suivie de (Glb Trans), avec pour (Intro $\uparrow$) un majorant choisi parmi ceux de τ'. Enfin la règle (Glb) est appliquée sur α avec τ' et son ancien majorant.
- Le cas où le premier argument de `:<`/2 est construit et son deuxième argument est une variable ($\tau \leq \alpha$) est traité symétriquement.
- Si les deux arguments de `:<`/2 sont des variables ($\alpha \leq \beta$), alors l'ensemble de contraintes $\{\delta \leq \delta' \mid (\delta \leq \alpha \in C \vee \delta = \alpha) \wedge (\beta \leq \delta' \in C \vee \delta' = \beta)\}$ est ajouté à C en utilisant la règle (Trans). Si $\tau \leq \alpha \in C$ alors on applique en plus les mêmes règles que lors du cas $\tau \leq \beta$ pour mettre à jour les bornes inférieures de β et de toutes les variables qui lui sont supérieures. De même, si $\beta \leq \tau' \in C$ alors on applique en plus les mêmes règles que lors du cas $\alpha \leq \tau'$ pour mettre à jour les bornes supérieures de α et de toutes les variables qui lui sont inférieures.

6.4.3 Mise à jour des contraintes

Cette stratégie est implantée au travers de contraintes CHR utilisées pour mettre à jour la contrainte `tclp__parameter`/5 pour chaque variable.

Exemple 6.8 : La règle CHR pour mettre à jour l'ensemble des variables au dessus d'une variable donnée est :

```
tclp__update_hiset(X, Hiset) ,
tclp__parameter(X, HiboundX, HisetX, LosetX, LoboundX)
     <=> ord_union(Hiset, SHisetX, NewHiset),
         tclp__parameter(X, HiboundX, NewHiset, LosetX, LoboundX).
```

Exemple 6.9 : La règle CHR pour mettre à jour l'ensemble des variables au dessus d'une variable donnée est :

```
tclp__update_hibound(X,Hibound) ,
tclp__parameter(X, HiboundX, HisetX, LosetX, LoboundX)
     <=> tclp__glb(HiboundX, Hibound, NewHibound),
         tclp__parameter(X, NewHibound, HisetX, LosetX, LoboundX).
```

La mise à jour de la borne supérieure d'une variable (correspondant à l'application de la factorisation de bornes présentée dans le tableau 6.1) présentée ci-dessus utilise le prédicat `tclp__glb`/3 qui dit que son troisième argument est la borne inférieure (dans le quasi-treillis) de ses deux premiers arguments. Ce prédicat fait appel à la contrainte `tclp__original_up`/2 pour trouver les variables à faire apparaître dans le troisième argument de `tclp__glb`/3. En fait, étant donné un ensemble de variables A, pour trouver γ_A, on procède en énumérant les contraintes `tclp__original_up`/2 jusqu'à trouver celle dont le deuxième argument correspond à l'ensemble de variables A. Son premier argument est alors γ_A. Si aucune contrainte `tclp__original_up`/2 ne correspond, une nouvelle variable est introduite pour représenter γ_A et la contrainte correspondante est ajoutée. La mise à jour de la borne inférieure se fait de manière symétrique en utilisant `tclp__original_down`/2 et `tclp__lub`/3.

6.4.4 Unification de variables de types

Étant donnée la complexité des algorithmes de résolution de contraintes de sous-typage non structurel (cubique dans le meilleur des cas), la possibilité de simplifier les contraintes a une grande importance [29, 61]. Une des simplifications possibles consiste à gérer l'unification des variables de type. Dans les algorithmes proposés par Smith et Trifonov [70] et parallèlement par Pottier [56], l'unification des variables de type se fait au travers des opérations de minimisation et de dépoussiérage des systèmes de contraintes, en particulier en détectant des cycles dans les inégalités. Ainsi, pour poser la contrainte $\alpha = \beta$, on pose les contraintes $\alpha \leq \beta$ et $\beta \leq \alpha$, les deux variables étant unifiées plus tard, au moment de la minimisation. Cette solution peut s'avérer coûteuse dans le sens où cette minimisation a lieu après la collection des inégalités de sous-typage. La présentation de l'algorithme sous forme de règles et la possibilité naturelle d'unifier deux variables en CHR a permis d'implanter directement la contrainte d'égalité dans le solveur, ce qui permet de simplifier le système le plus tôt possible.

La règle d'unification est simplement déclenchée par l'unification des variables Prolog/CHR représentant les deux variables de type à unifier. La variable résultant de cette unification possède alors deux contraintes `tclp__parameter`/2. Ces deux contraintes sont alors éliminées et une nouvelle contrainte `tclp__parameter`/2 est ajoutée, combinant les bornes et les ensembles de variables majorant et minorant α et β. On applique ensuite les mêmes règles que

celles que l'on aurait appliqué avec la double inégalité. Le gain obtenu par ce traitement est que l'unification est gérée le plus tôt possible, ce qui allège le traitement des contraintes `:<`/2 ajoutées par la suite.

Les conséquences de l'unification de deux variables de types vont cependant au delà des conséquenses directes de la double inégalité. En effet, l'identification de α et β implique l'identification de certaines variables de la forme γ_A ou λ_A. Plus précisément, pour tout A tel que $\alpha, \beta \notin A$, on a les identifications suivantes : $\{\gamma_{A\uplus\{\alpha,\beta\}} = \gamma_{A\uplus\{\alpha\}} = \gamma_{A\uplus\{\beta\}}\}$ et $\{\lambda_{A\uplus\{\alpha,\beta\}} = \lambda_{A\uplus\{\alpha\}} = \lambda_{A\uplus\{\beta\}}\}$. Plutôt que de générer systématiquement les contraintes d'égalité entre variables introduites correspondant à chaque unification de variables originelles, celles-ci sont générées lors de la recherche des variables introduites dans les prédicats `tclp__glb`/3 et `tclp__lub`/3. Étant donné un ensemble de variables A, on trouvera plusieurs contraintes `tclp__original_up`/2 liant A à différentes variables introduites. On unifie alors ces variables avant de construire la borne inférieure dans `tclp__glb`/3 (resp. la borne supérieure dans `tclp__lub`/3).

6.4.5 Performances

Comparaison avec l'implantation OCaml

La possibilité de gérer l'unification de variables de types α et β directement, et non simplement par l'intermédiaire de la double inégalité $\alpha \leq \beta, \beta \leq \alpha$, éventuellement simplifiée plus tard, permet un grand gain de performances, comme le montre le tableau 6.4. L'implantation OCaml utilise la bibliothèque de contraintes de sous-typage Wallace [58], qui ne gère pas directement l'unification des variables de type, contrairement à l'implantation CHR présentée ci-dessus. Le tableau 6.4 compare les temps de vérification des types (avec inférence du type des variables), décrite au chapitre 8 et d'inférence heuristique de types, décrite au chapitre 9, pour les prédicats entre les implantations OCaml et Prolog/CHR de TCLP. Les tests ont été effectués sur 12 bibliothèques[4] de SICStus Prolog, utilisant un treillis comme structure de type.

Alors que l'on pourrait s'attendre à une grande baisse de performance en passant de l'implantation OCaml à l'implantation CHR, les temps CHR sont comparables aux temps OCaml pour la vérification de types (en moyenne, le rapport des temps OCaml / CHR est de 1.05). L'implantation CHR est même bien plus rapide pour l'inférence de type, puisque la moyenne du rapport des temps OCaml / CHR est de 7.39 et atteint même 71.79 dans le cas de la bibliothèque `terms.pl`. La raison est que, en l'absence d'unification des variables de type, l'algorithme d'inférence des variables du programme associe une variable de type à

[4]Par la suite, les test sont réalisés sur 5 bibliothèques supplémentaires, qui n'ont pas pu être typées par la version OCaml, cette dernière ne gérant pas la surcharge.

Fichier	Taille	Vérification		Inférence	
		OCaml	CHR	OCaml	CHR
`assoc.pl`	209 l	5.3 s	6.0 s	40.1 s	13.6 s
`atts.pl`	216 l	7.4 s	5.5 s	77.5 s	12.4 s
`bdb.pl`	487 l	23.6 s	20.2 s	41.1 s	17.4 s
`charsio.pl`	97 l	1.3 s	1.0 s	2.4 s	1.3 s
`clpb.pl`	584 l	24.3 s	22.7 s	1827.3 s	224.8 s
`fastrw.pl`	54 l	0.4 s	0.5 s	0.7 s	0.7 s
`heaps.pl`	137 l	3.5 s	4.2 s	43.3 s	17.4 s
`lists.pl`	186 l	3.5 s	3.8 s	16.2 s	6.6 s
`ordsets.pl`	208 l	4.1 s	5.2 s	199.4 s	44.8 s
`queues.pl`	47 l	0.6 s	0.7 s	4.1 s	1.5 s
`terms.pl`	97 l	2.5 s	2.6 s	308.7 s	4.3 s
`trees.pl`	72 l	1.4 s	1.6 s	12.6 s	3.2 s

TAB. 6.4 – Comparaison des performances OCaml/CHR

chaque occurrence de variable du programme, alors qu'avec l'unification des variables de type, on obtient une seule variable de type par variable du programme. Dans le cadre de l'inférence de type pour les prédicats, un phénomène similaire se produit lors de l'application de l'heuristique sur les types des arguments des prédicats. De plus, celle-ci traite plusieurs clauses simultanément, ce qui amplifie le phénomène, ce qui peut conduire à des cas extrêmes comme celui du fichier `terms.pl`.

Comparaison entre treillis et quasi-treillis

Le tableau 6.5 permet de mesurer l'impact du passage d'une structure de treillis à une structure de quasi-treillis. Il compare l'utilisation de ces structures à travers les performances de vérification des types, avec inférence du type des variables. Ces tests ont étés effectués sur 17 bibliothèques SICStus Prolog. À chaque bibliothèque correspond une structure de quasi-treillis issue des déclarations de types. Pour chacune de ces structures, une structure de treillis est obtenue en lui ajoutant $\top$ et $\bot$.

La première colonne indique la bibliothèque. Les deux colonnes suivantes indiquent les temps pour la vérification des types. La deuxième colonne indique le temps de vérification des types en utilisant la structure de treillis. La troisième colonne indique le temps de vérification des types en utilisant une structure de quasi-treillis. Enfin la quatrième colonne indique le rapport colonne trois sur colonne deux. Cette comparaison permet d'évaluer le coût de l'énumération des bornes des variables de type. Dans le cas du quasi-treillis, l'énumération se fait

pour les bornes inférieures et supérieures, c'est-à-dire que les règles (Intro ↑) et (Intro ↓) sont toutes deux utilisées.

Bibliothèque	Vérification des types		Rapport
	Treillis	Q-Treillis	
`arrays`	0.94 s	3.67 s	3.90
`assoc`	2.18 s	5.02 s	2.30
`atts`	1.90 s	3.21 s	1.68
`bdb`	3.17 s	5.89 s	1.85
`charsio`	0.41 s	0.96 s	2.34
`clpb`	11.43 s	31.01 s	2.71
`clpr`	46.85 s	69.63 s	1.48
`fastrw`	0.26 s	0.44 s	1.69
`heaps`	1.87 s	4.44 s	2.37
`jasper`	0.98 s	1.60 s	1.63
`lists`	1.87 s	3.50 s	1.87
`ordsets`	2.38 s	5.89 s	2.47
`queues`	0.43 s	1.03 s	2.39
`sockets`	1.83 s	4.33 s	2.36
`terms`	1.35 s	3.25 s	2.40
`trees`	0.81 s	2.39 s	2.95
`ugraphs`	14.14 s	53.19 s	3.76

TAB. 6.5 – Comparaison des performances Treillis/Quasi-treillis

On constate que l'algorithme pour les quasi-treillis se comporte très bien, puisqu'il est en moyenne 2.42 et au pire 4 fois plus lent que l'algorithme sur les treillis alors que le problème passe d'une complexité polynomiale en $O(n^3)$ à une complexité NP. Ces performances sont obtenues grâce à une heuristique d'énumération des bornes privilégiant l'énumération de la borne manquante sur les variables déjà partiellement bornées. C'est-à-dire que l'on applique en priorité les règles (Intro ↑) (resp (Intro ↓)) sur des variables α telles qu'il existe une contrainte de la forme $\tau \leq \alpha$ (resp. $\alpha \leq \tau$) dans le système de contraintes courant.

Ces performances montrent que la structure de quasi-treillis est utilisable en pratique. Ainsi, nous utilisons de telles structures dans les bibliothèques de type que nous avons écrites pour ISO Prolog, GNU-Prolog et SICStus Prolog, et qui sont décrites par les figures 7.1 page 108 et 7.2 page 109.

Deuxième partie

Typage des langages logiques avec contraintes

Chapitre 7

Système de types

Dans ce chapitre, nous définissons et étudions dans un premier temps un système de type pour CLP($\mathcal{X}$), issu du système de Fages et Paltrinieri [23, 22]. Ce dernier ajoute la règle de sous-typage au système de Mycroft et O'Keefe [50]. Nous y intégrons la notion de surcharge [20, 13]. De manière similaire à ce que nous avons fait pour le système de Fages et Paltrinieri dans [22], nous prouvons la cohérence du système par rapport à la résolution CSLD, ainsi que par rapport à un modèle d'exécution typé, dans lequel le type est conservé sur les variables durant l'exécution.

Dans un deuxième temps, nous présentons les choix qui ont été faits pour la structure des types. Bien que le système supporte des ordres partiels quelconques, la décidabilité de la satisfiabilité des contraintes de sous-typage non structurel non homogène dans ces structures est encore, à notre connaissance, un problème ouvert. En revanche, les structures de quasi-treillis peuvent être utilisées, grâce aux algorithmes développés dans la première partie du mémoire. Ils constituent une structure plus riche que les treillis. En particulier, le type vide $\bot$ n'est pas imposé dans ces structures, ce qui est important pour l'inférence de type des variables.

7.1 Programmes bien typés

7.1.1 Notations

On rappelle que les programmes sont formés à partir d'un ensemble dénombrable $\mathcal{W}$ de variables notées $\mathtt{X}, \mathtt{Y}, \ldots$, d'un ensemble dénombrable S_f de symboles de fonction, donnés avec leur arité et notés $\mathtt{f}/m, \mathtt{g}/n, \ldots$ et d'un ensemble dénombrable S_p de symboles de prédicats, donnés avec leur arité et notés $\mathtt{p}/m, \mathtt{q}/n, \ldots$. Les termes formés sur $\mathcal{W}, S_f$ sont notés $t, t_1, \ldots$. Les substitutions de termes sont notées $\sigma, \sigma_1, \ldots$.

L'ensemble des types $\mathcal{T} = \mathcal{T}(\mathcal{S}_\mathcal{V})$ est basé sur la signature[1] $\mathcal{S}_\mathcal{V} = (\mathcal{K} \cup \mathcal{V}, \leq_\mathcal{K} \cup \mathrm{id}, \mathcal{L}^+, \mathcal{L}^-, ar_{\mathcal{K}\cup\mathcal{V}})$. On notera les types $\tau, \tau_1, \ldots$ et les paramètres (ou variables de types) $\alpha, \beta, \ldots$. On supposera de plus l'existence d'un constructeur de type constant pred, qui sera utilisé comme type pour les prédicats. On notera $\rho, \rho_1, \ldots$ les substitutions de types.

La notion de schéma de type permet de donner un type aux symboles de fonction et de prédicat :

Définition 7.1 *Un* schéma de types *est une expression de la forme* $\forall\bar{\alpha}.\tau_1 \times \ldots \times \tau_n \to \tau$*, où* $\bar{\alpha}$ *est l'ensemble des paramètres qui apparaissent dans* $\tau_1, \ldots, \tau_n, \tau$*.*

Lorsqu'il n'y a pas d'ambiguïté sur $\bar{\alpha}$, nous nous permettrons d'omettre $\forall\bar{\alpha}.$.

À chaque symbole de fonction $\mathtt{f}/n$ on associe un ensemble non vide $types(\mathtt{f}/n)$ de schémas de types de la forme $\forall\bar{\alpha}.\tau_1 \times \ldots \times \tau_n \to \tau$. On associe également à chaque symbole de prédicat $\mathtt{p}/n$ un ensemble non vide de schémas de types $types(\mathtt{p}/n)$, tous de la forme $\forall\bar{\alpha}.\tau_1 \times \ldots \times \tau_n \to \mathrm{pred}$. En particulier, on supposera par la suite que $\forall\alpha.\alpha \times \alpha \to \mathrm{pred} \in types(=/2)$.

Un *environnement de typage* Γ est une fonction d'une partie finie de $\mathcal{W}$ dans $\mathcal{T}$ qui associe un type aux variables des programmes. Si Γ et Γ' sont deux environnements de typage de domaines distincts, alors Γ, Γ' est l'environnement qui à $\mathtt{X}$ associe $\Gamma(\mathtt{X})$ si $\mathtt{X} \in dom(\Gamma)$ ou $\Gamma'(\mathtt{X})$ si $\mathtt{X} \in dom(\Gamma')$.

7.1.2 Règles de typage

Les *jugements de typage* sont de la forme $\Gamma \vdash S$, où Γ est un environnement de typage et S une assertion de typage. En particulier, $\Gamma \vdash t : \tau$ se lit : "t a le type τ dans l'environnement Γ". Ces jugements sont dérivés à partir de la table 7.1.

La règle de sous-typage *(Sub)* exprime si t a le type τ dans l'environnement Γ alors il peut être vu comme ayant le type τ' dans le même environnement Γ, et ceci pour tous les supertypes τ' de τ. Cette règle permet donc d'utiliser un terme de type τ là où un terme de type $\tau' \geq \tau$ est attendu.

La règle *(Var)* exprime que le type des variables est donné par l'environnement de typage. La règle *(Func)* décrit le typage des termes. L'utilisation d'une substitution ρ correspond à l'instanciation du schéma de types de $\mathtt{f}/n$. Plus précisément, pour un schéma de type $\forall\bar{\alpha}.\tau_1 \times \ldots \times \tau_n \to \tau$ associé à $\mathtt{f}/n$ (c'est-à-dire appartenant à $types(\mathtt{f}/n)$), et pour une substitution de type ρ, si chaque t_i a le type $\rho(\tau_i)$ dans l'environnement Γ, alors $\mathtt{f}(t_1, \ldots, t_n)$ a le type $\rho(\tau)$ dans l'environnement Γ. La possibilité de "choisir" le schéma de type à utiliser pour une occurrence d'un symbole de fonction donné exprime la surcharge de ce symbole. Ainsi, si, pour chaque symbole, l'ensemble de ses schémas de type se réduit à un singleton, on retrouve le système sans surcharge de Fages et Paltrinieri [23].

[1] définie comme dans le chapitre 3 page 39.

(Sub)	$\dfrac{\Gamma \vdash t : \tau, \tau \leq \tau'}{\Gamma \vdash t : \tau'}$	
(Var)	$\Gamma \vdash \mathtt{X} : \Gamma(\mathtt{X})$	
(Func)	$\dfrac{\Gamma \vdash t_1 : \rho(\tau_1) \ , \ldots , \ \Gamma \vdash t_n : \rho(\tau_n)}{\Gamma \vdash \mathtt{f}(t_1, \ldots, t_n) : \rho(\tau)}$	si $\forall \bar{\alpha}.\tau_1 \times \ldots \times \tau_n \rightarrow \tau \in \mathit{types}(\mathtt{f}/n)$ et ρ est une substitution de type
(Atom)	$\dfrac{\Gamma \vdash t_1 : \rho(\tau_1) \ , \ldots , \ \Gamma \vdash t_n : \rho(\tau_n)}{\Gamma \vdash \mathtt{p}(t_1, \ldots, t_n) \ \mathit{Atom}}$	si $\forall \bar{\alpha}.\tau_1 \times \ldots \times \tau_n \rightarrow \mathrm{pred} \in \mathit{types}(\mathtt{p}/n)$ et ρ est une substitution de type
(Query)	$\dfrac{\Gamma \vdash A_1 \ \mathit{Atom} \ , \ldots , \ \Gamma \vdash A_n \ \mathit{Atom}}{\Gamma \vdash A_1, \ldots, A_n \ \mathit{Query}}$	
(Head)	$\dfrac{\Gamma \vdash t_1 : \rho(\tau_1) \ , \ldots , \ \Gamma \vdash t_n : \rho(\tau_n)}{\Gamma \vdash \mathtt{p}(t_1, \ldots, t_n) \ \mathit{Head}_{\mathtt{p}:\tau_1,\ldots,\tau_n}}$	si $\forall \bar{\alpha}.\tau_1 \times \ldots \times \tau_n \rightarrow \mathrm{pred} \in \mathit{types}(\mathtt{p}/n)$ et ρ est un renommage de type
(Clause)	$\dfrac{\Gamma \vdash H \ \mathit{Head}_{\mathtt{p}:\tau_1,\ldots,\tau_n} \ , \ \Gamma \vdash Q \ \mathit{Query}}{\Gamma \vdash H \mathtt{:-} \ Q \ \mathit{Clause}_{\mathtt{p}:\tau_1,\ldots,\tau_n}}$	

TAB. 7.1 – Système de type avec surcharge pour CLP($\mathcal{X}$) [20]

.

La règle *(Atom)* est similaire à la règle *(Func)* et est utilisée pour typer les appels de prédicats et les contraintes. Le type résultat des prédicats est cependant limité à pred. La conclusion de la règle exprime que $\mathtt{p}(t_1, \ldots, t_n)$ est un atome bien typé dans l'environnement Γ. La règle *(Query)* exprime simplement qu'un but est bien typé dans l'environnement Γ si les atomes qui le composent sont eux aussi bien typés dans l'environnement Γ.

La conclusion de la règle *(Head)* signifie que $\mathtt{p}(t_1, \ldots, t_n)$ est un tête de clause bien typée dans l'environnement Γ. Cette règle diffère de la règle *(Atom)* dans le fait que ρ soit un renommage des paramètres et non une substitution quelconque. Elle permet ainsi de vérifier le principe de *généricité des définitions* énoncé par Lakshman et Reddy [44]. Ce principe établit que le type d'une tête de clause d'un prédicat doit être à un renommage près équivalent au type du prédicat.

Exemple 7.1 : Considérons le prédicat `length`/2, avec les types $\mathit{types}(\mathtt{length}/2) = \{\forall\alpha.\mathrm{list}(\alpha) \times \mathrm{int} \rightarrow \mathrm{pred}\}$. Voici deux clauses qui définissent ce prédicat :

```
length([],0).
length([_|L],N) :- length(L,NL), N is NL + 1.
```

Ces deux clauses vérifient bien la notion de généricité des définitions car rien ne force le type des argument des listes (c'est-à-dire le paramètre α sera simplement renommé lors de l'application des règles de typage). Au contraire, une clause comme :

```
length([1],1).
```

va violer la condition de généricité des définitions car le type des éléments de la liste est ici imposé à être int, c'est-à-dire que la variable de type α sera instanciée et non pas simplement renommée. ◇

De plus, le jugement *Head* issu de cette règle se voit attribué une annotation précisant le type du prédicat sur lequel il porte. Ainsi, le jugement $\Gamma \vdash \mathtt{p}(t_1, \ldots, t_n)$ $Head_{\mathtt{p}:\tau_1,\ldots,\tau_n}$ se lit ¡¡dans l'environnement Γ, $\mathtt{p}(t_1, \ldots, t_n)$ est une tête de clause bien typée pour le prédicat $\mathtt{p}/n$ pris avec le type $\forall\bar{\alpha}.\tau_1 \times \ldots \times \tau_n \rightarrow$ pred¿¿.

Enfin la règle *(Clause)* exprime que H `:-` Q est une clause bien typée dans l'environnement Γ pour le prédicat $\mathtt{p}/n$ pris avec le type $\forall\bar{\alpha}.\tau_1 \times \ldots \times \tau_n \rightarrow$ pred si H est une tête bien typée dans Γ pour le même prédicat avec le même type, et si Q est un but bien typé dans Γ.

Une clause H `:-` Q d'un prédicat $\mathtt{p}/n$ est *bien typée* si pour tout schéma de type de la forme $\forall\bar{\alpha}.\tau_1 \times \ldots \times \tau_n \rightarrow$ pred $\in$ *types*$(\mathtt{p}/n)$, il existe un environnement Γ tel que l'on peut dériver des règles de la table 7.1 le jugement $\Gamma \vdash H$ `:-` Q $Clause_{\mathtt{p}:\tau_1,\ldots,\tau_n}$. Un programme est *bien typé* si toutes ses clauses sont bien typées.

Substitutions de type et règles de typage

On étend les substitutions de types aux environnements de typage de manière naturelle : soit ρ une substitution de type et Γ un environnement de typage, alors $\rho(\Gamma)$ est défini par $\rho(\Gamma)(\mathtt{X}) = \rho(\Gamma(\mathtt{X}))$. On étend également les substitutions de type aux assertions de typage : $\rho(t : \tau) \stackrel{def}{=} t : \rho(\tau)$, les autres assertions restant inchangées par l'application de la substitution. Ces extensions permettent d'énoncer la propriété suivante d'instanciation des jugements :

Propriété 7.2 *Pour tout environnement de typage Γ, pour tout jugement de typage $\Gamma \vdash S$, si S n'est pas de la forme C Clause ou H Head, alors pour toute substitution de typage ρ, $\rho(\Gamma) \vdash \rho(S)$.*

Démonstration : On le montre par induction sur la dérivation de typage. Considérons la règle *(Sub)*. Par induction, $\rho(\Gamma) \vdash t : \rho(\tau)$. Par la propriété 3.7, $\rho(\tau) \leq \rho(\tau')$. Donc on a bien $\rho(\Gamma) \vdash t : \rho(\tau')$. Dans le cas de la règle *(Var)*, on a par définition $\rho(\Gamma) \vdash \mathtt{X} : \rho(\Gamma(\mathtt{X}))$. Dans le cas de la règle *(Func)*, si ρ' est la substitution de type utilisée lors de l'application de la règle, pour tout $1 \leq i \leq n$, on obtient par induction $\rho(\Gamma) \vdash t_i : \rho \circ \rho'(\tau_i)$, ce qui permet de déduire $\rho(\Gamma) \vdash \mathtt{f}(t_1, \ldots, t_n) : \rho \circ \rho'(\tau)$. On montre les cas *(Atom)* et *(Query)* de manière similaire. □

7.2 Cohérence au modèle d'exécution CSLD

Nous étudions ici la cohérence de la notion de programme bien typé par rapport à la résolution CSLD. Cette cohérence s'exprime à travers le théorème 7.3, dit d'auto-réduction. Ce théorème exprime que si un programme est bien typé, tout but bien typé n'engendre, par résolution CSLD en utilisant ce programme, que des buts bien typés.

Théorème 7.3 *Soit P un programme CLP($\mathcal{X}$) bien typé et B un but bien typé dans un certain environnement de typage Γ, c'est à dire $\Gamma \vdash B$ Query. Alors pour tout résolvant CSLD B' de B, il existe un environnement de typage Γ' tel que $\Gamma, \Gamma' \vdash B'$ Query.*

Démonstration : On peut considérer, sans perte de généralité, que $B = c|\mathtt{p}(s), A$ et que B' est un résolvant CSLD de B avec la clause $\mathtt{p}(t)\mathtt{:-}\ d|A'$. On a alors $B' = c, d, s = t|A, A'$. Comme B est bien typé, on a $\Gamma \vdash c|\mathtt{p}(s), A$ *Query*. Comme $\Gamma \vdash p(s)$ *Atom*, $\Gamma \vdash s : \rho(\tau)$ pour un certain schéma de type $\tau \rightarrow \text{pred} \in types(\mathtt{p}/1)$ et une certaine substitution ρ. Comme le programme est bien typé, il existe Γ'' tel que $\Gamma'' \vdash \mathtt{p}(t)\mathtt{:-}\ d|A'$ $Clause_{\mathtt{p}:\tau}$. Soit $\Gamma' = \rho(\Gamma'')$. De par la propriété 7.2, $\rho(\Gamma'') \vdash d|A'$ *Query*, donc $\Gamma, \Gamma' \vdash c, d|A, A'$ *Query*. Reste à montrer que $\Gamma, \Gamma' \vdash s = t$ *Atom*. Comme $\Gamma'' \vdash \mathtt{p}(t)$ $Head_{\mathtt{p}:\tau}$, $\Gamma'' \vdash t : \tau$. En utilisant la propriété 7.2, $\rho(\Gamma'') \vdash t : \rho(\tau)$. Comme $\Gamma \vdash s : \rho(\tau)$, on obtient $\Gamma, \Gamma' \vdash s = t$ *Atom*. □

On peut remarquer que ce théorème n'est pas valable sans le principe de généricité des définitions exprimé au travers de la règle *(Head)*, comme le montre l'exemple suivant :

Exemple 7.2 : Considérons deux constantes $\mathtt{a}/0$ et $\mathtt{b}/0$ telles que $types(\mathtt{a}/0) = \{\tau_{\mathtt{a}}\}$ et $types(\mathtt{b}/0) = \{\tau_{\mathtt{b}}\}$ avec $\tau_{\mathtt{a}}$ et $\tau_{\mathtt{b}}$ deux constantes de types n'ayant pas de majorant commun. Soit le prédicat $\mathtt{p}/1$, avec $types(\mathtt{p}/1) = \{\forall\alpha.\alpha \rightarrow \text{pred}\}$. Si $\mathtt{p}/1$ est défini par la clause (non générique) : `p(a)` , alors le but `p(b)` est bien typé (en instanciant α à $\tau_{\mathtt{b}}$). Cependant, son résolvant CSLD est `a=b`, qui lui est mal typé puisque $types(\mathtt{=}/2) = \{\forall\alpha.\alpha \times \alpha \rightarrow \text{pred}\}$, et que $\tau_{\mathtt{a}}$ et $\tau_{\mathtt{b}}$ deux constantes de types n'ont pas de supertype commun. ◇

7.3 Cohérence au modèle d'exécution avec substitutions

Comme expliqué dans le chapitre 2, les solveurs de contraintes effectuent entre deux étapes de résolution CSLD une étape de simplification et de substitution. Le théorème 7.3 exprime la propriété d'auto-réduction uniquement par rapport

aux étapes de résolution CSLD, c'est-à-dire par rapport au modèle d'exécution abstrait procédant par accumulation de contraintes, et non par rapport aux étapes de simplification et de substitution. L'exemple suivant montre que si l'on effectue les substitutions, on peut aboutir à un but mal typé :

Exemple 7.3 : Supposons la relation de sous-typage int$\leq_{\mathcal{K}}$term et supposons les types suivants : *types*(a/0) = {term}, *types*(p/1) = {term → pred} et *types*(q/1) = {int → pred}. Supposons enfin que p/1 soit défini par la clause `p(a).` . Considérons à présent le but `p(X),q(X)`. Ce but est bien typé dans un environnement Γ où Γ(X) = int. Il se réduit par une étape de résolution CSLD sur le but `X=a | q(X)` qui est également bien typé dans ce même environnement Γ. En revanche, si on effectue la substitution σ : X ↦ a, on obtient le but `a=a | q(a)` qui n'est pas bien typé car a ne peut être typé avec le type int. ◇

Ceci peut être vu comme une erreur de type se produisant à l'exécution. Cependant les systèmes de types prescriptifs en général ne sont pas à l'abri de ce genre d'erreur. Ainsi le système de type de ML ne détecte pas les divisions par zéro, une erreur étant déclenchée dans ce cas. De même, une erreur sera déclenchée si on essaie d'extraire la tête d'une liste vide à l'aide de la fonction `List.hd` en OCaml. Ces deux erreurs peuvent être considérées comme des erreurs de type, puisque le type du deuxième argument de la division devrait être "entier différent de 0" et le type de l'argument de `List.hd` devrait être "liste non vide". Dans le contexte des langages impératifs, le langage Ada définit de nombreux types numériques, parmi lesquels le type Natural pour les entiers positifs ou nuls. Avec la déclaration `X : Natural`, l'instruction `X := X - 1 ;` est bien typée. Cependant, si `X` vaut `0` au début de l'instruction, l'exception `CONSTRAINT_ERROR` sera levée. Ces exemples montrent simplement que ces systèmes de types ne sont cohérents que vis-à-vis d'un modèle d'exécution abstrait où ces erreurs n'apparaissent pas.

Il est donc intéressant d'étudier la cohérence du système de type par rapport à un modèle d'exécution qui prend en compte les opérations de substitution. L'exemple 7.3 montre que le système n'est pas cohérent avec un modèle d'exécution correspondant à la combinaison de la résolution CSLD avec des étapes de substitutions. Cette non cohérence est liée à la combinaison du sous-typage avec l'absence d'information sur le flot de données au moment de la vérification des types. Dans le cadre des langages à objets, Palsberg, O'Keefe et Smith [53, 54] ont montré que l'inférence de type en présence de sous-typage était équivalente à un calcul sur le flot de données du programme. Or, lors de l'unification, si on applique les substitutions, les données peuvent circuler à la fois de gauche à droite et de droite à gauche comme le montre l'exemple suivant :

Exemple 7.4 : Considérons la contrainte `X=Y` et supposons qu'au moment où elle est posée, la variable `X` soit unifiée avec le terme `t(U,b)` et la variable `Y` avec le

terme `t(a,V)`. Alors la variable `U` sera unifiée avec le terme `a`, ce qui signifie que de l'information est transmise de `Y` vers `X`. De même, la variable `V` sera unifiée avec le terme `b`, ce qui signifie que de l'information est transmise de `X` vers `Y`. ◇

L'exemple 7.5 montre, de manière informelle, la difficulté d'avoir un système de type permettant les coercitions de type entre domaines de contraintes, tout en donnant des types différents aux éléments de ces domaines :

Exemple 7.5 : Considérons un but contenant le sous-but `X #= Z-2, X #<=> Y`, où `#<=>/2` est une contrainte CLP($\mathcal{B}$) et `#=/2` est une contrainte CLP($\mathcal{FD}$). On suppose que ce but est bien typé dans le système de type et que les booléens sont représentés par `0` et `1`, ce qui permet de les utiliser conjointement avec les éléments de CLP($\mathcal{FD}$). Le type de la variable `X` doit être compatible avec celui est valeurs booléennes `0` et `1`, puisque cette variable est utilisée avec une contrainte booléenne. Cependant, la contrainte `#=/2` peut avoir au préalable provoqué l'instanciation de `X` à une valeur non booléenne, comme `2`. Dans ce cas, le sous-but devient `2 #= Z-2, 2 #<=> Y`, qui devra être mal typé puisque `2`, qui est alors utilisé avec `#<=>/2`, n'est pas une valeur booléenne. ◇

7.3.1 Modèle d'exécution typé

Une première solution à ce problème de flot de données consiste à fixer celui-ci, à l'aide de modes comme dans [62], ou dans le langage comme dans Mercury [66]. Cependant, l'utilisation des modes est considérée comme trop restrictive en général. En effet il est souvent difficile, voir impossible, de déterminer si contrainte va réduire suffisament le domaine des variables sur lesquelles elle est posée pour en instancier une partie.

Une autre solution consiste à utiliser un modèle d'exécution typé dans lequel le type des variables est conservé lors de l'exécution [22].

Cette solution nécessite l'utilisation de *contraintes de type* sur les variables, ainsi que sur certains termes du programme. Ces contraintes sont de la forme $t : \tau$, où t est un terme et τ un type.

Définition 7.4 *Étant donné un domaine de contraintes $\mathcal{X}$, le domaine de contraintes typé qui lui correspond est $\mathcal{X} \cup 2^{\mathcal{X}}$ où :*

- *Les types atomiques, c'est-à-dire correspondant à des constructeurs de type d'arité vide, sont interprétés comme des sous-ensembles distingués de $\mathcal{X}$.*
- *Les constructeurs de types d'arité non vide sont interprétés comme des fonctions des sous-ensembles de $\mathcal{X}$ vers les sous-ensembles de $\mathcal{X}$.*
- *L'interprétation des types est compatible avec la relation de sous-typage $\leq$, ainsi qu'avec les déclarations de type pour les symboles de fonction et de prédicats.*

– *Une valuation ρ satisfait une contrainte de type $t : \tau$ si $t\rho \in \tau\rho$.*

Exemple 7.6 : Prenons le domaine de Herbrand $\mathcal{H}$, avec les constructeurs de type list(.), int et term ; list$\leq_{\mathcal{K}}$term et int$\leq_{\mathcal{K}}$term. On considère les déclarations usuelles pour les listes, les nombres entiers sont déclarés avec le type int et tous les autres symboles de fonctions sont déclarés avec le type term $\times \ldots \times$ term $\rightarrow$ term. Le domaine de contraintes typés qui lui correspond est $\mathcal{H} \cup 2^{\mathcal{H}}$. Le type int dénote l'ensemble des entiers $\mathbb{Z}$. Le constructeur list(.) dénote la fonction qui à tout ensemble de termes T associe l'ensemble des listes dont les éléments appartiennent à T. Ceci fourni l'interprétation des contraintes de type : par exemple la contrainte $\texttt{[X]} : \text{list}(\alpha)$ sera satisfaite, entre autres, par la valuation $\rho : \alpha \mapsto \text{int}, \texttt{X} \mapsto 2$. $\diamond$

Lemme 7.5 *Dans un système de contraintes typé, $\texttt{X} : \tau \wedge \texttt{X} = t$ implique $t : \tau$.*

Démonstration : Pour toute valuation ρ, si $\rho \models \texttt{X} : \tau \wedge \texttt{X} = t$ alors $\texttt{X}\rho \in \tau\rho$ et $\texttt{X}\rho = t\rho$, donc $t\rho \in \tau\rho$. □

A tout environnement de typage Γ, il est possible d'associer un ensemble de contraintes de type qui lui correspond : $\{\texttt{X} : \tau \mid \texttt{X} \in dom(\Gamma) \wedge \Gamma(\texttt{X}) = \tau\}$. Par exemple, l'ensemble correspondant à $\Gamma : \texttt{X} \mapsto \text{int}, \texttt{Y} \mapsto \text{term}$ est $\{\texttt{X} : \text{int}, \texttt{Y} : \text{term}\}$. Cet ensemble de contraintes permet de définir le but (resp. la clause) dit TCLP associé à une but (resp. une clause) bien typé :

Définition 7.6 *Le but (resp. la clause) TCLP associé à un but Q (resp. une clause H :- Q) bien typé dans un environnement Γ est le but C_Γ, Q (resp. la clause H :- C_Γ, Q), où C_Γ est l'ensemble des contraintes de type associées à Γ.*

Théorème 7.7 *Soit P un programme TCLP associé à un programme CLP($\mathcal{X}$) bien typé. Soit Q un but TCLP associé à un but bien typé dans un environnement Γ. Si Q' est un résolvant CSLD de Q, alors si Γ' est l'environnement issu des contraintes de type de Q', $\Gamma' \vdash Q'$ Query. De plus, si Q' contient une égalité $\texttt{X} = t$, alors $\Gamma' \vdash Q'[t/\texttt{X}]$ Query.*

Démonstration : Comme Q est un but TCLP, il est bien typé dans l'environnement Γ. D'après le théorème 7.3, Q' est bien typé dans un environnement Γ' étendant Γ, or la construction de Γ' dans la preuve du théorème 7.3 correspond exactement aux contraintes de type présentes dans Q', ce qui prouve le premier point. Considérons à présent une contrainte $\texttt{X} = t$ dans Q'. Nous avons également une contrainte $\texttt{X} : \tau$ dans Q', qui combinée avec $\texttt{X} = t$ implique $t : \tau$, de par le lemme 7.5. Comme l'interprétation des contraintes de type respecte les déclarations de type, on obtient $\Gamma' \vdash t : \tau$. En remplaçant X par t dans la dérivation de $\Gamma' \vdash Q'$ Query, on obtient une dérivation de $\Gamma' \vdash Q'[t/\texttt{X}]$ Query. □

L'effet des contraintes de types ajoutées dans les programmes et les buts TCLP est d'empêcher les étapes de substitutions conduisant à des buts mal typés :

Exemple 7.7 : Reprenons l'exemple 7.3. Le but $Q = \mathtt{p(X)}, \mathtt{q(X)}$ est bien typé dans l'environnement Γ : $\mathtt{X} \mapsto$ int. Le but TCLP qui lui est associé est $\mathtt{X}$: int | $\mathtt{p(X)}, \mathtt{q(X)}$. Ce but ne peut se dériver en $\mathtt{X} = \mathtt{a}, \mathtt{X}$: int | $\mathtt{q(X)}$, car la contrainte $\mathtt{X} = \mathtt{a}, \mathtt{X}$: int est insatisfiable. ◇

7.4 Choix des types

Le choix des types donnés aux symboles de fonctions et de prédicats, tout comme celui de l'ordre entre les constructeurs de types, est en partie arbitraire. Nous discutons ici de différents choix de structures et de types, puis exposons les structures de types que nous avons utilisées pour ISO-Prolog et pour une combinaison de différents domaines de contraintes dans SICStus Prolog.

7.4.1 Méta-programmation

Lorsque l'on cherche à donner un type à certains prédicats de méta-programmation, on peut parfois hésiter entre le type term et une variable de type. C'est par exemple le cas du prédicat `functor`/3, qui extrait d'un terme le nom et l'arité de son symbole de tête. Deux types sont possibles pour ce prédicat : term × atom × int → pred et α × atom × int → pred. En effet, les deux types correspondent bien à la définition de ce prédicat, puisqu'il accepte n'importe quel terme comme premier argument. Dans les déclarations de type pour les dialectes que nous avons étudié, nous avons opté pour le premier type, afin de nous conformer au comportement de `functor`/3. L'exemple suivant montre le problème rencontré avec le second type :

Exemple 7.8 : Considérons une structure sans le type term, et que l'on utilise le type α × atom × int → pred pour `functor`/3. Considérons également le but `T=2, functor(T,[],0)`. Il se réduit en `T=2,T=[]`, qui n'est pas bien typé, car il n'y a pas de type correct pour `T`. ◇

L'intuition est que si un argument d'un prédicat a un type avec des paramètres, les termes qui correspondent à ces paramètres devraient être vus comme des boîtes noires. Or le prédicat `functor`/3, "ouvre" au contraire cette boîte, puisqu'il en analyse le contenu en faisant correspondre à un terme son symbole de tête et son arité. Au niveau des prédicats écrits par le programmeur, cette intuition est formalisée dans le système de type par la notion de généricité des définitions.

7.4.2 Surcharge et sous-typage

Le sous-typage et la surcharge peuvent chacun représenter une alternative dans la représentation de certains types. Nous illustrons ici ce phénomène tout d'abord à travers l'exemple du type io_mode. Ce type représente le différents modes possibles du prédicat `open`/3 utilisé pour obtenir un flux de lecture ou d'écriture dans un fichier, à savoir lecture (`read`), écriture (`write`), où écriture à la fin (`append`). Ces trois termes n'ayant pas d'argument, il se doivent également d'avoir le type atom, qui représente les termes sans argument. En particulier, ce type est utilisé pour le deuxième argument du prédicat `functor`/3, ou bien encore comme le type des chaînes de caractères[2]. Pour permettre les différentes utilisations possibles de ces termes, il est possible, soit de surcharger ces symboles de fonction en leur donner à la fois le type atom et le type io_mode, soit de faire du type io_mode un sous-type de atom. Le répercussions de ce choix vont se situer au niveau de ce qui sera autorisé ou pas par le système de type :

Exemple 7.9 : Considérons le prédicat `p`/3, qui se contente de faire un appel à `open`/3. Ce prédicat, ayant le même type que le prédicat `open`/3, c'est-à-dire atom × io_mode × stream → pred, est défini par la clause (erronée) suivante :

```
p(Fichier, Mode, Flux) :- open(Mode, Fichier, Flux).
```

On constate ici une inversion des deux premiers arguments. Cette inversion ne sera pas détectée par le système si io_mode est un sous-type de atom : il est en effet possible de donner aux variables `Fichier` et `Mode` le type io_mode pour que la clause soit bien typée. En revanche, si les termes `read`, `write` et `append` sont surchargés, c'est-à-dire que le type io_mode n'est pas un sous-type de atom, alors la clause est mal typée. ◇

Le symbole `-`/2 représente également un bon exemple d'utilisation de la surcharge. En effet ce symbole est utilisé à la fois pour représenter la soustraction et pour coder les paires, la paire (`a`, `b`) étant codée par `a-b`. Ce codage des paires est en particulier utilisé pour le prédicat `keysort`/2 et dans la bibliothèques `ugraphs` de SICStus Prolog. Dans le cas où `-`/2 est utilisé pour la soustraction, on veut lui donner un type arithmétique, tel que int_expr × int_expr → int_expr, alors que dans le cas du codage des paires, on lui donnera le type $\alpha \times \beta \rightarrow \text{pair}(\alpha, \beta)$. Contrairement à l'exemple du type io_mode précédent, il n'est pas possible d'utiliser le sous-typage pour obtenir un seul type pour `-`/2, ce qui montre bien l'intérêt de la surcharge dans le système de types.

[2]Il y a en général deux représentations possibles pour les chaîne de caractères dans les dialectes Prolog/CLP que nous étudions : les symboles de fonction sans arguments, la chaîne de caractère étant représentée par le nom du symbole, et les listes de caractères ou d'entiers.

7.4.3 Expressions arithmétiques

Le typage des expressions arithmétiques est complexe dans le sens où ces expressions sont naturellement surchargées : un exemple est l'opération d'addition, qui peut prendre deux arguments de types différents, comme un entier et un flottant. Il existe plusieurs manières de gérer le type des expressions arithmétique. Si on considère simplement les nombres entiers et flottants, on peut distinguer deux familles d'expressions, selon que leur résultat sera entier ou flottant, en leur donnant le type int_expr ou float_expr. Cependant, des expressions entières peuvent être des sous-expressions d'expressions flottantes. Ceci peut être reflété en surchargeant les opérations arithmétiques : par exemple $types(+/2) = \{\text{int\_expr} \times \text{int\_expr} \rightarrow \text{int\_expr}, \text{float\_expr} \times \text{int\_expr} \rightarrow \text{float\_expr}, \text{int\_expr} \times \text{float\_expr} \rightarrow \text{float\_expr}, \text{float\_expr} \times \text{float\_expr} \rightarrow \text{float\_expr}\}$.

Un autre point de vue consiste à voir les expressions entières comme un cas particulier des expressions flottantes, puisque les entiers sont un sous-ensemble des réels. Dans ce cas, on ajoute l'inégalité $\text{int\_expr} \leq \text{float\_expr}$ à la relation de sous-typage. On pourrait ainsi donner le type $\text{float\_expr} \times \text{float\_expr} \rightarrow \text{float\_expr}$ à l'opérateur +/2, ce dernier acceptant alors également les expressions entières comme argument, grâce à la règle de sous-typage. Cependant un tel type est moins précis, puisque l'addition de deux expressions entières donne une expression flottante. Il est possible de regagner cette précision en surchargeant le type des opérateurs. Ainsi, $types(+/2) = \{\text{int\_expr} \times \text{int\_expr} \rightarrow \text{int\_expr}, \text{float\_expr} \times \text{float\_expr} \rightarrow \text{float\_expr}\}$. Une autre manière de regagner cette précision serait d'ajouter la possibilité d'avoir des contraintes de type dans les schéma de type. Ainsi, on pourrait donner le type $\forall\alpha.\alpha \times \alpha \rightarrow \alpha, \alpha \leq \text{float\_expr}$ à +/2. Cependant, autoriser de tels schémas nécessiterait de modifier le système de type. En particulier, il faudrait pouvoir décider de l'implication de contraintes de sous-typage, ce qui reste encore un problème ouvert pour le sous-typage non structurel non homogène.

Dans les systèmes que nous avons étudiés, nous avons opté pour la première solution, la surcharge sans sous-typage entre int_expr et float_expr. Ce choix a été fait à cause du comportement de l'unification et des prédicats d'évaluation d'expressions arithmétiques dans ces systèmes. En effet, l'unification de l'entier `1` avec le flottant `1.0` échoue, bien qu'ils représentent en fait le même nombre. Il est cependant imaginable que cela ne soit pas le cas pour d'autres systèmes, auquel cas la deuxième solution serait plus adaptée.

7.4.4 Structures pour différents systèmes

Nous décrivons ici des structures des types que nous avons utilisées pour ISO-Prolog et pour la combinaison de CLP($\mathcal{B}$), CLP($\mathcal{FD}$), CLP($\mathcal{R}$) et CLP($\mathcal{Q}$) dans

SICStus Prolog.

Structure des types pour ISO-Prolog

Cette structure est représentée par la figure 7.1. Les types correspondant à des options de prédicats ont été regroupés sur la gauche et doivent être vus comme étant des sous-types directs de term.

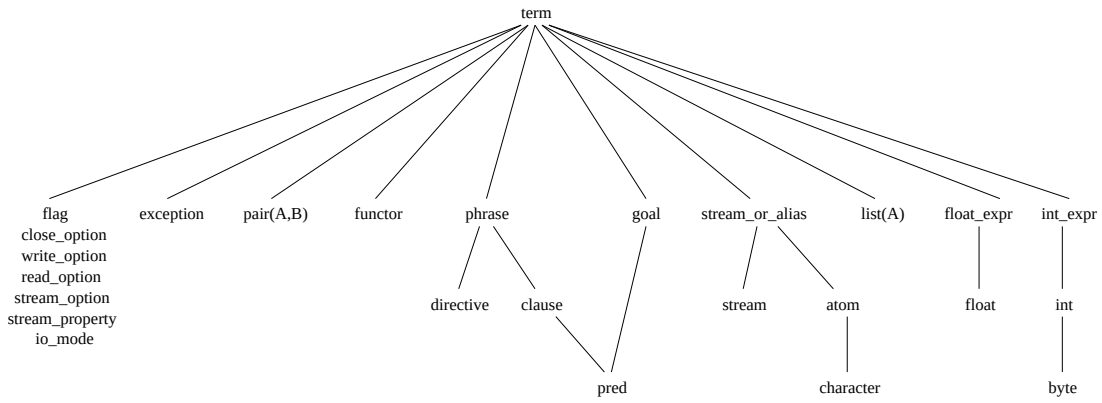


FIG. 7.1 – Structure des types ISO-Prolog

Comme expliqué précédemment, le type int_expr n'est pas un sous-type de float_expr. Le symbole `-/2` est utilisé comme constructeur de paires, il est donc également surchargé avec le type $\alpha \times \beta \rightarrow \text{pair}(\alpha, \beta)$. Le type float_expr a comme sous-type le type float, qui correspond aux nombres flottants. Le type int_expr a comme sous-types int et byte. La distinction entre nombres et expressions est utile car certains prédicats attendent un nombre et pas une expression en argument. Par exemple, `functor`/3 a le type term $\times$ atom $\times$ int $\rightarrow$ pred, et non le type term $\times$ atom $\times$ int_expr $\rightarrow$ pred.

Dans ISO-Prolog, il est possible de déclarer des "alias" pour les flux d'entrées/sorties. Ainsi l'alias `user`, représente l'entrée standard. N'importe quel terme atomique peut être utilisé comme alias. Le type stream_or_alias a donc été introduit comme une supertype des types atom et stream. Les caractères sont des termes atomiques dont le nom est exactement le caractère représenté. Ainsi, le type character est un sous-type de atom.

Le type des prédicats *pred* est à la fois un sous-type du type des buts goal et du type clause. En effet, une clause peut avoir un corps vide, ce qui signifie que des prédicats prenant comme argument une clause, par exemple `assert`/1, doivent également accepter pour ces arguments des termes correspondant à des prédicats. Une phrase dans un programme ISO-Prolog étant soit une clause, soit une directive, le type phrase est un supertype commun à clause et à directive.

Structure des types pour CLP($\mathcal{B}, \mathcal{FD}, \mathcal{R}, \mathcal{Q}$)

La structure des types pour la combinaison de CLP($\mathcal{B}$), CLP($\mathcal{FD}$), CLP($\mathcal{R}$) et CLP($\mathcal{Q}$) en SICStus Prolog est représentée par la figure 7.2. Cette figure représente seulement les types liés à ces domaines de contraintes et pas les types correspondant à SICStus Prolog en lui même, la structure de ces derniers étant une extension de la structure pour ISO-Prolog.

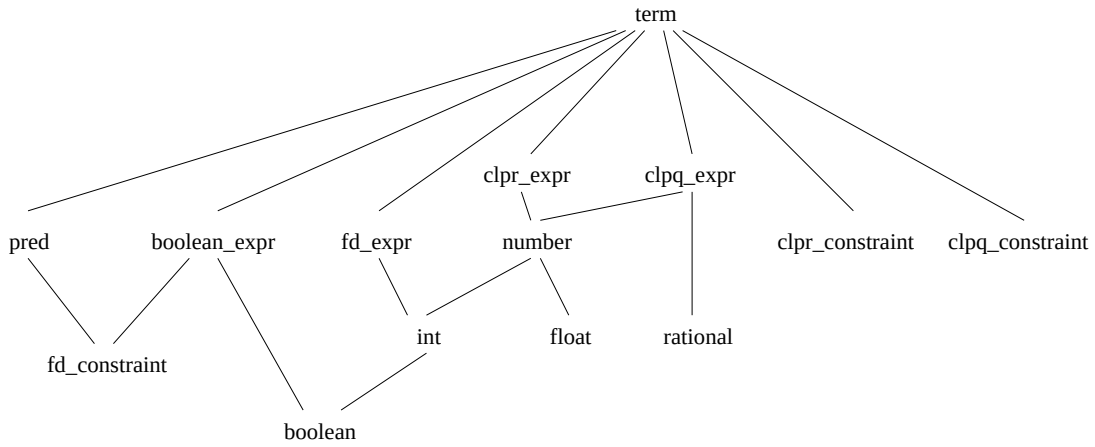


FIG. 7.2 – Structure des types pour CLP($\mathcal{B}, \mathcal{FD}, \mathcal{R}, \mathcal{Q}$)

Cette structure est un bon exemple de la complexité qui peut être atteinte avec la méta-programmation et la combinaison entre différents domaines de contraintes.

La méta-programmation apparaît au niveau des contraintes booléennes réifiées. Ainsi, une contrainte CLP($\mathcal{B}$) ou CLP($\mathcal{FD}$) peut être utilisée dans un but, mais également dans une expression booléenne. Ceci est reflété à travers le sous-typage fd_constraint $\leq$ pred et fd_constraint $\leq$ boolean_expr.

La combinaison des différents types numériques mène à une structure complexe pour les expressions arithmétiques correspondant à CLP($\mathcal{B}, \mathcal{FD}, \mathcal{R}, \mathcal{Q}$). En effet, les booléens `0` et `1` peuvent également être utilisés avec les expressions sur les domaines finis, et les entiers utilisés dans les expressions sur les réels ou les rationnels. Cependant, les expressions CLP($\mathcal{B}$) ne sont pas des expressions CLP($\mathcal{FD}$) ni des expressions CLP($\mathcal{R}$) ou CLP($\mathcal{Q}$). On remarque également que les types clpr_constraint et clpq_constraint représentant les contraintes CLP($\mathcal{R}$) et CLP($\mathcal{Q}$) ne sont pas des sous-types de pred. Ceci reflète le fait qu'elles ne peuvent être utilisées directement, mais doivent être évaluées par le prédicat {}/2.

Chapitre 8

Vérification des types

Le système de types présenté dans le chapitre 7 n'est pas déterministe : la règle *(Sub)* peut en effet être appliquée à n'importe quel endroit dans la dérivation, ce qui signifie que la forme de cette dernière ne dépend pas uniquement de l'expression à typer. De plus, le type des variables n'est pas déclaré : il faut donc inférer un environnement pour le programme à typer. Enfin, il faut découvrir quel type doit être utilisé pour chaque occurrence de symbole surchargé. Dans ce chapitre, nous montrerons tout d'abord un système de type équivalent au système présenté au chapitre 7, mais dont la forme des dérivations est déterminée par l'expression à typer. Nous en déduirons un algorithme de vérification des types avec inférence de type pour les variables, puis nous expliquerons comment gérer de manière efficace les symboles surchargés. Enfin nous présenterons des résultats expérimentaux sur la vérification de types.

8.1 Système de type dirigé par la syntaxe

Afin d'obtenir un système de types déterministe, on incorpore la règle *(Sub)* dans les règles *(Func)*, *(Atom)* et *(Head)*. On obtient alors le système décrit par les règles de la table 8.1.

L'*assignation de type aux symboles surchargés* d'une dérivation est la fonction qui à chaque occurrence d'un symbole surchargé $\mathtt{f}/n$ dans une expression, associe un schéma de type $\sigma \in \mathit{types}(\mathtt{f}/n)$. On peut remarquer que la forme des dérivations de ce système est imposée par la syntaxe de l'expression à typer. Si on fixe l'assignation de type aux symboles surchargés, on obtient alors un système déterministe.

On définit la notion de programme bien typé dans ce système de manière similaire à celle du système décrit dans le chapitre 7. Une clause H `:-` Q d'un prédicat $\mathtt{p}/n$ est *bien typée* si pour tout schéma de type de la forme $\forall \bar{\alpha}.\tau_1 \times \ldots \times \tau_n \rightarrow \mathrm{pred} \in \mathit{types}(\mathtt{p}/n)$, il existe un environnement Γ et une assignation de type

$$(Var^d) \quad \Gamma \vdash \mathtt{X} : \Gamma(\mathtt{X})$$

$$(Func^d) \quad \frac{\Gamma \vdash t_1 : \tau_1' \leq \rho(\tau_1) \ , \ldots , \ \Gamma \vdash t_n : \tau_n' \leq \rho(\tau_n)}{\Gamma \vdash \mathtt{f}(t_1, \ldots, t_n) : \rho(\tau)}$$

si $\forall \bar{\alpha}.\tau_1 \times \ldots \times \tau_n \rightarrow \tau \in \mathit{types}(\mathtt{f}/n)$ et ρ est une substitution de type

$$(Atom^d) \quad \frac{\Gamma \vdash t_1 : \tau_1' \leq \rho(\tau_1) \ , \ldots , \ \Gamma \vdash t_n : \tau_n' \leq \rho(\tau_n)}{\Gamma \vdash \mathtt{p}(t_1, \ldots, t_n) \ \mathit{Atom}}$$

si $\forall \bar{\alpha}.\tau_1 \times \ldots \times \tau_n \rightarrow \mathrm{pred} \in \mathit{types}(\mathtt{p}/n)$ et ρ est une substitution de type

$$(Head^d) \quad \frac{\Gamma \vdash t_1 : \tau_1' \leq \rho(\tau_1) \ , \ldots , \ \Gamma \vdash t_n : \tau_n' \leq \rho(\tau_n)}{\Gamma \vdash \mathtt{p}(t_1, \ldots, t_n) \ \mathit{Head}_{\mathtt{p}:\tau_1,\ldots,\tau_n}}$$

si $\forall \bar{\alpha}.\tau_1 \times \ldots \times \tau_n \rightarrow \mathrm{pred} \in \mathit{types}(\mathtt{p}/n)$ et ρ est un renommage de type

$$(Query^d) \quad \frac{\Gamma \vdash A_1 \ \mathit{Atom} \ , \ldots , \ \Gamma \vdash A_n \ \mathit{Atom}}{\Gamma \vdash A_1, \ldots, A_n \ \mathit{Query}}$$

$$(Clause^d) \quad \frac{\Gamma \vdash H \ \mathit{Head}_{\mathtt{p}:\tau_1,\ldots,\tau_n} \ , \ \Gamma \vdash Q \ \mathit{Query}}{\Gamma \vdash H \mathtt{:-} \ Q \ \mathit{Clause}_{\mathtt{p}:\tau_1,\ldots,\tau_n}}$$

TAB. 8.1 – Système de type déterministe avec surcharge pour CLP($\mathcal{X}$).

aux symboles surchargés tels que l'on peut dériver des règles de la table 8.1 le jugement $\Gamma \vdash H \texttt{:-}\ Q\ Clause_{\mathtt{p}:\tau_1,\ldots,\tau_n}$. Un programme est *bien typé* si toutes ses clauses sont bien typées. La propriété suivante indique que le système de types ainsi défini est équivalent au système décrit dans le chapitre 7 :

Propriété 8.1 *Un programme (resp. un but) est bien typé dans le système non déterministe défini au chapitre 7 si et seulement si il l'est dans le système déterministe défini par les règles de la table 8.1.*

Démonstration : Si un programme ou un but est typable dans le système déterministe, alors il l'est aussi dans le système non déterministe : il suffit en effet de remplacer les occurrences des règles *(Func^d)*, *(Atom^d)* et *(Head^d)* par les dérivations suivantes :

$$(Func)\frac{(Sub)\dfrac{\Gamma \vdash t_1 : \tau'_1, \tau'_1 \leq \rho(\tau_1)}{\Gamma \vdash t_1 : \rho(\tau)} \quad \cdots \quad (Sub)\dfrac{\Gamma \vdash t_n : \tau'_n, \tau'_n \leq \rho(\tau_n)}{\Gamma \vdash t_n : \rho(\tau)}}{\Gamma \vdash \mathtt{f}(t_1, \ldots, t_n) : \rho(\tau)}$$

$$(Atom)\frac{(Sub)\dfrac{\Gamma \vdash t_1 : \tau'_1, \tau'_1 \leq \rho(\tau_1)}{\Gamma \vdash t_1 : \rho(\tau)} \quad \cdots \quad (Sub)\dfrac{\Gamma \vdash t_n : \tau'_n, \tau'_n \leq \rho(\tau_n)}{\Gamma \vdash t_n : \rho(\tau)}}{\Gamma \vdash \mathtt{p}(t_1, \ldots, t_n)\ Atom}$$

$$(Head)\frac{(Sub)\dfrac{\Gamma \vdash t_1 : \tau'_1, \tau'_1 \leq \rho(\tau_1)}{\Gamma \vdash t_1 : \rho(\tau)} \quad \cdots \quad (Sub)\dfrac{\Gamma \vdash t_n : \tau'_n, \tau'_n \leq \rho(\tau_n)}{\Gamma \vdash t_n : \rho(\tau)}}{\Gamma \vdash \mathtt{p}(t_1, \ldots, t_n)\ Head_{\mathtt{p}:\tau_1,\ldots,\tau_n}}$$

On prouve à présent l'implication inverse, c'est-à-dire si un programme est typable dans le système non déterministe, ici noté $\vdash_{nd}$, alors il l'est dans le système déterministe, ici noté $\vdash_d$. Pour cela, on prouve par induction sur la dérivation que si $\Gamma \vdash_{nd} t : \tau$ alors il existe $\tau' \leq \tau$ tel que $\Gamma \vdash_d t : \tau'$. On raisonne par cas sur la dernière règle utilisée dans la dérivation de $\Gamma \vdash_{nd} t : \tau$:

- La règle *(Var)* est la même que la règle *(Var^d)*.
- La règle *(Sub)* nous permet de déduire $\Gamma \vdash_{nd} t : \tau$ à partir de $\Gamma \vdash_{nd} t : \tau''$ et $\tau'' \leq \tau$. Par induction, il existe τ' tel que $\Gamma \vdash t : \tau'$ avec $\tau' \leq \tau''$. Par transitivité on en déduit $\tau' \leq \tau$.
- Enfin, si la dernière règle utilisée est *(Func)*, on a $\Gamma \vdash_{nd} t_1 : \rho(\tau_1), \ldots, \Gamma \vdash_{nd} t_n : \rho(\tau_n)$. Par induction, il existe donc $\tau'_1 \leq \rho(\tau_1), \ldots, \tau'_n \leq \rho(\tau_n)$ tels que $\Gamma \vdash_d t_1 : \tau'_1, \ldots, \Gamma \vdash_d t_n : \tau'_n$. On peut donc appliquer la règle *(Func^d)* et obtenir $\Gamma \vdash_d \mathtt{f}(t_1, \ldots, t_n) : \rho(\tau)$.

Les règles *(Query^d)* et *(Clause^d)* sont identiques à *(Query)* et à *(Clause)*. En reproduisant le raisonnement sur la règle *(Func)*, on en déduit que si $\Gamma \vdash_{nd} A\ Atom$ (resp. $\Gamma \vdash_{nd} H\ Head_{\mathtt{p}:\tau_1,\ldots,\tau_n}$) alors $\Gamma \vdash_d A\ Atom$ (resp. $\Gamma \vdash_d H\ Head_{\mathtt{p}:\tau_1,\ldots,\tau_n}$). □

Si on fixe l'assignation de types aux symboles surchargés et l'environmment Γ, il est possible de calculer la substitution ρ nécessaire aux règles *(Func^d)*, *(Atom^d)* et *(Head^d)* en résolvant les contraintes de sous-typage collectées le long de la dérivation d'un jugement. Les paramètres apparaissant dans l'environnement, ainsi que ceux apparaissant dans le type des prédicats définis par les clauses typées ne doivent pas faire l'objet de ces substitutions. Il s'agit donc de limiter la portée des substitutions recherchées. A cette fin, on introduit de nouvelles constantes de types qui vont remplacer ces paramètres, ce qui permet d'éviter leur instanciation.

Considérons à présent le système Σ de contraintes de sous-typage imposé sur les types par les règles *(Func^d)*, *(Atom^d)* et *(Head^d)* au cours d'une dérivation. La taille, c'est-à-dire le nombre de symboles, du système Σ est $O(nvd)$, où v est la taille des déclarations de type pour les variables, n la taille du programme et d la taille des déclarations de type pour les symboles de fonction et de prédicats. Comme, à assignation de types aux symboles surchargés fixée, le système de types est déterministe, on a :

Propriété 8.2 *Un programme est bien typé si et seulement si il existe, pour chaque clause et pour chaque type $\tau_1, \ldots, \tau_n$ du prédicat $\mathtt{p}/n$ défini par cette clause, une assignation de types aux symboles surchargés telle que le système de contraintes de sous-typage associé à la dérivation de la clause respectant cette assignation est satisfiable.*

Démonstration : En effet, si la clause C est bien typée, alors pour chaque type $\tau_1, \ldots, \tau_n$ de $\mathtt{p}/n$, il existe un environnement Γ et une assignation de types aux symboles surchargés telle que le jugement $\Gamma \vdash C\ \mathit{Clause}_{\mathtt{p}:\tau_1,\ldots,\tau_n}$ soit obtenu par dérivation des règles de la table 8.1. Par définition, l'ensemble des contraintes accumulées lors de cette dérivation est satisfiable.

À l'inverse, supposons qu'il existe une assignation de types aux symboles surchargés telle que le système de contraintes de sous-typage associé à la dérivation de la clause C, avec le type $\tau_1, \ldots, \tau_n$, respectant cette assignation soit satisfiable. Ce système de contraintes a une solution ρ, qu'il suffit d'appliquer à la dérivation pour obtenir une dérivation des règles de la table 8.1 prouvant le jugement $\Gamma \vdash C\ \mathit{Clause}_{\mathtt{p}:\tau_1,\ldots,\tau_n}$. □

8.2 Algorithme de vérification des types

8.2.1 Inférence de types pour les variables

Le système décrit ci-dessus suppose un environnement Γ donné, c'est à dire des déclarations de type pour les variables. Si le programme est bien typé, il est cependant possible d'inférer pour chaque clause H :- Q et pour chaque schéma

de type $\forall\bar{\alpha}.\tau_1 \times \ldots \times \tau_n \rightarrow$ pred du prédicat $\mathtt{p}/n$ défini par cette clause, un environnement Γ tel que $\Gamma \vdash H \texttt{:-}\ Q\ Clause_{\mathtt{p}:\tau_1,\ldots,\tau_n}$. De même, pour un but B, il est possible d'inférer un environnement Γ tel que $\Gamma \vdash B\ Query$. Pour cela, le type des variables est remplacé par une inconnue de type. Puis on énumère les différentes assignations de type aux symboles surchargés possibles et on vérifie la satisfiabilité des contraintes de sous-typage associées à dérivation correspondante. L'environnement cherché est alors donné par la solution à ces contraintes.

8.2.2 Gestion de la surcharge

La vérification des types pour un programme ou un but nécessite de connaître le type des occurrences de chaque symbole de fonction surchargé. Si ces symboles sont nombreux, il est cependant très coûteux d'énumérer les assignations de type aux symboles surchargés puis de vérifier la satisfiabilité des contraintes de sous-typage. Afin de réduire le coût de l'énumération des assignations de type aux symboles surchargés, nous utilisons le principe Andorra. Ce principe, introduit pour la parallélisation de Prolog [15], consiste à retarder l'exécution des points de choix jusqu'au moment où tous les buts déterministes ont été exécutés. Il s'avère que cette stratégie de contrôle, au coeur de la programmation par contraintes, est suffisante pour gérer efficacement les symboles surchargés dans TCLP. Cette efficacité vient du fait que le contexte d'une expression contenant des symboles surchargés fournit souvent une information suffisante pour désambiguïser le type des symboles surchargés. Si le contexte n'est pas suffisant, on peut alors énumérer par backtracking les différentes possibilités.

8.2.3 Description de l'algorithme

L'algorithme de vérification des types avec surcharge et inférence du type des variables procède donc comme suit :

1. on associe à chaque variable de l'expression à typer une inconnue de type ;
2. on associe à chaque occurrence o d'un symbole $\mathtt{f}/n$ surchargé un type $\alpha_1^o \times \ldots \times \alpha_n^o \rightarrow \alpha^o$.
3. on collecte les inégalités de type le long de la dérivation, obtenant ainsi un système de contraintes C.
4. on énumère enfin les assignations de type aux symboles surchargés possibles comme suit :
   (a) Pour chaque occurrence o d'un symbole $\mathtt{f}/n$ surchargé, pour chaque $\tau_1 \times \ldots \times \tau_n \rightarrow \tau \in types(\mathtt{f}/n)$ on teste la satisfiabilité de $C[\alpha_i^o/\tau_i]$. Tous les types rendant le système insatisfiable sont éliminés des types possibles pour o. Si tous les types de o sont éliminés, c'est un échec.

(b) Si une occurrence d'un symbole n'a plus qu'un type possible, on ajoute les contraintes de sous-typage correspondantes à C.

(c) Les points (a) et (b) sont répétés jusqu'à obtention d'un point fixe.

(d) Une occurrence non résolue est alors choisie, et chaque type possible est alors essayé en itérant la procédure, à partir du point 4., par backtracking.

8.2.4 Implantation en CHR

Type des variables

L'association d'un type à chaque variable se fait en utilisant la contrainte CHR `::/2`. Pour chaque occurrence d'une variable `X`, on pose la contrainte `X::T`, où `T` est une variable de type représentant le type de `X`. La seule règle pour la contrainte `::/2` est :

```
X::T1 \ X::T2 <=> T1=T2.
```

Cette règle exprime simplement que la variable `X` possède un type unique. `T1` et `T2` correspondent à des variables de type et la contrainte `T1=T2` unifie ces deux variables. On bénéficie donc ici de la possibilité d'unifier les variables de type dans le solveur CHR pour le sous-typage décrit au chapitre 6.

Retardement des inégalités de sous-typage

L'algorithme décrit précédemment utilise l'instanciation de variables de type. Bien que le solveur de contraintes de sous-typage permette l'unification de variables de types, il ne permet pas d'unifier une variable de type avec un type construit. Ceci est dû à l'hypothèse selon laquelle les types sont manipulés dans le solveur sous forme de types plats. En particulier, que faire des variables de la forme γ_A et λ_A, introduites pour la factorisation, dans le cas où A contiendrait une variable instanciée ?

Afin de contourner cette difficulté, les contraintes CHR `:</2` correspondant aux inégalités de sous-typage sont suspendues lorsqu'elles sont appliquées à des variables qui ne sont pas des variables de types, c'est-à-dire qui ne sont pas associées à une contrainte `tclp__parameter/5`. Ces variables peuvent donc être unifiées plus tard avec soit des variables de type, soit avec des types construits. Une contrainte `:</2` est ensuite réveillée lorsque chacun de ses arguments est soit associé à une contrainte `tclp__parameter/5`, soit un type construit.

Lorsque, au point 2. de l'algorithme, on associe un type $\alpha_1^o \times \ldots \times \alpha_n^o \rightarrow \alpha^o$ à chaque occurrence de symbole de fonction, les variables α^o, α_i^o sont représentées par des variables non contraintes avec `tclp__parameter/5`. Les substitutions des points (a), (b) et (d) peuvent alors être faites directement sur les variables α^o, α_i^o.

Contraintes CHR pour la surcharge

La surcharge est gérée à travers deux contraintes CHR : `abstract_type`/4 et `multi_type`/2. La contrainte `abstract_type`/4 correspond à la description de l'occurrence de symbole surchargé considérée. Le premier argument est un identificateur correspondant à l'occurrence o et le second est une liste qui contient les α_i^o et α^o. Les deux derniers arguments correspondent respectivement à la description du symbole et à l'emplacement dans le texte du programme de l'occurrence, et sont utilisés lors de l'affichage des erreurs trouvées par TCLP. La contrainte `multi_type`/2 lie l'occurrence aux différents types possibles pour le symbole. Le premier argument est l'identificateur de l'occurrence, le second est une liste contenant les types possibles pour le symbole.

Le prédicat `apply_scheme`/5 effectue la substitution des α_i^o et α^o par les τ_i et τ. Le premier argument correspond aux τ_i et le second aux α_i^o. Le troisième et le quatrième argument contiennent des informations pour l'affichage d'erreur. Le dernier argument est `true` si l'erreur doit être affichée en cas d'échec de la substitution, `false` sinon.

Les deux règles suivantes gèrent le cas où tous les types d'une occurrence de symbole surchargé ont été supprimées et le cas où il ne reste qu'un type possible pour un symbole.

```
failure @ abstract_type(Id, _, FuncDesc, Location),
          multi_type(Id,[])
          <=> fail.

instanciate @ multi_type(Id,[ ConcreteScheme ]),
              abstract_type(Id, AbstractScheme, FuncDesc, Location)
              <=> apply_scheme(ConcreteScheme, AbstractScheme,
                               FuncDesc, Location, true).
```

8.3 Résultats expérimentaux

8.3.1 Détection d'erreurs de programmation

Nous présentons ici différentes erreurs détectées par le système de types.

Mauvaise utilisation d'un prédicat ou d'un symbole de fonction. Il s'agit de l'erreur la plus classique : un des arguments d'un symbole n'a pas le bon type.

Exemple 8.1 : Considérons la clause suivante :

```
p(X,Y) :- Y is (3.5 // X).
```

avec les déclarations

```
:- typeof is(int,int_expr) is pred.
:- typeof //(int_expr,int_expr) is int_expr.
```

Le vérificateur de types produit l'erreur :

```
Incompatible type : 3.5/0 has type float but is required to
have type int_expr
```

Cet autre exemple montre une erreur détectée lors de l'utilisation d'un prédicat externe, défini par exemple à travers une interface Prolog / C :

Exemple 8.2 : Considérons la déclaration suivante d'un prédicat `p`/1 défini en C en utilisant l'interface SICStus/C[1] :

```
foreign(p, p(+integer)).
```

Le typer déduit de cette déclaration que le prédicat `p`/1 a le type int → pred. La directive

```
:- p(3.14).
```

conduit à l'erreur :

```
Incompatible type : 3.14/0 has type float but is required to have
type int
```

Inversion d'arguments. Cette erreur est un cas particulier de la précédente qui apparaît lorsque le programmeur inverse la position de deux arguments, par exemple lors de l'appel d'un prédicat.

Exemple 8.3 : Considérons la clause suivante dans laquelle les arguments du prédicat `length`/2 on été inversés :

```
p(L1,L2,N) :- append(L1,L2,L3), length(N,L3).
```

avec les déclarations :

```
:- typeof append(list(A),list(A),list(A)) is pred.
:- typeof length(list(A),int) is pred.
```

Le vérificateur de types produit l'erreur suivante :

[1] Le + correspond ici au mode du prédicat : l'argument doit être instancié et être un entier

```
Incompatible types for L3 : list(top) and int
```

Cet exemple montre l'utilité d'utiliser un quasi-treillis de types sans type $\perp$ plus petit que tous les autres types. Un tel type pourrait en effet toujours être inféré comme type pour n'importe quelle variable du programme.

Exemple 8.4 : Ce type d'erreur à été trouvé dans une bibliothèque SICStus, `db.pl` :

```
db_open(DBName, Mode, SpecList, EnvRef, DBRef) :-
      ErrGoal = db_open(EnvRef, DBName, Mode, SpecList, DBRef),
      db_open_int(EnvRef, DBName, Mode, SpecList, DBRef, ErrGoal).
```

avec les déclarations :

```
:- typeof db_open(atom,db_mode,list(db_spec),db_env_ref,db_ref).
:- typeof db_open_int(db_env_ref,atom,db_mode,list(db_spec),
                      db_ref,goal).
```

Le vérficateur de types a alors produit l'erreur :

```
Incompatible types for EnvRef : atom and db_env_ref
```

Le term unifié avec `ErrGoal` est ici mal formé, la variable `EnvRef` étant utilisée comme premier argument et non en tant que quatrième argument. $\diamond$

Définition de prédicat incompatible avec sa déclaration de type. Cette erreur est détectée lors du typage de la tête de clause. Dans les deux exemples qui suivent, on considère la déclaration le prédicat `p/1` :

```
:- typeof p(int) is pred.
```

Exemple 8.5 : La clause : `p(1.2).` provoque l'erreur :

```
Incompatible type : 1.2/0 has type float but is required to have
type int
```

Exemple 8.6 : La clause et la déclaration :

```
p(X) :- length(X,2).
:- typeof length(list(A),int) is pred.
```

provoquent l'erreur :

```
Incompatible types for X : list(top) and int
```

L'exemple suivant correspond au cas particulier de violation de la règle de généricité des définitions, c'est à dire que le type de la tête de clause est un sous-type d'un instance et non d'un renommage du type déclaré pour le prédicat :

Exemple 8.7: Considérons la déclaration de type et la clause non générique suivantes du prédicat `p/2` :

```
:- typeof p(list(A)) is pred.
p([1]).
```

Le vérificateur de types produit l'erreur :

```
Incompatible type : 1/0 has type byte but is required to have type A
```

8.3.2 Performances

Le tableau 8.2 indique les performances de TCLP sur 17 bibliothèques SICStus Prolog, ainsi que sur le code de TCLP lui-même, sur la dernière ligne. La première colonne indique la bibliothèque et la deuxième, sa taille en nombre de lignes de code (hors commentaires et lignes blanches). La troisième colonne indique le temps total d'exécution, la quatrième le temps de calcul des structures de données et des clauses utilisées pour représenter l'ordre de sous-typage. La cinquième colonne indique le temps de vérification des types. La dernière colonne indique la partie du temps de vérification de types correspondant à la désambiguïsation des symboles surchargés, le pourcentage étant calculé par rapport à la cinquième colonne.

Les temps totaux montrent que TCLP peut être utilisé en pratique, même avec des programmes de grande taille, puisqu'il suffit de 49.68 s pour vérifier les types de la bibliothèque `clpr` qui représente 4286 lignes de code, et de 54.91 s pour vérifier les 4124 lignes de code de TCLP. Ces temps peuvent cependant paraître élévés si on les compare aux temps de typage d'autres langages tels que C ou ML en pratique, mais le système de types que nous utilisons ici est plus complexe, cette complexité étant nécessaire au typage de programmes réels, écrit dans des dialectes de programmation logique avec contraintes non typés au départ.

La différence entre le temps total et le temps de vérification des types est due principalement au calcul des structures d'ordre sur les types, qui dure entre 1.29 s et 4.78 s. Si l'impact de ce calcul est faible pour les grosses bibliothèques comme `clpr` ou `ugraphs`, il est important pour de petites bibliothèques telles que `fastrw` ou `queues` dont le temps de vérification est inférieur à la seconde (respectivement 0.20 s et 0.44 s). Le temps de résolution des symboles surchargés représente entre 17 %, pour `charsio`, et 61 %, pour `clpr`, du temps de vérification des types selon les fichiers. En particulier, la bibliothèque `clpr` surcharge de nombreux symboles

Bibliothèque	Taille	Total	Structure	Vérification	Surcharge	
arrays	96 l	2.52 s	1.40 s	0.80 s	0.18 s	22 %
assoc	209 l	3.89 s	1.30 s	2.16 s	0.52 s	24 %
atts	216 l	3.72 s	1.45 s	1.92 s	0.75 s	39 %
bdb	487 l	6.08 s	2.32 s	3.14 s	0.84 s	26 %
charsio	97 l	1.99 s	1.50 s	0.40 s	0.07 s	17 %
clpb	584 l	11.60 s	1.36 s	10.04 s	4.08 s	41 %
clpr	4286 l	49.68 s	1.63 s	47.05 s	29.10 s	61 %
fastrw	54 l	1.83 s	1.30 s	0.20 s	0.05 s	25 %
heaps	137 l	3.58 s	1.30 s	1.87 s	0.49 s	26 %
jasper	179 l	3.21 s	1.79 s	0.98 s	0.32 s	32 %
lists	186 l	3.44 s	1.24 s	1.86 s	0.96 s	51 %
ordsets	208 l	3.92 s	1.23 s	2.35 s	0.89 s	37 %
queues	47 l	2.14 s	1.31 s	0.44 s	0.12 s	27 %
sockets	211 l	4.02 s	1.61 s	1.83 s	0.82 s	44 %
terms	97 l	2.90 s	1.21 s	1.32 s	0.44 s	33 %
trees	72 l	2.47 s	1.29 s	0.79 s	0.27 s	34 %
ugraphs	542 l	16.28 s	1.48 s	14.17 s	7.39 s	52 %
tclp	4124 l	54.91 s	4.78 s	49.35 s	27.65 s	56 %

TAB. 8.2 – Performances de vérification des types

arithmétiques, ainsi que la virgule et le point-virgule qui sont utilisés pour écrire les buts CLP($\mathcal{R}$) entre accolades. Malgré cela, la résolution de la surcharge ne prend que 29.1 s, ce qui montre l'efficacité de la procédure de résolution des symboles surchargés, le problème étant potentiellement exponentiel.

Chapitre 9

Inférence de type des prédicats

Lors de la vérification des types, le typeur infère le type des variables, évitant ainsi au programmeur la tâche de déclaration de type pour ces variables. Il lui faut cependant déclarer le type des prédicats. Nous proposons ici une méthode pour inférer le type des prédicats à partir de leur définition, afin de soulager la tâche du programmeur. Inférer le type des prédicats permet ainsi un prototypage plus rapide : le programmeur peut modifier de grandes parties de code sans se soucier d'avoir à effectuer les changements de déclarations de types, potentiellement nombreux, qui correspondent à ces modifications. L'inférence de type des prédicats permet également de modifier plus facilement les structures de données correspondant à un type abstrait. Considérons par exemple une bibliothèque pour gérer des ensembles d'entiers, avec les opération d'ajout et de suppression d'éléments, d'union, d'intersection etc... Les types des prédicats publics de la bibliothèque reflètent le fait qu'elle manipule des ensemble d'entiers, mais pas les structures de données qui sont utilisées dans la bibliothèque pour gérer ces ensembles. Dans une première version cette bibliothèque peut être implantée à l'aide de listes d'entiers. Un peu plus tard, elle est réimplantée avec des arbres binaires. Le type des prédicats internes à la bibliothèque va changer, mais on veut que le type des prédicats publics reste inchangé. On va déclarer le type des prédicats publics et utiliser l'inférence de type pour les prédicats internes.

Le problème de l'inférence de type en présence de récursion polymorphe étant indécidable, nous utilisons une inférence type avec récursion monomorphe. La première idée venant à l'esprit pour inférer un type pour les prédicats, consiste à remplacer, dans la dérivation de typage, le type des arguments du prédicat par des variables de types sur lesquelles on accumule des contraintes de sous-typage. Pour une assignation de type aux symboles surchargés donnée, la résolution de ces contraintes de sous-typage fournit alors un type pour chaque argument du prédicat. Cependant, dans notre système de type covariants, les inégalités ainsi

introduites sont uniquement de la forme $\tau \leq \alpha_i$ où α_i est la variable de type correspondant au i-ème argument du prédicat dont on cherche le type, c'est-à-dire que ces variables sont exclusivement bornées inférieurement. Or les structures de type que nous utilisons pour typer les programmes logiques avec contraintes contiennent le type term de méta-programmation, plus grand que tous les autres types. Cela signifie que $\text{term} \times \ldots \times \text{term} \rightarrow \text{pred}$ est toujours un type valable pour les prédicats définis dans un programme. Ce type n'est cependant pas satisfaisant en général : il ne restreint pas suffisamment l'usage des prédicats et n'est pas assez informatif. Par exemple, on aimerait que le prédicat `append`/3 de concaténation de listes ait un type en rapport avec les listes[1].

Il faut donc choisir, parmi les solutions au système de contraintes de sous-typage, celles qui reflètent le mieux l'utilisation de ce prédicat. Cela nous conduit à utiliser une inférence de type heuristique [22]. Pour choisir le type d'un prédicat, nous nous appuyons sur le fait qu'il devrait refléter au mieux celui des arguments de la tête des clauses qui le définissent. Si un de ces arguments est une variable, le type du prédicat doit refléter l'utilisation de cette variable, c'est à dire son type inféré. Si un argument est un terme partiellement instancié, il correspond à un filtrage sur une certaine structure de donnée et le type du prédicat devrait refléter le type de cette structure. Cette dernière remarque nous rapproche des systèmes de types descriptifs, dans lesquels le type inféré pour un prédicat est une approximation de l'ensemble des succès de ce prédicat. Cela n'est cependant pas notre objectif ici : nous cherchons un ensemble de termes (décrit par un type) sur lequel le prédicat est "légitimement" applicable et non un ensemble de termes sur lequel le prédicat obtient des succès. Un exemple typique est celui du prédicat `append`/3 :

Exemple 9.1 : Soit la définition suivante du prédicat `append`/3 :

```
append([],R,R).
append([X|L],L2,[X|R]) :- append(L,L2,R).
```

Un système de type descriptif inférera un type de la forme $\forall\alpha.\text{list}(\alpha) \times \text{term} \times \text{term} \rightarrow \text{pred}$, car `append([],1,1)` admet un succès. Cependant l'utilisation de ce prédicat devrait plutôt être restreinte à des listes, par exemple avec le type $\forall\alpha.\text{list}(\alpha) \times \text{list}(\alpha) \times \text{list}(\alpha) \rightarrow \text{pred}$. ◇

L'exemple suivant illustre également la différence entre systèmes prescriptifs et systèmes descriptifs, à travers une itération par backtracking. Cette technique, souvent utilisée en programmation logique avec contraintes, consiste à échouer volontairement afin de libérer de la mémoire.

[1] ceci, même si le but `append([],X,X)` a un succès avec un argument `X` de type quelconque, et contrairement à ce qui serait inféré dans un système descriptif : voir l'exemple 9.1.

Exemple 9.2 : Considérons le programme suivant qui affiche des termes lus sur l'entrée standard.

```
ecrit(X) :- write(X), nl, fail.
:- repeat, read(X), ( X == end_of_file ; ecrit(X) ).
```

Le prédicat `repeat`/0, créée un nouveau point de choix à chaque fois qu'il est appelé, c'est-à-dire que la fin du but, à partir de l'appel à `read`/1 sera réexécutée tant que le but n'aura pas de succès.

Ici, un système de type descriptif inférerait un type vide pour le prédicat `ecrit`/1, puisque celui-ci n'a aucun succès. Il est cependant utilisable avec n'importe quel terme ce qui correspond au type term → pred. ◇

9.1 Inférence heuristique

Pour procéder à l'inférence de type, on plonge d'abord le quasi-treillis des types dans un treillis en ajoutant les constructeurs $\top$ et $\bot$, ce qui est déjà fait dans le solveur de contraintes de sous-typage (*c.f.* chapitre 6). La présence de $\top$ comme borne supérieure signifie l'absence de réelle borne supérieure dans le système de contraintes. De même, la présence de $\bot$ comme borne inférieure signifie l'absence de réelle borne inférieure. On effectue ensuite le calcul du type heuristique des prédicats à inférer, en procédant par composantes connexes du graphe d'appel.

9.1.1 Calcul du type heuristique

Le calcul du type heuristique se déroule en trois étapes. On collecte tout d'abord les inégalités de sous-typage correspondant aux dérivations des clauses des prédicats à inférer. Puis, pour chacun de ces prédicats, on calcul un premier type monomorphe heuristique. Enfin on généralise ce premier type heuristique pour obtenir le type inféré pour chaque prédicat. Nous détaillons à présent ces trois étapes.

Collecte des inégalités de sous-typage. L'inférence débute par la collecte des inégalités de sous-typage le long des dérivations de typage des clauses de tous les prédicats appartenant à une même partie connexe du graphe d'appel. Dans ces dérivations, les types des arguments des prédicats inférés sont remplacés par des variables. Les symboles surchargés sont alors résolus simultanément pour toutes ces clauses. Il peut y avoir plusieurs solutions à cette surcharge. Dans l'implantation, seule la première est considérée, ceci pour éviter une explosion combinatoire du nombre de symboles surchargés[2]. On obtient alors un système

[2]Malgré cette restriction, le pourcentage des types inférés correspondant au type entré à la main reste bon (voir page 130).

de contraintes de sous-typage C, qui servira de base au calcul heuristique du type des prédicats.

Calcul d'un premier type monomorphe. Un premier type est calculé récursivement pour chaque argument i du prédicat à inférer, à partir du système C qui sera augmenté au fur et à mesure du calcul. Dans la description de ce calcul, $\tilde{\tau}$ désigne le type calculé et α la variable représentant le type à inférer, c'est-à-dire, au premier appel, celle remplaçant, dans les dérivations, le type de l'argument i du prédicat inféré.

1. Soit $\underline{\kappa}$ le constructeur de la borne inférieure de α dans le système de contraintes C courant. $\underline{\kappa}$ correspond aux termes (éventuellement partiellement) instanciés qui apparaissent en tête de clause. Cette borne est simplement obtenue par simplification des contraintes accumulées lors de la dérivation de typage.
2. α est unifiée avec toutes les variables qui lui sont inférieures dans C. Cette unification est heuristique car elle ne correspond à rien formellement dans le système de type. Elle permet cependant de répercuter sur α les contraintes sur le type des variables qui apparaissent dans la tête de clause.
   - Si le système de contraintes C' résultant est satisfiable et ne crée pas de cycle sur α, alors on note $\overline{\kappa}$ le constructeur de la borne supérieure de α dans C', et C est remplacé par C'.
   - sinon, on ajoute les contraintes term $\leq \alpha, \alpha \leq$ term dans C. $\underline{\kappa}$ devient alors term et on pose également $\overline{\kappa} =$ term.
3. On détermine ensuite le constructeur de tête de $\tilde{\tau}$, noté $\tilde{\kappa}$, selon les valeurs de $\underline{\kappa}$ et $\overline{\kappa}$:
   - Si $\overline{\kappa} \neq \top$ alors $\tilde{\kappa} = \overline{\kappa}$;
   - sinon, si $\underline{\kappa} \neq \bot$ alors $\tilde{\kappa} = \underline{\kappa}$;
   - sinon $\tilde{\kappa} = \alpha$.

   Dans le dernier cas, α est potentiellement généralisable.
4. Si $\tilde{\tau}$ n'est pas une variable, on construit ses arguments récursivement[3] : pour chaque étiquette $l \in ar(\tilde{\kappa})$, on détermine $\tilde{\tau}/l$ à partir des variables $\overline{\tau}/l$ et $\underline{\tau_i}/l$ en les unifiant si elles existent toutes deux.

Pour chaque argument i, la contrainte $\alpha_i \leq \tilde{\tau}_i$ est alors ajoutée à C. Ceci permet d'assurer que le type heuristique sera un type correct pour le prédicat inféré. En cas d'échec, le type de l'argument i devient le type term.

Généralisation du type monomorphe. Le type heuristique τ_i de l'argument i est déduit du type $\tilde{\tau}_i$ ainsi obtenu, en généralisant les variables qui apparaissent

[3] la terminaison est ici obtenue par l'élimination des cycles au point 2.

dans ce dernier. Pour ce faire, ces variables sont instanciées par des constantes incomparables aux autres types et la satisfiabilité du système de contraintes C résultant est testée. Si C est satisfiable alors la variable est considérée comme correctement généralisée, sinon elle est remplacée par le type term. Il est important de noter que l'instanciation d'une variable avec une telle constante peut provoquer l'instanciation d'autres variables dans les types des arguments du prédicat inféré. C'est par exemple ce qui se passe avec `append`/3 où l'instanciation d'une variable d'un des arguments avec une constante conduit à celle de toutes les autres variables à la même constante.

9.1.2 Exemples de types inférés

Nous illustrons à présent le fonctionnement de cette heuristique à partir des quelques exemples suivants :

Exemple 9.3 : Considérons le prédicat `p`/1 défini par :

```
p(1).
```

Comme `1` a le type int, $\underline{\tau_1} = \text{int}$. Il n'y a pas de variables donc $\overline{\tau_1} = \top$. On obtient donc $\tau_1 = \tilde{\tau_1} = \text{int}$ et le type inféré pour `p`/1 est int $\rightarrow$ pred. $\diamond$

Exemple 9.4 : Soit `q`/2 le prédicat défini par :

```
q(X,Y) :- X is 2 + Y.
```

Après résolution de la surcharge, le type de `is`/2 est int $\times$ int_expr $\rightarrow$ pred et celui de `+`/2 est int_expr $\times$ int_expr $\rightarrow$ int_expr[4]. Le type de `X` est donc int et celui de `Y` est int_expr. Comme les deux arguments de `q`/2 en tête de clause sont des variables, $\underline{\tau_1} = \bot$ et $\underline{\tau_2} = \bot$. En utilisant les types de `X` et `Y`, on obtient $\overline{\tau}_1 = \text{int}$ et $\overline{\tau}_2 = \text{int\_expr}$. On a donc $\tilde{\tau_1} = \text{int}$ et $\tilde{\tau_2} = \text{int\_expr}$. On obtient donc le type int $\times$ int_expr $\rightarrow$ pred pour `q`/2. $\diamond$

Exemple 9.5 : Soit `r`/1 le prédicat défini par :

```
r(1).
r(X) :- X < 0.
```

Après résolution de la surcharge, le type de `<`/2 est int_expr $\times$ int_expr $\rightarrow$ pred. Si on note $\alpha_{\texttt{X}}$ la variable représentant le type de `X` et α_1 la variable représentant le type de l'argument de `r`/1, on obtient $\alpha_{\texttt{X}} \leq \alpha_1$, int $\leq \alpha_1$ et $\alpha_X \leq$ int_expr. Après unification de α_1 avec α_X, on obtient $\underline{\kappa}_1 = \text{int}$ et $\overline{\kappa}_1 = \text{int\_expr}$, d'où $\tilde{\kappa}_1 = \text{int\_expr}$, ce qui donne pour `r`/1 le type int_expr $\rightarrow$ pred. $\diamond$

[4]La surcharge aurait également pu être résolue avec les types float $\times$ float_expr $\rightarrow$ pred pour `is`/2 et int_expr $\times$ float_expr $\rightarrow$ float_expr pour `+`/2, ce qui aurait mené à un type inféré différent, à savoir float $\times$ float_expr $\rightarrow$ pred.

Exemple 9.6 : Considérons le prédicat `append`/3 défini classiquement par :

```
append([], A, A).
append([ X | L1 ], L2, [ X | R ]) :- append(L1, L2, R).
```

La collecte des contraintes de sous-typage donne les types suivants pour les variables : $\alpha_{\mathtt{A}} \leq \alpha_2$, $\alpha_{\mathtt{A}} \leq \alpha_3$, $\text{list}(\beta) \leq \alpha_1$, $\alpha_{\mathtt{X}} \leq \beta$, $\alpha_{\mathtt{L1}} \leq \text{list}(\beta)$, $\alpha_{\mathtt{L2}} \leq \alpha_2$, $\text{list}(\beta') \leq \alpha_3$, $\alpha_{\mathtt{X}} \leq \beta'$, $\alpha_{\mathtt{R}} \leq \text{list}(\beta')$, $\alpha_{\mathtt{L1}} \leq \alpha_1$, $\alpha_{\mathtt{R}} \leq \alpha_3$.

On a $\underline{\kappa}_1 = \text{list}$ et $\overline{\kappa}_1 = \text{list}$, après unification de α_1 avec $\alpha_{\mathtt{L1}}$. On obtient donc $\tilde{\tau}_1 = \text{list}(\alpha'_1)$, et on détermine α'_1 à partir de β. On a $\underline{\kappa}'_1 = \bot$ et, après unification de β avec $\alpha_{\mathtt{X}}$, $\overline{\kappa}'_1 = \top$. On obtient donc $\alpha'_1 = \beta$, d'où $\tilde{\tau}_1 = \text{list}(\beta)$.

On a $\underline{\kappa}_2 = \bot$ et, après unification de α_2 avec $\alpha_{\mathtt{A}}$ et $\alpha_{\mathtt{L2}}$, on obtient $\overline{\kappa}_2 = \top$, d'où $\tilde{\tau}_2 = \alpha_2$.

On a $\underline{\kappa}_3 = \text{list}(\beta')$ et, après unification de α_3 avec $\alpha_{\mathtt{A}}$ et $\alpha_{\mathtt{R}}$, $\overline{\kappa}_3 = \text{list}(\beta')$. On obtient donc $\tilde{\tau}_3 = \text{list}(\alpha'_3)$, et on détermine α'_3 à partir de β'. On a $\underline{\kappa}'_3 = \bot$ et, après unification de β avec $\alpha_{\mathtt{X}} = \beta$, $\overline{\kappa}'_3 = \top$. On obtient donc $\alpha'_3 = \beta$, d'où $\tilde{\tau}_3 = \text{list}(\beta)$. On peut noter au passage que comme α_3 a été unifiée avec $\alpha_{\mathtt{A}} = \alpha_2$, on a $\alpha_3 = \alpha_2$, ce qui aura un impact sur le type inféré à la fin.

Après ajout des contraintes $\alpha_i \leq \tilde{\tau}_i$, on obtient les trois bornes supérieures $\tau_1 = \text{list}(\beta)$, $\tau_2 = \text{list}(\beta)$ et $\tau_3 = \text{list}(\beta)$. Ce qui donne, après généralisation de β, le type $\forall\beta.\text{list}(\beta) \times \text{list}(\beta) \times \text{list}(\beta) \rightarrow \text{pred}$ pour `append`/3. ◇

9.2 Résultats expérimentaux

Rapidité du typage

Le tableau 9.1 compare les temps de typage des 17 bibliothèques SICStus Prolog donnés dans le tableau 8.2, page 121, avec le temps pour l'inférence de type pour les prédicats. Le nombre de prédicats inférés par rapport au nombre total de prédicats est indiqué entre parenthèses. Pour chaque bibliothèque, deux séries de temps sont indiquées. La première, "vérification simple", concerne le typage sans inférence de type pour les prédicats, alors que la deuxième concerne le typage avec inférence du type des prédicats. Pour chacun de ces deux cas, trois temps sont donnés : le temps total, le temps du typage et, la part de ce temps de typage correspondant à la résolution de la surcharge.

Comme on peut s'y attendre, les temps de typage avec inférence de type pour les prédicats sont plus importants. Cela est dû au fait que dans le cas de la vérification simple l'on considère chaque clause indépendamment, alors que lors de l'inférence de type pour les prédicats, les clauses sont regroupées par composantes connexes du graphe d'appel. Ceci conduit à gérer simultanément un plus grand ensemble de contraintes de sous-typage et à une combinatoire des symboles surchargés plus importante. De plus, il y a un temps supplémentaire qui

Bibliothèque		Vérification simple			Inférence pour les prédicats		
	(# inf./# tot.)	Total	Typage	(surch.)	Total	Typage	(surch.)
`arrays`	(6/13)	2.52 s	0.80 s	0.18 s	2.73 s	1.00 s	0.19 s
`assoc`	(13/31)	3.89 s	2.16 s	0.52 s	5.61 s	3.88 s	0.87 s
`atts`	(18/20)	3.72 s	1.92 s	0.75 s	5.04 s	3.26 s	1.35 s
`bdb`	(55/79)	6.08 s	3.14 s	0.84 s	7.06 s	4.19 s	1.07 s
`charsio`	(2/17)	1.99 s	0.40 s	0.07 s	2.00 s	0.43 s	0.09 s
`clpb`	(51/58)	11.60 s	10.04 s	4.08 s	40.64 s	39.06 s	25.31 s
`clpr`	(396/419)	49.68 s	47.05 s	29.10 s	145.76 s	142.49 s	97.60 s
`fastrw`	(4/12)	1.83 s	0.20 s	0.05 s	1.98 s	0.32 s	0.12 s
`heaps`	(11/21)	3.58 s	1.87 s	0.49 s	7.24 s	5.50 s	1.51 s
`jasper`	(10/24)	3.21 s	0.98 s	0.32 s	3.52 s	1.36 s	0.48 s
`lists`	(10/39)	3.44 s	1.86 s	0.96 s	4.24 s	2.63 s	1.23 s
`ordsets`	(17/35)	3.92 s	2.35 s	0.89 s	8.92 s	7.33 s	3.64 s
`queues`	(3/12)	2.14 s	0.44 s	0.12 s	2.26 s	0.55 s	0.17 s
`sockets`	(12/27)	4.02 s	1.83 s	0.82 s	4.15 s	2.12 s	0.77 s
`terms`	(4/14)	2.90 s	1.32 s	0.44 s	3.31 s	1.72 s	0.54 s
`trees`	(8/16)	2.47 s	0.79 s	0.27 s	2.89 s	1.17 s	0.32 s
`ugraphs`	(62/90)	16.28 s	14.17 s	7.39 s	34.04 s	31.97 s	11.20 s
`tclp`	(368/494)	54.91 s	49.35 s	27.65 s	830.09 s	824.73 s	751.82 s

TAB. 9.1 – Surcoût de l'inférence des types pour les prédicats

correspond à l'application de l'heuristique. Cet allongement du temps de typage est particulièrement important pour l'inférence du type des prédicats de `tclp`, puisque l'on passe de 54.91 s à 830.09 s, soit un temps 15 fois plus long. Cet impact sur les performances est dû, dans ce cas particulier, à une composante connexe très large, comprenant 54 prédicats. Le temps d'inférence de cette composante est de 741.57 s, dont 699.24 s pour la résolution des symboles surchargés. En revanche, les bibliothèques SICStus ne présentent pas de tel cas. Ainsi, en moyenne, le temps de typage est multiplié par 1.88. Toujours en moyenne, le temps nécessaire à lever l'ambiguïté des symboles surchargés est multiplié par 2.08. On pourrait s'attendre à une augmentation plus spectaculaire des temps de typage, en particulier des temps de résolution de la surcharge, exception faite du cas de `tclp`. Ces augmentations restent raisonnables car, même si elles sont traitées en même temps, les clauses d'une même composante connexe restent assez indépendantes : les liens avec les autres clauses n'apparaissant que dans les têtes de clauses et au niveau des appels récursifs. Les différences de temps entre le typage avec et sans inférence du type des prédicats varient cependant grandement selon les bibliothèques. Certaines bibliothèques verront leur temps de typage multiplié par 3 comme `clpr`, `heaps` ou `ordsets`, voir même 4 dans le cas de `clpb`, alors que d'autres auront des temps assez proches, comme `bdb`

dont le rapport entre les temps de typage est de 1.33. De même la proportion du temps de résolution de la surcharge dans le temps de typage varie selon que l'inférence de type des prédicats est, ou n'est pas, effectuée. Ainsi, la bibliothèque `ugraphs` voit la proportion du temps de typage correspondant à la résolution de la surcharge passer de 52% en vérification simple à 35% en inférence de type pour les prédicats, alors qu'au contraire pour la bibliothèque `ordsets`, il passe de 38% à 50%.

Précision de l'inférence

Le tableau 9.2 compare la précision de l'inférence de type selon que l'on utilise ou pas la surcharge pour les symboles de prédicats. Par précision, on entend le pourcentage de types de prédicats inférés qui correspond à la déclaration de type qui aurait été donnée par l'utilisateur. Les bibliothèques testées sont les mêmes que pour le tableau 9.1, à l'exception de `clpb`, `clpr` et `tclp`.

Le typage des bibliothèques sans la surcharge a nécessité une extension de la structure des types afin d'introduire des sous-types pour représenter le fait que certain symboles de fonction soient utilisables dans des contextes différents, comme expliqué dans la section 7.4.2. De plus certains programmes ont été transformés : par exemple l'opérateur `-/2` à été remplacé par `--/2` aux endroits ou il était utilisé pour le codage des paires.

Bibliothèque	% types corrects surcharge		Bibliothèque	% types corrects surcharge	
	sans	avec		sans	avec
`arrays`	23%	68%	`lists`	98%	98%
`assoc`	68%	91%	`ordsets`	97%	97%
`atts`	64%	91%	`queues`	75%	96%
`bdb`	64%	66%	`sockets`	68%	92%
`charsio`	33%	74%	`terms`	77%	77%
`fastrw`	100%	100%	`trees`	31%	75%
`heaps`	71%	97%	`ugraphs`	67%	67%
`jasper`	84%	84%			

TAB. 9.2 – Précision de l'inférence de type pour les prédicats

On constate, lors du passage du typage sans surcharge au typage avec surcharge, une très nette amélioration des pourcentages de types inférés correspondant à ce qui aurait été déclaré par l'utilisateur. En moyenne, la précision passe de 68% à 85%, le taux minimal passant de 23% pour `arrays` à 66% pour `bdb`. Il y a plusieurs explications à ce phénomène. La première est la simplification de la structure des types, puisque qu'un certain nombre de sous-types communs

artificiellement ajoutés disparaissent de la structure utilisée pour le typage avec surcharge. La seconde est liée à la surcharge des expressions arithmétiques en particulier. Sans surcharge, le type des opération arithmétiques entières, int_expr est un sous-type des opération arithmétiques flottantes, float_expr. Par exemple, dans ce contexte, le type de `+`/2 est float_expr × float_expr → float_expr. Cela conduit à inférer le type float_expr pour de nombreuses expressions arithmétiques là où int_expr est attendu. C'est en particulier le cas pour le prédicat `length`/2 qui calcule la longueur d'une liste et pour de nombreux prédicats de la bibliothèques arrays.

Comme énoncé précédemment, le taux de types correctement inférés atteint en moyenne 85%, ce qui montre que l'heuristique utilisée est plutôt bonne. Les types incorrects le sont en général lorsqu'un argument se voit donner le type term alors que l'utilisateur attendait un type plus précis. Ceci est dû à l'utilisation de la contrainte d'égalité, comme illustré dans l'exemple suivant :

Exemple 9.7 : Considérons le prédicat `p`/1 défini par la clause suivante :

```
p(X) :- X = 2.
```

Le type attendu pour `p`/1 est probablement[5] : int → pred. Cependant le type inféré sera le type term, car la seule contrainte sur le type $\tau_{\texttt{X}}$ de `X` est qu'il doit avoir un supertype commun avec int, c'est-à-dire il doit être un sous-type de term. ◇

Une autre raison pour obtenir un type incorrect est que lors de l'énumération des types des symboles surchargés, seule la première combinaison de types permettant le typage des clauses est considérée pour l'inférence[6]. Il est alors possible que ce choix ne corresponde pas au type voulu par l'utilisateur :

Exemple 9.8 : Supposons que le symbole `-`/2 soit déclaré avec les deux types $\alpha \times \beta \rightarrow \text{pair}(\alpha, \beta)$ et int_expr × int_expr → int_expr. Considérons le prédicat `p`/1 défini par :

```
p( X - Y ).
```

Ce prédicat peut avoir à la fois les types $\text{pair}(\alpha, \beta) \rightarrow$ pred et int_expr → pred, or l'inférence de type n'en donnera qu'un seul, qui n'est pas forcément celui auquel a pensé le programmeur. ◇

Dans les bibliothèques testées, ce cas ne s'est cependant pas souvent produit, l'heuristique consistant à essayer en priorité le dernier type déclaré pour un symbole s'avérant ici efficace.

[5] En fait, tout type au dessus de int, comme par exemple int_expr, est également un candidat possible.

[6] Ceci, afin de limiter l'explosion combinatoire de la résolution des symboles surchargés pour les clauses typées ultérieurement.

Chapitre 10

Conclusion

Notre travail avait comme point de départ l'implantation du système de types de Fages et Paltrinieri [23] avec polymorphisme paramétrique et sous-typage pour les langages logiques avec contraintes.

Cette implantation, nommée TCLP, a conduit, d'une part au développement d'algorithmes de résolution des contraintes de sous-typage dans des structures de quasi-treillis, et d'autre part à l'intégration de la surcharge au système de types, ainsi qu'à l'élaboration d'une heuristique d'inférence de types pour les prédicats. TCLP a également permis d'expérimenter le système de types sur des programmes issus d'applications réelles, comme les bibliothèques de SICStus Prolog, ou encore TCLP lui-même.

Dans une première partie, nous avons étudié les contraintes de sous-typage non structurel non homogène dans les quasi-treillis. Après avoir présenté le formalisme des types, issu de Pottier [57], nous avons étudié les quasi-treillis de types dans ce formalisme. En particulier nous avons exhibé des conditions suffisantes pour qu'un quasi-treillis de constructeurs de types engendre un quasi-treillis de types. Nous avons ensuite montré que le problème de la satisfiabilité des contraintes de sous-typage dans ces structures est NP-complet si les extrema du quasi-treillis sont en nombre fini et tous constants. L'algorithme permettant de tester la satisfaction des contraintes voit sa complexité réduite à $O(n^3)$, où n est la taille du système de contraintes, dans le cas où toutes les variables du système sont bornées. Enfin, nous avons développé un algorithme de calcul explicite de solution aux contraintes de sous-typage, les solutions obtenues étant maximales dans les structures de type covariantes. De plus, cet algorithme fonctionne avec une hypothèse affaiblie qui demande que les maxima du quasi-treillis soient en nombre fini et tous constants, mais qui n'impose pas de condition sur les minima.

Dans une deuxième partie nous avons étudié le système de types avec surcharge. Dans un premier temps, nous avons présenté ce système et montré le

théorème d'auto-réduction pour le modèle d'exécution CSLD, ainsi que pour un modèle d'exécution typé avec substitutions. Nous avons ensuite discuté les choix de types que nous avons effectués pour les prédicats et les structures de données pour ISO-Prolog et SICStus Prolog. Nous avons présenté l'algorithme que nous avons développé pour la vérification des types. Cet algorithme infère le type des variables et gère la désambiguïsation des symboles surchargés. Nous avons réalisé une implantation, TCLP, de ces algorithmes, écrite en SICStus Prolog et CHR. Cette implantation nous a permis d'effectuer des tests grandeur nature, en particulier sur des bibliothèques SICStus Prolog, ainsi que sur le code de TCLP lui-même. Ces tests ont montré l'efficacité de l'algorithme précédent, celui-ci s'avérant suffisamment rapide pour être utilisé en pratique, même si, à cause de la complexité du système de types et des contraintes de sous-typage, les temps de typage sont bien plus importants que pour le typage de langages tels que C ou ML en pratique. Les tests ont également montré la pertinence du système de type en matière de détection d'erreurs de programmation. Enfin nous avons développé une heuristique pour l'inférence du type des prédicats. Cette heuristique a également été implantée dans TCLP. Les tests ont là aussi montré qu'elle est à la fois pertinente et suffisamment rapide pour être utilisée en pratique. Tous ces tests ont été réalisés à partir de TCLP, une implantation en SICStus Prolog et CHR, capable de se vérifier elle-même.

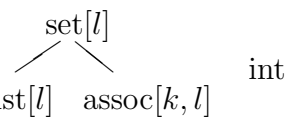
Problèmes ouverts, perspectives

Bien que cette thèse ait répondu à certaines questions, il en reste de nombreuses en suspens.

Contraintes de sous-typage

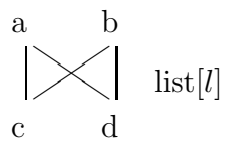
Concernant les contraintes de sous-typage, plusieurs problèmes restent ouverts, en particulier sur la satisfiabilité des contraintes de sous-typage dans des structures plus générales que les quasi-treillis dont les maxima sont des constantes en nombre fini.

L'algorithme présenté au chapitre 6 requiert que les maxima du quasi-treillis soient des constantes. Ceci est gênant lorsque l'un de ces maxima représente une structure de donnée paramétrique. L'exemple de structure ci-contre illustre ce problème. Ici, le constructeur set$[l]$ a un argument et est un maximum. Ce cas peut se présenter lorsque le programme à typer n'utilise pas de méta-programmation, ce qui rend le supertype term inutile. Ajouter un supertype à set$[l]$ revient à introduire artificiellement un type pour les ensembles non homogènes. On peut également voir cela comme une



réintroduction partielle d'un type $\top$, dans la mesure où $\text{set}(\tau_1)$ et $\text{set}(\tau_2)$ auraient un supertype commun, même si τ_1 et τ_2 n'en n'ont pas. Cependant, la décidabilité de satisfiabilité des contraintes de sous-typage dans des structures où les maxima et les minima ne sont pas des constantes reste un problème ouvert. La difficulté réside dans le traitement des variables non bornées. Lors de l'application de la règle (Intro $\uparrow$), une contrainte de la forme $\alpha \leq \tau$ est ajoutée au système. Or, si certains maxima ne sont pas des constantes, alors τ peut ne pas être atomique. Dans ce cas, on ajoute une nouvelle variable au système pour chaque argument du constructeur de tête de τ. Comme ces variables sont fraîches, elles ne sont pas bornées, ce qui va déclencher à terme une nouvelle utilisation de la règle (Intro $\uparrow$), ce qui va introduire de nouvelles variables, et ainsi de suite.

Un autre problème consiste à considérer un ordre partiel quelconque au lieu d'un quasi-treillis pour l'ensemble des constructeurs de types. Le problème à été montré PSPACE-complet dans le cas non structurel homogène avec des types finis par Frey [24]. Le problème reste ouvert pour le cas non structurel non homogène. La difficulté est ici liée au fait qu'un ensemble de types S n'a pas nécessairement de borne inférieure (ou supérieure), mais un ensemble de d'éléments maximaux (ou minimaux) dans l'ensemble des minorants (ou majorants) car la valeur que doit prendre $\gamma_{\{\beta_1,\beta_2\}}$ peut être différente selon α. Considérons par exemple la structure ci-contre, et le système de contraintes $C = \beta_1 \leq$ a, $\beta_2 \leq$ b, c $\leq \delta_1$, d $\leq \delta_2, \alpha_1 \leq \text{list}(\beta_1), \alpha_1 \leq \text{list}(\beta_2), \alpha_2 \leq \text{list}(\beta_1), \alpha_2 \leq \text{list}(\beta_2), \text{list}(\delta_1) \leq \alpha_1, \text{list}(\delta_2) \leq \alpha_2$. Dans le cas de α_1, la valuation de $\gamma_{\{\beta_1,\beta_2\}}$ doit être c, à cause de δ_1, alors que pour α_2 elle doit être d à cause de δ_2. Il faut donc deux variables différentes $\gamma^{\alpha_1}_{\{\beta_1,\beta_2\}}$ et $\gamma^{\alpha_2}_{\{\beta_1,\beta_2\}}$ pour factoriser les inégalités sur α_1 et α_2. Cependant, une variable γ^{α}_{A} ainsi introduite pour factoriser des inégalités sur une variable α peut nécessiter elle même une factorisation, c'est à dire l'introduction d'une variable $\gamma^{\gamma^{\alpha}_{A}}_{B}$, et ainsi de suite, ce qui peut alors mener à l'introduction d'une infinité de variables.



Enfin, les contraintes de sous-typage étant des contraintes d'ordre, leur utilisation pourrait être étendue à d'autres domaines que le typage. En particulier, il serait envisageable d'utiliser ces contraintes pour le raisonnement ontologique [43], puisque les ontologies sont des structures ordonnées.

Cette utilisation des contraintes conduit naturellement à l'étude de la négation dans les contraintes de sous-typage. Dans le cas du sous-typage *structurel*, Kuncak et Rinard [41] ont montré que la théorie du premier ordre des types non récursifs est décidable. Aiken et *al.* [68] ont montré que, dans le cas non structurel, le problème est décidable si les constructeurs de types ont au plus un argument, et est indécidable en général.

Système de types

Le système de type peut également être étendu dans plusieurs directions.

La notion de sous-typage est prépondérante dans les langages objets. Il serait intéressant d'étudier les extensions objet des langages logiques avec contraintes telles que LogTalk [49] et la bibliothèque d'objets de SICStus Prolog [6].

Dans le cas de SICStus Prolog, les objets sont vus comme des collections de définitions de prédicats, et le système d'objets repose fortement sur le système de modules. Typer les objets en SICStus Prolog nécessite donc de développer un système de types pour les modules. Le système d'héritage est basé sur la notion de prototype. Un objet n'est pas une instance d'une classe comme dans la majorité des langages à objets, mais est issu d'un autre objet, le prototype, qui lui sert de modèle. L'objet ainsi obtenu est une spécialisation du prototype. Le système de type doit donc intégrer dans la notion de sous-typage la relation prototype - spécialisation.

Dans LogTalk, les objets peuvent être soit des spécialisations de prototypes, soit des instances de classes. Les deux mécanismes peuvent cohabiter dans un même programme, mais ils doivent être appliqués à des hiérarchies différentes. Dans le cas de l'utilisation de classes, chaque classe représente un type et la relation de sous-typage peut être directement déduite de la relation d'héritage entre classes.

Une autre direction concerne le typage des langages de règles. Si on considère par exemple le langage CHR, la différence principale entre une règle CHR et une clause d'un programme logique avec contraintes est que la règle possède plusieurs têtes. Cela suggère que le système de type peut être adapté assez simplement pour typer CHR. À chaque contrainte CHR $\mathtt{c}/n$ est associé un type $\tau_1 \times \ldots \times \tau_n \rightarrow$ chr_constraint, les têtes de la règle étant typées avec une règle similaire à *(Head)* et le corps de la règle étant typé de manière similaire à la règle *(Query)*. Cependant, une difficulté apparaît lorsque le type des contraintes est polymorphe : comment les variables apparaissant dans les types des différentes têtes d'une règles sont-elles liées ? L'exemple suivant illustre ce problème :

Exemple 10.1 : Soit la contrainte CHR `link`/2 associant une clé à une donnée, de type $\alpha \times \beta \rightarrow$ chr_constraint. Soit la règle suivante, qui exprime qu'il ne peut y avoir qu'une seule donnée attachée à chaque clé :

```
link(X,A), link(X,B) <=> A=B, link(X,A).
```

Pour que la règle soit correctement typée, les types des têtes doivent être renommés de la même manière par rapport au type de la contrainte `link`/2, c'est-à-dire `A` et `B` doivent avoir le même type. Dans le cas contraire, la contrainte `A=B` serait mal typée. Cependant, si on considère l'ensemble de contraintes

$C =$ `link(X,3),link(X,'toto')`, qui se réécrit en $C' =$ `3=A, 'toto'=B, A=B, link(X,A)`, et que l'on suppose que le type de `3`, int, et le type de `'toto'`, atom, n'ont pas de supertype commun, alors C' n'est pas bien typé. ◇

La solution la plus simple serait d'interdire les contraintes CHR avec un type polymorphe. En effet, il semblerait que la majorité des solveurs actuellement écrits en CHR pourraient très bien s'accommoder de cette restriction. Une autre solution consisterait à avoir recours à un modèle d'exécution typé, par exemple en ajoutant aux contraintes CHR un argument correspondant aux variables apparaissant dans le type de ces contraintes.

Le système de types présenté dans cette thèse est également étudié pour typer les langages de règles actuellement développés pour raisonner sur le web sémantique, tels que Xcerpt [3]. En effet les règles de ce langage ont une forme proche des clauses dans les langages logiques avec contraintes. La différence principale se situe au niveau des termes manipulés, puisque l'arité des constructeurs de termes en Xcerpt n'est pas fixée. De même, l'ordre des sous-termes immédiats d'un terme donné n'est lui non plus pas toujours déterminé. Une solution pour adapter le système décrit dans cette thèse à Xcerpt serait de considérer que tous les arguments d'un même constructeur de terme doivent avoir le même type, et d'utiliser le sous-typage pour permettre l'utilisation d'éléments variés dans un terme. Par exemple, une bibliographie pourrait comprendre des auteurs, de type author, et des livres, de type book. Il suffit alors d'introduire un type, biblio_elem, qui serait un supertype commun à author et à book. En imposant que le type des éléments d'une bibliographie soit biblio_elem, on autorise les bibliographies à contenir à la fois des auteurs et des livres. Une deuxième différence est que le flux de données est connu statiquement dans Xcerpt. En utilisant les idées de [62], on pourrait obtenir un théorème d'auto-réduction pour un modèle d'exécution non typé avec substitutions.

Bibliographie

[1] C. Beierle. Type inferencing for polymorphic order-sorted logic programs. In *12th International Conference on Logic Programming ICLP'95*, pages 765–779. The MIT Press, 1995.

[2] G. Birkhoff. *A Survey of Modern Algebra*. Prentice Hall (Sd), 1977.

[3] F. Bry and S. Schaffert. A gentle introduction into Xcerpt, a rule-based query and transformation language for XML. In *Proceedings of International Workshop on Rule Markup Languages for Business Rules on the Semantic Web*, June 2002.

[4] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3) :138–164, 1988.

[5] L. Cardelli. Typeful programming. Technical report, Digital Equipment Corporation Systems Research Center, May 1989. SRC Research Report 45.

[6] M. Carlsson. SICStus Prolog. `http://www.sics.se/sicstus/`.

[7] A. Colmerauer. Equations and inequalitions on finite and infinite trees. In *Proc. of the International conference on fifth generation computer systems FGCS'84*, pages 85–99. ICOT, 1984.

[8] A. Colmerauer. Prolog in 10 figures. *Communications of the ACM*, 28(12) :1296–1310, 1985.

[9] A. Colmerauer. Specification of Prolog IV. Technical report, LIM Technical Report, 1996.

[10] E. Coquery. TCLP. `http://contraintes.inria.fr/~coquery/tclp/`.

[11] E. Coquery. TCLP : A type checker for CLP($\mathcal{X}$). In F. Mesnard and A. Serebrenik, editors, *Proceedings of the 13th Workshop on Logic Programming Environments*, pages 17–30. Katholieke Universiteit Leuven, 2003.

[12] E. Coquery and F. Fages. From typing constraints to typed constraints in CHR. In *Workshop Proceedings of the Rule-Based Constraint Reasoning and Programming*, 2001.

[13] E. Coquery and F. Fages. TCLP : overloading, subtyping and parametric polymorphism made practical for constraint logic programming. Technical report, INRIA Rocquencourt, 2002.

[14] E. Coquery and F. Fages. Subtyping constraints in quasi-lattices. In P. K. Pandya and J. Radhakrishnan, editors, *Foundations of Software Tehcnology and Theoretical Computer Science, FSTTCS'03*, volume 2914 of *LNCS*, pages 136–148. Springer-Verlag, 2003.

[15] V. Santos Costa, D.H.D. Warren, and R. Yang. The Andorra-I preprocessor : Supporting full Prolog on the basic Andorra model. In *Proceedings of the 8th International Conference on Logic Programming ICLP'91*, pages 443–456. MIT Press, 1991.

[16] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25 :95–169, 1983.

[17] L. Damas and R. Milner. Principal type-schemes for functional programs. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 207–212, 1982.

[18] R. Dietrich and F. Hagl. A polymorphic type system with subtypes for Prolog. In H. Ganzinger, editor, *Proceedings of the European Symposium on Programming ESOP'88*, LNCS, pages 79–93. Springer-Verlag, 1988.

[19] W. Drabent, J. Małuszyński, and P. Pietrzak. Using parametric set constraints for locating errors in CLP programs. *Theory and Practice of Logic Programming*, 2(4-5) :549–611, 2002.

[20] E. Coquery et F. Fages. Surcharge et sous-typage dans TCLP. In *Actes des Journées Francophones de la Programmation en Logique avec Contraintes JFPLC'2002*, 2002.

[21] F. Fages. *Programmation Logique par Contraintes*. Collection Cours de l'Ecole Polytechnique. Ellipses, Mai 1996.

[22] F. Fages and E. Coquery. Typing constraint logic programs. *Theory and Practice of Logic Programming*, 1(6) :751–777, November 2001.

[23] F. Fages and M. Paltrinieri. A generic type system for CLP($\mathcal{X}$). Technical report, Ecole Normale Supérieure LIENS 97-16, December 1997. Poster à JICSLP'98, MIT Press, 1998.

[24] A. Frey. Satisfying subtype inequalities in polynomial space. In *Proceedings of the 1997 International Static Analysis Symposium*, number 1302 in LNCS, 1997.

[25] T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3) :95–138, October 1998.

[26] T. Frühwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In G. Kahn, editor, *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science*, pages 300–309. IEEE Computer Society Press, July 1991.

[27] Y.-C. Fuh and P. Mishra. Type inference with subtypes. In H. Ganzinger, editor, *Proceedings of the European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 94–114. Springer-Verlag, 1988.

[28] Y.-C. Fuh and P. Mishra. Polymorphic subtype inference : Closing the theory-practice gap. In J. Diaz and F. Orejas, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Developpement*, volume 351 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1989.

[29] M. Fähndrich and A. Aiken. Making set-constraint program analyses scale. In *The CP'96 Workshop on Set Constraints*, August 1996. Available as UC Berkeley Computer Science Technical Report UCB//CSD-96-917.

[30] I. Gnaedig, C. Kirchner, and H. Kirchner. Equational completion in order-sorted algebras. *Theoretical Computer Science*, 72 :169–202, 1990.

[31] J.A. Goguen and J. Meseguer. Equality, types, modules and generics for logic programming. In *Proceedings of the 2nd International Conference on Logic Programming*, pages 115–126, Uppsala, Sweden, 1984.

[32] M. Hanus. *Logic Programming with Type Specifications*, chapter 3, pages 91–140. MIT Press, 1992. In [55].

[33] N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *Proceedings of the 17th ACM Symposium on Principle of Programming Langages*, pages 197–209, 1990.

[34] N. Heintze and J. Jaffar. *Semantic Types for Logic Programs*, chapter 4, pages 141–155. MIT Press, 1992. In [55].

[35] P. Hill and J. Lloyd. *The Gödel programming language*. MIT Press, 1994.

[36] P. M. Hill and R. W. Topor. *A Semantics for Typed Logic Programs*, chapter 1, pages 1–61. MIT Press, 1992. In [55].

[37] G. Huet. *Résolution d'équations dans des langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université Paris VII, 1976.

[38] J. Jaffar and J.-L. Lassez. Constraint logic programming. Technical Report TR 86/73, Monash University, 1986.

[39] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, POPL'87*, pages 111–119. ACM SIGACT-SIGPLAN, 1987.

[40] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(3) :339–395, July 1992.

[41] V. Kuncak and M. Rinard. Structural subtyping of non-recursive types is decidable. In P. G. Kolaitis, editor, *18th IEEE Symposium on Logic in*

Computer Science (LICS 2003), pages 96–107. IEEE Computer Society, June 2003.

[42] K. Kwon, G. Nadathur, and D. Wilson. Implementing polymorphic typing in a logic programming language. *Computer Languages*, 20(1) :25–42, 1994.

[43] F. Laburthe. Constraints over ontologies. In F. Rossi, editor, *CP*, volume 2833 of *Lecture Notes in Computer Science*, pages 878–882. Springer, 2003.

[44] T.K. Lakshman and U.S. Reddy. Typed Prolog : A semantic reconstruction of the Mycroft-O'Keefe type system. In V. Saraswat and K. Ueda, editors, *Proceedings of the 1991 International Symposium on Logic Programming*, pages 202–217. MIT Press, 1991.

[45] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3) :348–375, December 1978.

[46] P. Mishra. Towards a theory of types in Prolog. In *Proceedings of International Symposium on Logic Programming*, pages 289–298. IEEE, 1984.

[47] J. Mitchell. Coercion and type inference. In *Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages POPL'84*, pages 175–185, 1984.

[48] J. Mitchell. Type inference with simple subtypes. *Journal of Functionnal Programming*, 1(3) :245–286, 1991.

[49] P. Moura. Logtalk. `http://www.logtalk.org`.

[50] A. Mycroft and R.A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23 :295–307, 1984.

[51] G. Nadathur and D. Miller. An overview of lambda Prolog. In K. Bowen and R. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827. MIT Press, August 1988.

[52] M. Nivat and A. Podelski. Definite tree languages. *Bulletin of the EATCS*, 38 :186–190, June 1989.

[53] J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems*, 17(4) :576–599, July 1995. Preliminary version in Proceedings of POPL'95, 22nd Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 367-378.

[54] J. Palsberg and S. Smith. Constrained types and their expressiveness. *ACM Transactions on Programming Languages and Systems*, 18(5) :519–527, September 1996.

[55] F. Pfenning, editor. *Types in Logic Programming*. MIT Press, 1992.

[56] F. Pottier. *Synthèse de types en présence de sous-typage : de la théorie à la pratique*. PhD thesis, Université Paris 7, July 1998.

[57] F. Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4) :312–347, November 2000.

[58] F. Pottier. `Wallace` : an efficient implementation of type inference with subtyping, February 2000. `http://pauillac.inria.fr/~fpottier/wallace/`.

[59] V. Pratt and J. Tiuryn. Satisfiability of inequalities in a poset. *Fundamenta Informaticae*, 28(1-2) :165–182, 1996.

[60] J.C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, volume 19 of *LNCS*, pages 408–425. Springer Verlag, 1974.

[61] V. Simonet. Type inference with structural subtyping : A faithful formalization of an efficient constraint solver. In A. Ohori, editor, *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS'03)*, volume 2895 of *Lecture Notes in Computer Science*, pages 283–302, Beijing, China, November 2003. Springer-Verlag.

[62] J.-G. Smaus, F. Fages, and P. Deransart. Using modes to ensure subject reduction for typed logic programs with subtyping. In *Proceedings of FSTTCS '2000*, number 1974 in LNCS. Springer-Verlag, 2000.

[63] G. Smolka. Logic programming with polymorphically order-sorted types. In *Algebraic and Logic Programming ALP'88*, number 343 in LNCS, pages 53–70. J. Grabowski, P. Lescanne, W. Wechler, 1988.

[64] G. Smolka. *Logic Programming over Polymorphically Order-Sorted Types*. PhD thesis, Universitat Kaiserslautern, 1989.

[65] G. Smolka and W. Nutt. Order sorted equational computation. In *Proceedings of CREAS*, Austin, Texas, May 1987.

[66] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3) :17–64, 1996.

[67] C. Strachey. Fundamental concepts in programming languages. *Lecture Notes, International Summer School in Computer Programming, Copenhagen*, August 1967. Reprinted in *Higher-Order and Symbolic Computation*, 13(1/2), pp. 1-49, 2000.

[68] Z. Su, A. Aiken, J. Niehren, T. Priesnitz, and R. Treinen. First-order theory of subtyping constraints. In J. Mitchell, editor, *Proceedings of the 29th ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, pages 203–216. ACM SIGACT-SIGPLAN, January 2002.

[69] J. Tiuryn. Subtype inequalities. In *7th IEEE Symposium on Logic in Computer Science*, pages 308–315. IEEE Computer Society Press, June 1992.

[70] V. Trifonov and S. Smith. Subtyping constrained types. In *Proc. 3rd Int'l Symposium on Static Analysis*, number 1145 in LNCS, pages 349–365. Springer, 1996.

[71] E. Yardeni, T. Frühwirth, and E. Shapiro. *Polymorphically typed logic programs*, chapter 2, pages 63–90. MIT Press, 1992. In [55].

Liste des tableaux et figures

Index

Liste des symboles