Mémoire d'habilitation à diriger des recherches en informatique

présenté à

l'Université de Paris 7

par

François Pottier

Types et contraintes

Soutenu le 15 décembre 2004 devant le jury composé de :

MM.	Guy Cousineau	Président
	Roberto Di Cosmo	Rapporteurs
	Jean-Pierre Jouannaud	
	Martin Odersky	
	Didier Rémy	Directeur

Table des matières

Chapitre 1

Introduction

LE TYPAGE est une discipline permettant d'associer des spécifications, ou *types*, aux programmes. Un type est typiquement un terme de petite taille. Le typage se distingue en général des autres analyses de programmes par une relative simplicité, due en partie à sa définition compositionnelle : le type attribué à un fragment de programme n'est fonction que des types attribués aux sous-fragments qui le composent.

L'intérêt du typage est multiple. D'abord, un théorème de *sûreté du typage*, démontré par le concepteur de tout système de types, garantit qu'un programme bien typé ne peut pas « planter : » son exécution ne peut pas échouer inopinément. Ce résultat est obtenu de façon *statique*, c'est-à-dire avant l'exécution du programme. Il fournit à l'utilisateur une garantie partielle de robustesse, ainsi qu'une propriété de *sécurité* certes limitée, mais sans laquelle aucune garantie plus poussée ne peut être fournie.

Ensuite, certains systèmes de types, plus ambitieux, permettent d'établir statiquement des propriétés plus avancées, par exemple l'obéissance à une politique de contrôle d'*accès* aux ressources, ou encore l'obéissance à une politique de contrôle des *flots d'information*. D'autres permettent l'analyse de *flots de données*, qui n'intéresse en général guère le programmeur, mais permet souvent de produire un programme compilé plus efficace.

Enfin, de par son caractère compositionnel, et de par l'exploitation des notions duales de *polymorphisme* paramétrique et d'*abstraction* de types, le typage favorise la *modularité*, c'est-à-dire le découpage des programmes en unités indépendantes et complémentaires, activité centrale pour le développement de systèmes logiciels complexes.

On souhaite souvent que l'analyse d'un programme par le système de types se fasse de façon automatique ou presque automatique. En effet, même si la discipline de typage est supposée connue du programmeur, il semble utile de ne pas lui imposer un surcroît de travail, donc de n'exiger de lui qu'une aide minime. Ainsi, on lui demandera typiquement de fournir la spécification de chacun des modules qui composent un programme, mais, si possible, pas plus. Par ailleurs, lorsque les résultats de l'analyse sont destinés à être exploités par un compilateur, il est bon que celle-ci soit entièrement automatique. Il semble donc souhaitable d'étudier le problème de *l'inférence de types*, à savoir la recherche automatique ou presque automatique du ou des types qu'admet un programme.

Parce que le typage est de nature compositionnelle, les problèmes d'inférence de types admettent également une décomposition naturelle. En d'autres termes, le problème d'inférence associé à un fragment de programme admet en général une solution si et seulement si les sous-problèmes associés aux sous-fragments qui le composent admettent également des solutions, et si celles-ci sont cohérentes les unes vis-à-vis des autres. Les conditions de cohérence requises varient d'un système à l'autre, mais on retrouvera souvent, sous une forme ou une autre, l'exigence que deux sous-fragments admettent un type commun.

Dès lors, on constate que tout langage adapté à l'expression des problèmes d'inférence de types doit offrir la *conjonction*, permettant de combiner plusieurs sous-problèmes ; la *quantification existentielle*, permettant d'introduire une *variable de types* dénotant un type à déterminer ; et des *prédicats* portant sur les types, par exemple le prédicat binaire d'égalité, permettant d'exiger la coïncidence de deux types. En d'autres termes, l'inférence de types semble devoir naturellement se réduire à la satisfaction

de formules logiques, ou *contraintes*, appartenant typiquement à une théorie du premier ordre fondée sur un ou plusieurs prédicats tels que l'égalité.

L'expérience montre que, même si le langage de programmation considéré est riche, le langage de contraintes nécessaire pour en exprimer les problèmes d'inférence reste relativement restreint. La réduction des problèmes d'inférence vers les problèmes de satisfaction de contraintes permet donc de restreindre sensiblement l'univers du discours, et constitue une première étape utile.

Enfin, les contraintes peuvent non seulement permettre l'expression des problèmes d'inférence de types, mais également participer à la définition même d'un système de types. En effet, en leur absence, ne sont disponibles que quelques mécanismes classiques, très utilisés mais d'une expressivité et d'une élégance limitées, tels l'emploi répété d'une même méta-variable pour coder une équation, ou l'application d'une substitution pour coder une implication. L'emploi explicite de contraintes dans la définition d'un système de types permet d'échapper à ces idiosyncrasies et d'accéder à un plus large espace de conception.

Types et *contraintes* seront donc les principales entités manipulées tout au long de ce mémoire. Mon propos est de présenter, dans un cadre relativement général, une discipline de typage et un algorithme d'inférence de types, tous deux à base de types et contraintes ; puis d'en illustrer plusieurs applications, directes ou indirectes, variantes, et extensions. J'espère ainsi donner un aperçu assez large des techniques de typage à base de contraintes et des divers partis que l'on peut en tirer.

Ce mémoire est constitué d'un ensemble de chapitres, dont chacun décrit brièvement et commente informellement un ou plusieurs articles précédemment publiés. Afin d'éviter une trop grande redondance, je choisis de ne pas reproduire ici ces articles. De plus, je mettrai délibérément l'accent sur les aspects les moins développés dans les versions publiées : cheminement historique, choix et compromis de conception, etc.

Dans le chapitre 2, je m'intéresse aux systèmes de types à base de contraintes en général. Sans émettre beaucoup d'hypothèses à propos du langage de contraintes employé ou du langage de programmation considéré, on peut démontrer quelques résultats généraux, comme la sûreté du typage et la réduction de l'inférence de types à la résolution de contraintes. Ces deux résultats acquis, on pourra se concentrer, lors de la conception d'une application particulière, sur le choix du langage de contraintes et le problème de leur résolution.

Le chapitre 3 fournit une application directe du cadre théorique développé au chapitre précédent. Celle-ci se fait en deux temps. D'abord, je définis de façon détaillée un langage de contraintes particulier, dont l'expressivité me semble remarquable et dont la complexité théorique reste polynomiale. L'interprétation de la relation de sous-typage y est non structurelle et contient la théorie équationnelle des rangées. De plus, le langage de contraintes offre non seulement le prédicat binaire de sous-typage, mais également une famille de prédicats ternaires de sous-typage conditionnel. Ensuite, je m'intéresse à un langage de programmation particulier, doté d'enregistrements auxquels on peut appliquer une opération de concaténation, et j'explique comment attribuer à cette opération un schéma de types contraint sûr et précis.

Le chapitre 4 fournit deux applications indirectes de ce même cadre théorique. Dans ces deux cas, on souhaite obtenir, grâce au typage, une garantie inhabituelle. Dans le premier cas, il s'agit de contrôler les flots d'information à travers un programme purement fonctionnel. Si la sémantique du langage de programmation considéré est standard, la propriété souhaitée, à savoir la non-interférence, est inhabituelle, et n'est pas une propriété de sûreté. Dans le second cas, il s'agit de contrôler l'accès aux ressources à l'aide d'un mécanisme inspiré de celui du langage Java : à l'inverse, ici, la propriété souhaitée, à savoir la sûreté du typage, est standard, mais la sémantique du langage de programmation considéré est inhabituelle, car elle emploie la technique dite d'inspection de pile. Dans les deux cas, j'emploie un codage pour réduire le problème à une formulation standard.

Le chapitre 5 décrit une variante du système de types présenté au chapitre 2, dont l'objectif est de contrôler les flots d'information à travers un programme mêlant traits fonctionnels et traits impératifs. Pour résoudre ce problème plus complexe, l'emploi d'un codage ne semble plus possible. Aussi, je suis contraint de définir une variante du système de types initial, elle aussi à base de contraintes, mais

doté de règles différentes, et plus complexes. Pour démontrer la correction de ce système, on ne peut malheureusement plus s'appuyer directement sur les résultats généraux du chapitre 2. Néanmoins, les notions et techniques de preuve requises sont essentiellement inchangées.

Le chapitre 6 décrit une extension du système de types du chapitre 2. Celle-ci permet, grâce à des règles de typage plus élaborées, d'une part, de considérer de nouveaux programmes comme bien typés, ce qui augmente l'expressivité du langage de programmation ; et, d'autre part, d'attribuer des types plus précis à certains programmes existants, ce qui permet d'éliminer plus d'erreurs potentielles. Le trait caractéristique de cette extension est de permettre à un test dynamique de fournir une information statique supplémentaire, ce qui se traduit, en termes de contraintes, par l'exploitation du connecteur implication. J'illustre d'abord l'intérêt de cette extension en démontrant qu'elle permet de considérer comme bien typés tous les programmes obtenus par défonctionalisation. Puis, je m'intéresse à la réduction de l'inférence de types à la résolution de contraintes, qui cause, dans ce cadre, quelques difficultés nouvelles.

Enfin, le chapitre 7 est consacré à une courte réflexion sur l'état du domaine et sur la façon dont celui-ci pourrait, selon moi, être amené à évoluer.

Chapitre 2

Cadre général

Ce chapitre commente une partie du matériau publié dans [1–4].

LE SYSTÈME DE TYPES des langages de programmation de la famille ML, dont les membres les plus influents sont Standard ML, Objective Caml et Haskell, est fondé sur la discipline dite de Hindley [5] et Milner [6]. Hindley a résolu le problème de l'inférence de types pour le λ-calcul simplement typé en démontrant de façon constructive que toute expression admet un *type principal*. Le λ-calcul simplement typé est un système de types *monomorphe*, où une expression e ne peut admettre *simultanément* qu'un seul type. Néanmoins, le type principal de e, exhibé par l'algorithme de Hindley, peut contenir des *variables de types*, ce qui indique alors que e admet en fait plusieurs (une infinité de) types. Ainsi, l'étude de l'inférence de types fait surgir une forme de *polymorphisme* paramétrique, une notion due à Strachey [7]. L'idée de Milner sera d'internaliser la notion de type principal en introduisant les *schémas de types* et en permettant effectivement à une expression d'avoir simultanément plusieurs types. L'inférence de types reste alors possible, et repose toujours indirectement sur l'unification du premier ordre, mais le lien semble plus ténu : la notion de contrainte n'apparaît pas clairement dans les algorithmes proposés par Milner [6]. Le lien entre inférence de types et résolution de contraintes se renforcera progressivement, au cours des deux décennies qui vont suivre, au fil des propositions visant à généraliser la discipline de Hindley et Milner pour y introduire des notions plus complexes, comme le sous-typage. Petit à petit, les contraintes prennent un rôle central, jusqu'à participer explicitement à la définition même des systèmes de types héritiers de la discipline de Hindley et Milner aussi bien qu'à la formulation de leur algorithme d'inférence de types.

Ma thèse de doctorat [8], sur laquelle je ne reviendrai pas ici, concernait l'inférence de types en présence de sous-typage non structurel, et c'est pourquoi j'ai été amené à étudier de près le rôle des contraintes dans les systèmes de types. Ce chapitre donne un aperçu de la compréhension que j'en ai aujourd'hui. Son contenu provient en grande partie d'un texte écrit en collaboration avec Didier Rémy [1], et qui vise à donner une présentation moderne du système de types de ML et de ses extensions à base de contraintes. Il s'agit d'un travail d'exposition et de modernisation d'idées connues, plus que de recherche. Néanmoins, parce que l'effort de rédaction a été important – la version longue de [1], non publiée car toujours en friche, atteint deux cents pages – et parce que ces idées anciennes sont formulées de façon nouvelle, j'ai souhaité donner à ce travail une place importante dans ce mémoire. De plus, cela me permet de monter le décor, c'est-à-dire d'introduire le cadre général dans lequel viendront s'inscrire les chapitres suivants de ce mémoire. Dans ce chapitre, je cite également quelques travaux de moindre ampleur auxquels j'ai participé et qui concernent le système HM(X) [2, 4] et son adaptation au join-calcul JOIN(X) [3].

En bref, le contenu de ce chapitre est le suivant. Je m'intéresse initialement à la réduction de l'inférence de types vers la résolution de contraintes, d'abord pour le λ-calcul simplement typé (§2.1), puis pour le système de types de Damas et Milner (§2.2). Ensuite, j'explique comment les contraintes peuvent participer à la définition même d'un système de types, et présente brièvement le système HM(X) (§2.3). Enfin, je décris et compare plusieurs façons d'établir la sûreté du typage pour un système à base de contraintes tel que HM(X) (§2.4).

$$\Gamma \vdash x : \Gamma(x) \qquad \frac{\Gamma; x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}$$

FIG. 2.1 – Le λ-calcul simplement typé

2.1 Le λ-calcul simplement typé

La définition du λ-calcul simplement typé est donnée par la figure 2.1. On y trouve la définition inductive d'un prédicat dont la forme générale est $\Gamma \vdash e : \tau$, où Γ est un *environnement*, e une *expression* et τ un *type*. Les expressions sont données par la grammaire

$$e ::= x \mid \lambda x.e \mid e\, e$$

où x dénote une *variable*. Les types sont donnés par la grammaire

$$\tau ::= \alpha \mid \tau \to \tau$$

où α dénote une *variable de types*. Un environnement est une fonction partielle des variables vers les types. Un triplet $\Gamma \vdash e : \tau$ est dit *pré-jugement*. Il est dit *jugement* s'il est dérivable d'après les règles de la figure 2.1. Une paire (Γ, τ) est un *typage* de e si et seulement si $\Gamma \vdash e : \tau$ est un jugement ; c'est un typage *principal* de e si et seulement si tout autre typage de e s'en déduit par substitution de variables de types par des types. L'expression e est dite *typable* si et seulement si elle admet un typage.

On sait que, dans le λ-calcul simplement typé, toute expression typable e possède un typage principal. Ce fait a été démontré, dans le cadre de la logique combinatoire, indépendamment par Curry et par Hindley [5]. La preuve de Curry est directe, tandis que celle de Hindley fait appel à l'algorithme d'unification de Robinson [9]. Hindley n'emploie cependant pas explicitement la notion de contrainte d'égalité : au lieu de cela, il utilise l'algorithme d'unification comme une boîte noire qui, étant donnés deux schémas de types arbitraires, produit leur borne supérieure au sens de la relation d'instanciation, dénommée *highest common instance*, si elle existe. Hindley fait donc implicitement alterner génération et résolution de contraintes.

Il semble difficile de déterminer avec exactitude à partir de quelle période on a pu considérer l'inférence de types pour le λ-calcul simplement typé comme la combinaison de deux phases de génération et de résolution de contraintes. Cette vision s'est probablement mise en place de façon progressive au cours des années 1980, à la faveur de l'introduction de contraintes plus générales, comme les contraintes de sous-typage [10]. On la trouve, par exemple, de façon plus ou moins informelle, chez Clément, Despeyroux, Despeyroux et Kahn [11] ou chez Cardelli [12]. La première réduction explicite de l'inférence de types pour le λ-calcul simplement typé à la résolution de contraintes d'égalité semble due à Wand [13].

2.1.1 Une réduction de l'inférence à la résolution de contraintes

Les contraintes engendrées par l'algorithme de Wand sont constituées d'équations entre types et de conjonctions, à l'exclusion de toute autre construction. De ce fait, Wand ne peut traiter la notion de variable de types *fraîche* que de manière informelle. Suivant Jouannaud et Kirchner [14], je préfère ajouter au langage de contraintes un quantificateur existentiel, ce qui permet un traitement élégant et formel de cette notion.

On peut alors réduire l'inférence de types pour le λ-calcul simplement typé à la résolution de contraintes d'égalité de la façon suivante. Définissons d'abord le langage de contraintes considéré :

$$C ::= \tau = \tau \mid C \wedge C \mid \exists \alpha.C$$

Les contraintes sont formées à partir d'équations entre types à l'aide de la conjonction et de la quantification existentielle. Leur interprétation logique se fait traditionnellement dans l'univers de Herbrand,

$$\begin{array}{rcl} [\![\Gamma \vdash x : \tau]\!] & = & \Gamma(x) = \tau \\ [\![\Gamma \vdash \lambda x.e : \tau]\!] & = & \exists\alpha_1\alpha_2.([\![\Gamma; x : \alpha_1 \vdash e : \alpha_2]\!] \wedge \alpha_1 \to \alpha_2 = \tau) \\ [\![\Gamma \vdash e_1\, e_2 : \tau]\!] & = & \exists\alpha.([\![\Gamma \vdash e_1 : \alpha \to \tau]\!] \wedge [\![\Gamma \vdash e_2 : \alpha]\!]) \end{array}$$

FIG. 2.2 – Génération de contraintes pour le λ-calcul simplement typé

c'est-à-dire dans un modèle d'arbres finis. On pourrait les interpréter dans un univers d'arbres infinis ou d'arbres réguliers ; on obtiendrait alors un algorithme d'inférence pour le λ-calcul simplement typé doté de types récursifs. La résolution de contraintes consiste à déterminer si une contrainte donnée C est ou non satisfiable. Les algorithmes de résolution usuels [14] procèdent par réécriture vers une forme dite *résolue*, laquelle permet, si on le désire, d'exhiber aisément un unificateur principal.

À chaque pré-jugement $\Gamma \vdash e : \tau$, où le domaine de Γ contient les variables libres de e, on peut associer une contrainte, notée $[\![\Gamma \vdash e : \tau]\!]$. La définition de cette contrainte se fait par induction sur la structure de l'expression e, et est donnée par la figure 2.2.

Il est entendu que les variables de types α_1, α_2 et α doivent être prises *fraîches* vis-à-vis de Γ et τ, c'est-à-dire non libres dans Γ ou τ. Ce critère de fraîcheur est formel, car exprimable de façon locale. On pourrait l'expliciter en ajoutant la condition de bord $\alpha_1, \alpha_2, \alpha \notin \mathrm{ftv}(\Gamma, \tau)$, ou encore, suivant Gabbay et Pitts [15], en écrivant $И\alpha_1\alpha_2.\exists\alpha_1\alpha_2$ et $И\alpha.\exists\alpha$ au lieu de $\exists\alpha_1\alpha_2$ et $\exists\alpha$, respectivement. (Le quantificateur $И$ appartient au *méta-langage* et lie la *méta-variable* α, tandis que le quantificateur $\exists$ appartient au langage *objet*, c'est-à-dire au langage de contraintes, et lie la variable de types *dénotée* par la méta-variable α.) On notera que toute variable libre dans $[\![\Gamma \vdash e : \tau]\!]$ est nécessairement libre dans Γ ou dans τ.

On peut alors démontrer les propriétés suivantes :

Théorème 2.1 *ϕ est solution de $[\![\Gamma \vdash e : \tau]\!]$ si et seulement si $(\phi\Gamma, \phi\tau)$ est un typage de e.* ◇

Corollaire 2.2 *Soit α une variable de types arbitraire. Soit Γ un environnement associant à chaque variable libre dans e une variable de types arbitraire distincte et distincte de α. Alors e est typable si et seulement si $[\![\Gamma \vdash e : \alpha]\!]$ est satisfiable. De plus, si ϕ est solution principale de $[\![\Gamma \vdash e : \alpha]\!]$, alors $(\phi\Gamma, \phi\alpha)$ est un typage principal de e.* ◇

Ces résultats montrent tout d'abord que l'on a réduit l'inférence de types, c'est-à-dire la question de savoir si un terme est typable, à la résolution de contraintes, c'est-à-dire à la question de savoir si une contrainte est satisfiable. De plus, ils prouvent que le λ-calcul simplement typé admet l'existence de typages principaux :

Corollaire 2.3 *Si e est typable, alors e admet un typage principal.* ◇

Cette propriété signifie qu'il est possible d'analyser une expression e *hors de son contexte* et d'inférer non seulement le type de son résultat, mais également les exigences qu'elle porte sur son environnement. Elle a donc des applications en termes d'analyse et de compilation séparées [16, 17]. Elle n'est malheureusement pas satisfaite par le système de types de ML ; je reviendrai sur ce point (§2.2.1).

Il est aisé de vérifier que la contrainte $[\![\Gamma \vdash e : \alpha]\!]$ du corollaire 2.2 est de taille $O(n)$, où n est la taille de e, et peut être construite en temps $O(n)$ ou $O(n \log n)$, selon la façon dont les variables et l'environnement sont représentées. Un algorithme d'unification classique, tel que celui de Huet [18], fondé sur la structure d'*union-find* de Tarjan [19], permet d'en déterminer la satisfiabilité en temps $O(n\alpha(n))$. (L'algorithme d'unification linéaire de Paterson et Wegman [20] ne m'intéresse pas ici, car il n'est pas incrémental et ne sera plus utilisable dans le cadre de ML.) La complexité de l'inférence de types pour le λ-calcul simplement typé est donc $O(n \log n)$ au pire.

2.1.2 Une seconde réduction de l'inférence à la résolution

Dans les paragraphes précédents, j'ai décomposé le problème de l'inférence de telle façon que la notion d'environnement, sa construction et sa consultation, sont entièrement locales à la première phase,

$$\begin{array}{rcl} [\![x : \tau]\!] & = & x = \tau \\ [\![\lambda x.e : \tau]\!] & = & \exists \alpha_1 \alpha_2.(\textbf{\textit{def}}\ x : \alpha_1\ \textbf{\textit{in}}\ [\![e : \alpha_2]\!] \wedge \alpha_1 \to \alpha_2 = \tau) \\ [\![e_1\ e_2 : \tau]\!] & = & \exists \alpha.([\![e_1 : \alpha \to \tau]\!] \wedge [\![e_2 : \alpha]\!]) \end{array}$$

FIG. 2.3 – Génération de contraintes pour le λ-calcul simplement typé (variante)

à savoir la génération de contraintes. La seconde phase, c'est-à-dire la résolution de contraintes, n'a plus de notion d'environnement, puisque cette notion n'apparaît pas dans la grammaire des contraintes. Or, il est possible, si on le souhaite, d'effectuer la décomposition de façon légèrement différente, de sorte que la notion d'environnement soit, au contraire, propre à la seconde phase.

Pour cela, j'enrichis la syntaxe des contraintes :

$$C ::= \ldots \mid x = \tau \mid \textbf{\textit{def}}\ x : \tau\ \textbf{\textit{in}}\ C$$

J'autorise à présent des variables x à apparaître libres dans les contraintes. Par conséquent, l'interprétation logique des contraintes se fera non seulement vis-à-vis d'une valuation ϕ, qui à chaque variable de types α associe un élément du modèle dans lequel sont interprétés les types, mais également vis-à-vis d'une seconde valuation ψ, qui à chaque variable x associe un tel élément. J'introduis deux nouvelles formes de contraintes, dotées de l'interprétation suivante. L'équation $x = \tau$ est satisfaite par les valuations ϕ et ψ si et seulement si ψx et $\phi\tau$ coïncident. La contrainte ***def*** $x : \tau$ ***in*** C est satisfaite par ϕ et ψ si et seulement si C est satisfaite par ϕ et $\psi[x \mapsto \phi\tau]$.

L'expressivité du langage de contraintes n'est guère affectée par cette extension. En effet, on peut vérifier que la loi d'équivalence

$$\textbf{\textit{def}}\ x : \tau\ \textbf{\textit{in}}\ C \equiv [\tau/x]C$$

est validée par cette interprétation. (Deux contraintes sont équivalentes si et seulement si elles sont satisfaites par les mêmes valuations.) En d'autres termes, la construction ***def*** est une forme de substitution explicite. Elle peut être considérée, dans une certaine mesure, comme un sucre syntaxique : en effet, si une contrainte C n'a aucune variable libre, alors la loi ci-dessus, orientée de gauche à droite et considérée comme une règle de réécriture, permet de réécrire C en une contrainte équivalente et exprimée dans la syntaxe initiale. Le gain d'expressivité réside donc dans l'existence de contraintes comportant des variables libres : par exemple, $\exists\alpha.(x_1 = \alpha \wedge x_2 = \alpha)$ exprime le fait que les variables x_1 et x_2 doivent avoir le même type. On peut également écrire $x = \tau_1 \wedge x = \tau_2$, ce qui pourrait rappeler la notion de *type intersection* [21, 22] ; cependant, d'après l'interprétation ci-dessus, cette contrainte est équivalente à $x = \tau_1 \wedge \tau_1 = \tau_2$. Nous ne quittons donc pas le cadre simplement typé, où toute entité (variable ou expression) est monomorphe.

On peut à présent proposer une seconde réduction de l'inférence de types pour le λ-calcul simplement typé à la résolution de contraintes. Elle est donnée par la figure 2.3, où apparaît la définition d'une fonction qui à toute expression e et à tout type τ associe une contrainte $[\![e : \tau]\!]$. Notons que la notion d'environnement a disparu ! La consultation de l'environnement, qui permettait d'engendrer une égalité de la forme $\Gamma(x) = \tau$, a été supprimée, et on engendre à présent l'égalité $x = \tau$, où le nom x n'est pas résolu. L'extension de l'environnement, notée $\Gamma; x : \alpha_1$ et employée pour associer le type α_1 à la variable x lors de l'analyse d'une λ-abstraction $\lambda x.e$, a également disparu. À présent, l'analyse du corps e de l'abstraction produit une contrainte $[\![e : \alpha_2]\!]$ dans laquelle peuvent apparaître des occurrences libres de x. On leur donne un sens a posteriori en enveloppant cette contrainte dans le contexte ***def*** $x : \alpha_1$ ***in*** $[]$.

Quels sont les avantages de cette présentation vis-à-vis de l'approche initiale ? On peut en citer plusieurs, tous relativement mineurs. D'abord, la spécification est maintenant plus abstraite. Le solveur de contraintes peut choisir de procéder initialement à une élimination des ***def*** par substitution, comme suggéré plus haut, et on retrouve alors essentiellement l'approche initiale. Ou bien il peut adopter une autre stratégie de réécriture, par exemple *bottom-up*, et on obtient alors un algorithme différent. Ensuite, le corollaire 2.2 peut être reformulé de façon légèrement plus simple, car on n'a plus besoin de construire explicitement un environnement Γ constitué de variables fraîches distinctes :

$$\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma; x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \textbf{\textit{let}}\ x = e_1\ \textbf{\textit{in}}\ e_2 : \tau} \qquad \frac{\Gamma \vdash e : \tau \qquad \bar{\alpha}\ \#\ \mathrm{ftv}(\Gamma)}{\Gamma \vdash e : \forall \bar{\alpha}.\tau} \qquad \frac{\Gamma \vdash e : \forall \bar{\alpha}.\tau}{\Gamma \vdash e : [\vec{\tau}/\vec{\alpha}]\tau}$$

FIG. 2.4 – Le système de types de Damas et Milner

Théorème 2.4 *Soit α une variable de types arbitraire. Alors e est typable si et seulement si $[\![e : \alpha]\!]$ est satisfiable.* ◇

Dans le cas où $[\![e : \alpha]\!]$ est satisfiable, un typage principal de e peut être reconstruit de façon simple à partir d'une forme résolue de cette contrainte. Par exemple, une forme résolue de $[\![x + 1 : \alpha]\!]$ est $x = int \wedge \alpha = int$, d'où on peut déduire qu'un typage principal de $x+1$ est $(x : int, int)$. Je ne détaille pas ici cette construction.

Dans le cadre du λ-calcul simplement typé, donc, les deux réductions de l'inférence de types vers la résolution de contraintes proposées plus haut ne diffèrent que de façon cosmétique. Le véritable intérêt de la seconde réside dans le fait qu'elle seule se généralise élégamment dans le cadre de ML (§2.2.2).

2.2 Le système de Damas-Milner

Le système de types de ML, défini par Milner [6] puis par Damas et Milner [23], étend le λ-calcul simplement typé en permettant l'attribution d'un *schéma de types*, c'est-à-dire d'un type polymorphe, aux variables dont la définition est connue, c'est-à-dire aux variables liées par une nouvelle construction ***let*** :

$$e ::= \ldots \mid \textbf{\textit{let}}\ x = e\ \textbf{\textit{in}}\ e$$

Par opposition, les variables liées par λ, c'est-à-dire les arguments formels de fonctions, peuvent recevoir une valeur arbitraire à l'exécution, donc sont considérés comme n'ayant pas une définition connue, et devront rester monomorphes.

Un schéma de types est un type dans lequel certaines variables de types ont été universellement quantifiées :

$$\sigma ::= \forall \bar{\alpha}.\tau$$

Le système de types de Damas et Milner est obtenu en ajoutant à la définition du λ-calcul simplement typé (figure 2.1) les règles de la figure 2.4.

2.2.1 Les algorithmes $\mathcal{W}$ et $\mathcal{J}$

L'introduction du polymorphisme rend le problème de l'inférence de types plus complexe. D'un point de vue théorique, d'abord, sa complexité augmente sensiblement, puisqu'il devient DEXPTIME-complet [24, 25]. D'un point de vue plus pragmatique, ensuite, il a longtemps semblé difficile de donner une description claire de l'algorithme d'inférence de types pour le système de types de Damas et Milner. Les deux algorithmes équivalents donnés par Milner [6], $\mathcal{W}$ et $\mathcal{J}$, sont assez tortueux, car ils entremêlent des appels à un algorithme d'unification sous-jacent, des compositions et applications de substitutions, et des opérations de généralisation et instanciation de schémas de types, qui demandent de déterminer si certaines variables de types apparaissent ou non libres dans l'environnement courant. De plus, pour une raison que je ne m'explique pas, la postérité semble avoir retenu $\mathcal{W}$, alors que $\mathcal{J}$ est présenté – avec raison ! – par Milner comme plus simple. Toutefois, la présentation par Jones [26] de l'inférence de types pour Haskell contient en son cœur une implémentation de $\mathcal{J}$.

Afin d'illustrer ces propos, voici une définition de $\mathcal{J}$, issue d'une version de mes notes de cours de DEA [27], et relativement proche de celle de Milner. Elle s'appuie sur une fonction $\mathrm{mgu}(\cdot)$, qui à une conjonction de contraintes d'égalité associe un unificateur principal ou bien échoue. Je ne rappelle pas la définition classique de cette fonction, qui satisfait les propriétés auxiliaires suivantes : (i) toute variable de types fraîche vis-à-vis de C est fraîche vis-à-vis de $\mathrm{mgu}(C)$; et (ii) $\mathrm{mgu}(C)$ est idempotente.

$$
\begin{array}{lcl}
\textbf{\textit{fresh}} & = & \textbf{\textit{do}}\ \alpha \in V \\
 & & \textbf{\textit{do}}\ V \leftarrow V \setminus \{\alpha\} \\
 & & \textbf{\textit{return}}\ \alpha \\
\mathcal{J}(\Gamma \vdash x) & = & \textbf{\textit{let}}\ \forall\alpha_1 \ldots \alpha_n.\tau = \Gamma(x) \\
 & & \textbf{\textit{do}}\ \alpha'_1, \ldots, \alpha'_n = \textbf{\textit{fresh}}, \ldots, \textbf{\textit{fresh}} \\
 & & \textbf{\textit{return}}\ [\alpha'_i/\alpha_i]_{i=1}^{n}(\tau) \\
\mathcal{J}(\Gamma \vdash \lambda x.e_1) & = & \textbf{\textit{do}}\ \alpha = \textbf{\textit{fresh}} \\
 & & \textbf{\textit{do}}\ \tau_1 = \mathcal{J}(\Gamma; x : \alpha \vdash e_1) \\
 & & \textbf{\textit{return}}\ \alpha \to \tau_1 \\
\mathcal{J}(\Gamma \vdash e_1\, e_2) & = & \textbf{\textit{do}}\ \tau_1 = \mathcal{J}(\Gamma \vdash e_1) \\
 & & \textbf{\textit{do}}\ \tau_2 = \mathcal{J}(\Gamma \vdash e_2) \\
 & & \textbf{\textit{do}}\ \alpha = \textbf{\textit{fresh}} \\
 & & \textbf{\textit{do}}\ \phi \leftarrow \mathrm{mgu}(\phi(\tau_1) = \phi(\tau_2 \to \alpha)) \circ \phi \\
 & & \textbf{\textit{return}}\ \alpha \\
\mathcal{J}(\Gamma \vdash \textbf{\textit{let}}\ x = e_1\ \textbf{\textit{in}}\ e_2) & = & \textbf{\textit{do}}\ \tau_1 = \mathcal{J}(\Gamma \vdash e_1) \\
 & & \textbf{\textit{let}}\ \sigma = \overline{\forall}\, \mathrm{ftv}(\phi(\Gamma)).\phi(\tau_1) \\
 & & \textbf{\textit{return}}\ \mathcal{J}(\Gamma; x : \sigma \vdash e_2)
\end{array}
$$

FIG. 2.5 – L'algorithme $\mathcal{J}$

Pour plus de lisibilité, l'algorithme est présenté dans un style d'apparence impérative : il emploie deux variables globales ϕ et V. ϕ est la *substitution courante* : initialement l'identité, elle représente l'unificateur principal des équations résolues jusqu'ici. V est un ensemble arbitraire mais infini de variables de types : à chaque fois que l'algorithme a besoin d'une variable de types « fraîche, » il la choisit dans V et l'en retire. D'un point de vue formel, on peut considérer que ϕ et V sont deux paramètres et deux résultats implicites de l'algorithme, lequel devient alors purement fonctionnel. La syntaxe d'apparence impérative que j'emploie n'est en fait autre que la notation ***do*** de Haskell [28].

L'algorithme accepte un environnement Γ et une expression e. Il produit un type τ ou bien échoue. Il satisfait les invariants suivants : (i) ϕ est de la forme $\mathrm{mgu}(C)$ pour une certaine contrainte C ; (ii) aucune variable de V n'apparaît libre dans C, dans Γ, ou dans le type résultat τ. Ces invariants entraînent d'une part que ϕ est idempotente, d'autre part que V est frais vis-à-vis de ϕ. L'algorithme est défini par la figure 2.5.

La façon dont la substitution courante ϕ est mise à jour, à la quatrième ligne du cas de l'application, est telle que, si ϕ était initialement l'unificateur principal d'une contrainte C, alors il est, après la mise à jour, l'unificateur principal de la conjonction $C \wedge \tau_1 = \tau_2 \to \alpha$. Cela incite à penser que, au lieu de mémoriser une substitution courante ϕ, on pourrait mémoriser une conjonction d'équations courante C, et en repousser la résolution à plus tard. En fait, à quel moment a-t-on besoin de calculer ϕ et de l'appliquer ?

La réponse apparaît un peu plus bas, dans le cas de la construction ***let***. On y construit un schéma de types σ obtenu à partir du type $\phi(\tau_1)$ en quantifiant universellement toutes les variables de types qui n'apparaissent pas libres dans $\phi(\Gamma)$. (J'emploie ici la notation $\overline{\forall}\bar{\alpha}$ pour quantifier toutes les variables de types *à l'exception de* $\bar{\alpha}$.) On voit mal ici comment construire σ sans exhiber ϕ, donc sans résoudre C. C'est peut-être pour cette raison qu'on a longtemps considéré que, dans le cas de ML, génération et résolution de contraintes devaient alterner et ne pouvaient pas être entièrement séparées. Or, on verra bientôt (§2.2.2) que cette croyance n'avait pas de véritable raison d'être.

Que dire de la preuve de l'algorithme $\mathcal{J}$? On peut établir successivement les propriétés suivantes :

Théorème 2.5 (Correction) *Si $\mathcal{J}(\Gamma \vdash e)$ termine dans l'état (ϕ, V) et renvoie τ, alors $\phi(\Gamma) \vdash e : \phi(\tau)$ est un jugement.* ◇

Théorème 2.6 (Complétude) *Soit Γ un environnement de typage. Soit (ϕ_0, V_0) un état satisfaisant les invariants de l'algorithme. Supposons donnés θ_0 et τ_0 tels que $\theta_0\phi_0(\Gamma) \vdash e : \tau_0$ soit un jugement. Alors, l'exécution de $\mathcal{J}(\Gamma \vdash e)$ à partir de l'état initial (ϕ_0, V_0) réussit. Soient (ϕ_1, V_1) son état final et τ_1*

le type renvoyé. Alors il existe une substitution θ_1 telle que $\theta_0\phi_0$ et $\theta_1\phi_1$ coïncident en dehors de V_0 et telle que τ_0 s'écrit $\theta_1\phi_1(\tau_1)$. ◇

Si le premier énoncé ci-dessus est simple et se prouve aisément par induction structurelle, il n'en va pas de même du second. Cet énoncé est difficile à déchiffrer. Comme on peut l'imaginer, sa preuve est lourde [27] et, ce qui est plus grave, la lecture de cette preuve n'éclaire en rien, à mon avis, le fonctionnement de l'algorithme.

À titre de remarque historique, il est intéressant de noter que, pendant de nombreuses années, l'unique preuve de complétude de $\mathcal{W}$ est restée celle de Damas [29], qui n'a été que très peu diffusée. Il semble qu'il a fallu attendre les années 1990 pour que cette preuve soit plus largement publiée [30] puis mécanisée [31–33].

On peut également noter que les preuves des algorithmes $\mathcal{W}$, $\mathcal{J}$, ou d'une variante telle que $\mathcal{M}$ [34] se révèlent suffisamment différentes dans la forme, même si le fond est identique, pour ne guère pouvoir partager que quelques lemmes préliminaires. L'approche à base de contraintes que je décris plus loin (§2.2.2) est supérieure en ce que la preuve du générateur de contraintes est réalisée une fois pour toutes. Les différents algorithmes classiques, $\mathcal{W}$, $\mathcal{J}$ et $\mathcal{M}$, correspondent alors simplement à diverses *stratégies* de résolution de contraintes, dont la preuve de correction n'est pas difficile.

Posons quelques définitions. Un typage (Γ', τ) est *relatif à* Γ si et seulement si sa première composante Γ' est instance de Γ. Un typage de e est *principal relativement à* Γ si et seulement si il est relatif à Γ et tout typage de e relatif à Γ en est une instance. Alors, on peut tirer des deux théorèmes précédents la conclusion suivante.

Corollaire 2.7 (Principalité) *L'évaluation de $\mathcal{J}(\Gamma \vdash e)$ réussit si et seulement si e admet un typage relatif à Γ. De plus, si ϕ_1 et τ_1 sont les résultats produits par l'algorithme, alors $(\phi_1(\Gamma), \phi_1(\tau_1))$ est un typage de e et est principal relativement à Γ.* ◇

Il est instructif de comparer l'énoncé de ce corollaire avec celui du corollaire 2.3. Dans le cas du λ-calcul simplement typé, l'algorithme d'inférence acceptait pour seul argument l'expression e, et en produisait un typage principal ou bien échouait. Dans le cas de ML, l'algorithme $\mathcal{J}$ attend non seulement e mais également un environnement Γ, et produit un typage de e principal *relativement à* Γ, c'est-à-dire principal uniquement parmi les typages de e dont la première composante est instance de Γ. La recherche ne se fait donc que dans une partie de l'espace des typages de e. Pour cette raison, on dit traditionnellement que le système de types de Damas et Milner n'a pas la propriété des *typages principaux*, mais que néanmoins il a la propriété des *types principaux*. Pour plus de détails, on pourra consulter [16, 17].

On pourrait bien sûr fournir à l'algorithme un environnement Γ constitué, comme dans le corollaire 2.2, de variables de types distinctes. Si tout environnement était instance d'un tel Γ, comme c'est le cas pour le λ-calcul simplement typé, alors l'algorithme produirait un typage principal. Malheureusement, tel n'est pas le cas, car un schéma non trivial n'est pas instance d'une variable de types : par exemple, le schéma de types $\forall\beta.\beta \to \beta$ n'est pas instance de α. (Le *type* $\beta \to \beta$ est instance de α.) En d'autres termes, en fournissant à l'algorithme $\mathcal{J}$ un tel environnement Γ, on lui impose en fait d'attribuer un type monomorphe à toutes les variables qui apparaissent libres dans e. Pour que l'une d'elles puisse se comporter de façon polymorphe, il faudrait que l'environnement initial Γ attribue à cette variable un *schéma* de types non trivial. Pour résumer, l'algorithme est incapable d'inférer la nécessité d'attribuer un type polymorphe à une variable libre.

2.2.2 Une approche à base de contraintes

Les algorithmes $\mathcal{W}$ et $\mathcal{J}$ emploient l'algorithme d'unification comme une boîte noire qui, étant donnés deux types arbitraires, produit leur unificateur principal (*most general unifier*). Les algorithmes sont présentés de telle façon qu'un appel à $\mathrm{mgu}(\cdot)$ semble nécessaire au moins à chaque nœud ***let***. Chez Milner, donc, comme chez Hindley (§2.1), on a alternance implicite entre génération et résolution de contraintes. Or, pour des raisons de modularité et de simplification conceptuelle, on souhaiterait séparer ces deux phases.

$$\begin{array}{rcl}
[\![x : \tau]\!] & = & x \preceq \tau \\
[\![\lambda x.e : \tau]\!] & = & \exists\alpha_1\alpha_2.(\textbf{\textit{def}}\ x : \alpha_1\ \textbf{\textit{in}}\ [\![e : \alpha_2]\!] \wedge \alpha_1 \to \alpha_2 = \tau) \\
[\![e_1\, e_2 : \tau]\!] & = & \exists\alpha.([\![e_1 : \alpha \to \tau]\!] \wedge [\![e_2 : \alpha]\!]) \\
[\![\textbf{\textit{let}}\ x = e_1\ \textbf{\textit{in}}\ e_2 : \tau]\!] & = & \textbf{\textit{let}}\ x : \forall\alpha[[\![e_1 : \alpha]\!]].\alpha\ \textbf{\textit{in}}\ [\![e_2 : \tau]\!]
\end{array}$$

FIG. 2.6 – Génération de contraintes pour le système de types de Damas et Milner

Il est intéressant de noter que les présentations classiques des extensions de ML à base de contraintes, comme HM(X) [35], souffrent du même problème, quoique de façon moins ostensible. Dans HM(X), la création d'un schéma de types n'exige pas, en apparence, la résolution de la contrainte courante, car les schémas de types sont de la forme $\forall\bar{\alpha}[C].\tau$, où la contrainte C n'est pas nécessairement sous forme résolue. Mais, en réalité, parce que la contrainte C sera *dupliquée* à chaque fois qu'on voudra obtenir une instance du schéma ainsi créé, il est important que C soit résolue et simplifiée préalablement à la création du schéma. La présentation originale de l'algorithme d'inférence de types pour HM(X) fait d'ailleurs alterner génération et résolution [35].

La solution que je propose à présent, qui est tirée de [1], emploie des contraintes ***def***, similaires à celles introduites en §2.1.2. Elle est applicable aussi bien dans le cas du système de types de Damas et Milner que dans le cas de HM(X). Je me limite ici au premier, et dirai plus loin quelques mots du second (§2.3).

L'idée est d'enrichir le langage de contraintes considéré en §2.1.2 en autorisant une variable x à dénoter non seulement un type, mais plus généralement un schéma de types. La syntaxe des contraintes et des schémas de types devient alors :

$$\begin{array}{rcl}
C & ::= & \tau = \tau \mid C \wedge C \mid \exists\alpha.C \mid x \preceq \tau \mid \textbf{\textit{def}}\ x : \sigma\ \textbf{\textit{in}}\ C \\
\sigma & ::= & \forall\bar{\alpha}[C].\tau
\end{array}$$

L'interprétation logique des contraintes se fait à présent vis-à-vis d'une valuation ϕ qui à chaque variable de types α associe un élément du modèle dans lequel sont interprétés les types, comme précédemment, et vis-à-vis d'une valuation ψ qui à chaque variable x associe un *ensemble* de tels éléments. Un schéma de types est en effet interprété comme un ensemble d'éléments du modèle. La contrainte $x \preceq \tau$, que l'on peut lire « le type τ est instance du schéma x, » est satisfaite par ϕ et ψ si et seulement si $\phi\tau$ est élément de ψx. La contrainte ***def*** $x : \sigma$ ***in*** C est satisfaite par ϕ et ψ si et seulement si C est satisfaite par ϕ et $\psi[x \mapsto {}^{\psi}_{\phi}(\sigma)]$, où l'interprétation ${}^{\psi}_{\phi}(\sigma)$ d'un schéma σ est définie comme suit : si σ s'écrit $\forall\bar{\alpha}[C].\tau$, alors son interprétation est l'ensemble des $\phi'\tau$, où ϕ et ϕ' coïncident hors de $\bar{\alpha}$ et ϕ' et ψ satisfont C.

Encore une fois, l'effet des définitions ci-dessus, qui peuvent sembler techniques au premier abord, est uniquement de valider la loi d'équivalence

$$\textbf{\textit{def}}\ x : \sigma\ \textbf{\textit{in}}\ C \equiv [\sigma/x]C$$

La construction ***def*** n'est donc qu'une forme de *substitution explicite*. Pour donner un sens à la loi ci-dessus, il faut toutefois préciser la signification de la pseudo-contrainte $\sigma \preceq \tau$, que l'on peut lire « le type τ est instance du schéma σ, » qui peut apparaître lorsqu'on remplace x par σ dans une contrainte de la forme $x \preceq \tau$. Pour cela, il suffit d'indiquer comment cette contrainte est interprétée : $\sigma \preceq \tau$ est satisfaite par ϕ et ψ si et seulement si $\phi\tau$ est élément de l'ensemble ${}^{\psi}_{\phi}(\sigma)$. De façon équivalente, on peut également considérer cette contrainte comme un sucre syntaxique : en effet, si σ s'écrit $\forall\bar{\alpha}[C].\tau'$, et si $\bar{\alpha}$ est frais vis-à-vis de τ, alors $\sigma \preceq \tau$ est équivalente à $\exists\bar{\alpha}.(C \wedge \tau = \tau')$. En d'autres termes, le type τ est une instance du schéma contraint $\forall\bar{\alpha}[C].\tau'$ si, pour une certaine valeur des variables $\bar{\alpha}$ satisfaisant C, τ' coïncide avec τ.

Grâce à ce langage de contraintes étendu, il est à présent possible non seulement d'exprimer des schémas de types contraints, ce qui permet, comme dans HM(X), de construire un schéma de types sans devoir au préalable appeler l'algorithme de résolution, mais également de faire référence à un tel schéma à travers une variable x, ce qui évite de devoir le *dupliquer* immédiatement à chaque fois que l'on désire en obtenir une instance.

Nous sommes donc en mesure d'exprimer un algorithme de génération de contraintes pour le système de types de Damas et Milner dans le même style que celui exposé en §2.1.2 (figure 2.3). Le nouvel algorithme est donné par la figure 2.6. Les trois premières lignes sont identiques à celles de la figure 2.3, modulo le remplacement de $x = \tau$ par $x \preceq \tau$, pour refléter le fait que x dénote à présent un schéma et que τ peut alors en être une instance arbitraire. La principale nouveauté réside dans la quatrième ligne. En première lecture, on peut considérer la construction ***let***, non encore définie, comme synonyme de ***def***. L'algorithme construit d'abord le schéma de types $\sigma_1 = \forall\alpha[[\![e_1 : \alpha]\!].\alpha$, où α est une variable de types arbitraire, schéma qui est principal pour e_1. La contrainte qui exprime que e_2 est bien typée, à savoir $[\![e_2 : \tau]\!]$, peut contenir des occurrences libres de x. Elle est donc placée dans le contexte ***let*** $x : \sigma_1$ ***in*** $[]$, de façon à ce que ces occurrences dénotent le schéma σ_1.

En réalité, si ***let*** était défini comme synonyme de ***def***, cet algorithme de génération de contraintes ne serait pas tout-à-fait correct. Dans le cas particulier où x n'apparaît pas libre dans e_2, la contrainte ***def*** $x : \forall\alpha[[\![e_1 : \alpha]\!].\alpha$ ***in*** $[\![e_2 : \tau]\!]$ est équivalente à $[\![e_2 : \tau]\!]$, et ne garantit donc pas que e_1 est bien typée. On introduit donc la construction ***let*** $x : \sigma$ ***in*** C en tant que sucre syntaxique pour ***def*** $x : \sigma$ ***in*** $(\exists\alpha.x \preceq \alpha \wedge C)$. On peut alors vérifier que la contrainte ***let*** $x : \forall\alpha[[\![e_1 : \alpha]\!].\alpha$ ***in*** $[\![e_2 : \tau]\!]$ implique $\exists\alpha.[\![e_1 : \alpha]\!]$, et garantit donc que e_1 est bien typée.

On peut démontrer la correction et la complétude de cet algorithme de génération de contraintes vis-à-vis de la spécification du système de types de Damas et Milner. En voici un énoncé :

Théorème 2.8 *Soit Γ un environnement de domaine $\mathrm{fv}(e)$. L'expression e est bien typée relativement à Γ si et seulement si* ***def*** Γ ***in*** $\exists\alpha.[\![e : \alpha]\!]$ *est satisfiable.* ◇

Il est important de noter que l'algorithme de génération de contraintes est de complexité linéaire. En d'autres termes, la décomposition modulaire entre génération et résolution de contraintes peut être effectivement employée comme technique d'implantation, et n'entraîne aucun surcoût asymptotique.

L'emploi des constructions ***def*** ou ***let*** a précisément pour but de permettre l'expression d'une contrainte de taille linéaire. (Rappelons, en effet, que leur élimination par expansion naïve pourrait provoquer une croissance exponentielle de la taille de la contrainte.) Cette technique semble due à Müller [36]. Elle a indépendamment été employée par Gustavsson et Svenningsson [37], dans un cadre où le seul prédicat de base est une relation de sous-typage entre variables dénotant des atomes. Gustavsson et Svenningsson démontrent que, dans ce cadre précis, la stratégie qui consiste à simplifier le corps d'une définition ***let*** avant de la dupliquer mène à un algorithme de complexité cubique. L'explosion exponentielle est donc entièrement évitée. Malheureusement, ce résultat ne s'applique pas au cas où les contraintes de base sont des équations entre termes, puisque le problème de satisfiabilité est alors nécessairement DEXPTIME-difficile, de par l'existence même d'une réduction de l'inférence de types pour le système de types de Damas et Milner vers lui. Néanmoins, McAllester [38] propose un résultat de complexité original : sous la double hypothèse que les types inférés sont de taille bornée et que le programmeur n'imbrique pas les constructions ***let*** à gauche au-delà d'une profondeur bornée, les contraintes engendrées peuvent être résolues en temps linéaire. Cette hypothèse peut être discutée ; néanmoins, ce résultat semble expliquer pourquoi l'inférence de types pour le système de types de Damas et Milner peut être effectuée de façon « efficace en pratique. »

On pourrait mettre en doute l'intérêt de l'approche à base de contraintes en argumentant que mon langage de programmation n'a que quatre constructions (variable, abstraction, application et ***let***), tandis que mon langage de contraintes en a autant, voire plus (équation, conjonction, quantification existentielle, instanciation et construction ***let***). Cela signifie-t-il que l'emploi d'un langage de contraintes n'éclaire en rien le problème de l'inférence de types ? Non. L'ajout de produits et de sommes, de types algébriques, de références, d'exceptions, pour citer quelques-uns des traits du langage ML, ne nécessite aucune extension du langage de contraintes. L'ajout du filtrage et de certaines formes d'annotations de types en demande deux extensions mineures [1]. Dans le cas où le langage de programmation considéré est un langage réel, donc, le passage par le langage de contraintes permet une réelle simplification du problème.

2.2.3 Quelques mots de la résolution des contraintes

Un algorithme de résolution de contraintes est généralement présenté sous la forme d'un système de réécriture. Pour en démontrer la correction, on établit trois propriétés, à savoir : (i) toute étape de réécriture préserve l'interprétation logique de la contrainte ; (ii) le système de réécriture est fortement normalisant ; et (iii) il est immédiat de décider si une forme normale (dite forme *résolue*) est satisfiable ou non.

L'un des intérêts de l'approche à base de contraintes est la possibilité de définir plusieurs algorithmes de résolution distincts, qui correspondent souvent à différentes stratégies au sein d'un même système de réécriture. Ces différents algorithmes partagent alors la même preuve de correction.

Le langage de contraintes considéré ici est constitué d'un langage noyau, contenant les équations entre termes, la conjonction et la quantification existentielle, et enrichi par un mécanisme de substitution explicite, représenté par la construction ***let*** et par les contraintes d'instanciation. Il est donc naturel de définir un système de réécriture, de façon modulaire, comme la combinaison d'un solveur pour le langage noyau et d'un jeu de règles dédiées à la gestion des substitutions explicites. Le premier n'est autre qu'un algorithme d'unification du premier ordre, et pourra être arbitraire ; on doit néanmoins supposer connue la structure de ses formes résolues. Le second effectue les opérations connues sous les noms de « généralisation » et « instanciation » dans les implantations classiques de ML.

La façon la plus naturelle de traiter les contraintes ***let*** est celle implicitement employée par les algorithmes $\mathcal{W}$ et $\mathcal{J}$ de Milner, ainsi que par Rémy [39], Müller [36] ou Gustavsson et Svenningsson [37] : étant donnée une contrainte ***let*** $x : \sigma$ ***in*** C, on simplifiera d'abord le schéma σ, afin de lui donner la forme la plus compacte possible, avant d'éliminer la construction ***let*** en remplaçant toutes les occurrences de x par σ dans C. Le fait de simplifier σ préalablement à sa duplication permet une économie. Simplifier un schéma consiste, au minimum, à résoudre la contrainte qu'il contient. On peut néanmoins aller plus loin, notamment en diminuant le nombre de variables de types universellement quantifiées, puis en déplaçant certaines contraintes de l'intérieur du schéma vers l'extérieur de la construction ***let***, ce qui évite de les dupliquer. Ces idées, dues à Rémy [39], sont développées en détail dans [1, §8]. Une implantation efficace de ces techniques nécessite d'associer à chaque variable de types un *rang* entier. Ce mécanisme, imaginé par Rémy [39] et redécouvert par McAllester [38], peut être compris en termes logiques : le rang d'une variable de types indique le point où elle est liée. Diminuer son rang revient alors à déplacer son lieur vers le haut au sein de la contrainte courante, donc à effectuer une extrusion de portée.

Une stratégie de résolution alternative consiste à traiter les contraintes ***let*** différemment selon que la variable x concernée est liée par λ ou par ***let*** dans le programme initial. Dans le premier cas, la contrainte ***def*** $x : \tau$ ***in*** C est traitée par élimination de la construction ***def*** *après* résolution de la contrainte C. Dans le second cas, la contrainte ***let*** $x : \sigma$ ***in*** C est traitée par simplification de σ puis élimination de la construction ***let*** *préalablement* à l'analyse de C, comme plus haut. Cette approche alternative correspond à l'algorithme PTL de Mitchell [40], et a été employée dans certains autres travaux [41–43]. Un de ses intérêts, selon Chitil [43], serait de faciliter la localisation interactive des erreurs de typage, en empêchant le solveur de contraintes de propager l'information de façon *transversale* vis-à-vis de la structure d'arbre du programme.

2.2.4 Discussion

L'approche modulaire de l'inférence de types peut être critiquée. Voici par exemple ce qu'écrivait l'un des relecteurs de [1] à propos de ma présentation du système de types de Damas et Milner :

> *Votre approche sépare joliment le problème en deux : la génération et la résolution de contraintes. Cependant, il semble que cette division revient un peu à séparer l'Empire State Building en deux : le bouton de porte et le reste du bâtiment. Le générateur de contraintes est presque trivial tandis que le solveur semble vraiment plutôt compliqué.*

Il me semble qu'on peut argumenter, en réponse, que si la génération de contraintes semble « triviale, » c'est justement parce qu'elle a été bien comprise, et exprimée dans un formalisme aussi naturel et concis que possible. De ce fait, et par contraste, la description du solveur de contraintes peut sembler complexe, surtout si l'on s'attache à décrire de façon formelle une implantation efficace. Néanmoins, la

spécification du solveur reste simple, parce que la satisfiabilité d'une contrainte a été définie de façon directe, à l'aide d'un modèle logique.

On pourra comparer mon approche à une autre plus classique, comme les algorithmes $\mathcal{W}$ et $\mathcal{J}$ de Milner [6] : bien que l'accent n'y soit pas mis sur l'efficacité, et bien que l'algorithme d'unification y soit employé comme une boîte noire séparée, leur définition est loin d'être simple.

En conclusion, je prétends donc que s'il est vrai, à l'heure actuelle, que le gros de la difficulté réside dans la conception d'un solveur de contraintes efficace, cela est aussi le fruit de l'effort théorique qui a été fourni pour offrir, en amont, un cadre général au typage à base de contraintes.

2.3 Le système HM(X)

J'ai jusqu'ici employé les contraintes uniquement comme langage intermédiaire pour l'inférence de types. Or, elles ont également et simultanément été utilisées pour augmenter l'expressivité de la discipline de Hindley et Milner. En effet, il doit être clair à présent que celle-ci se prête à une généralisation : plutôt que d'employer un langage de contraintes fondé exclusivement sur l'égalité entre types, pourquoi ne pas exploiter d'autres prédicats de base ? On peut par exemple songer à remplacer la relation d'égalité par une relation d'ordre, communément appelée *sous-typage*, pour plus de flexibilité. On peut également songer à faire apparaître des prédicats arbitraires dans les schémas de types associés à certaines opérations primitives, comme dans le cas des *type classes* de Haskell [28, 44–46].

L'introduction de sous-typage demande une très légère modification de l'interprétation des schémas de types : un schéma sera interprété non plus par l'ensemble de ses instances, mais par la clôture supérieure de celui-ci vis-à-vis de la relation de sous-typage. Ceci garantit que $x \preceq \tau \wedge \tau \leq \tau'$ entraîne $x \preceq \tau'$. Les règles de génération de contraintes (figure 2.6) reçoivent également une modification mineure, à savoir le remplacement du symbole $=$ par $\leq$ dans la seconde ligne. L'introduction de prédicats de base autres que le sous-typage, quant à elle, ne nécessite aucune modification des règles de génération de contraintes.

La généralisation de l'algorithme d'inférence de types est donc immédiate, au moins en ce qui concerne le générateur de contraintes. (L'adaptation du solveur peut être une autre paire de manches, car la conception d'un solveur efficace est habituellement très dépendante des détails de la syntaxe et de l'interprétation des contraintes.) Naturellement, il faut généraliser également la spécification du système en termes de règles de typage, de manière à ce que le résultat de correction et de complétude du générateur de contraintes vis-à-vis de la spécification du système (théorème 2.8) soit préservé.

Pour cela, on enrichit la forme des pré-jugements de typage en leur ajoutant une nouvelle hypothèse : une contrainte. Les pré-jugements $\Gamma \vdash e : \sigma$ du système de types de Damas et Milner sont donc remplacés par des pré-jugements de la forme $C, \Gamma \vdash e : \sigma$. De plus, les schémas de types σ sont maintenant contraints : leur forme est celle introduite en §2.2.2.

Rétrospectivement, on peut considérer l'absence de toute contrainte dans la spécification du système de types de Damas et Milner comme un accident syntaxique. Du fait que toute contrainte d'unification satisfiable admet un unificateur principal, l'information contenue dans une contrainte C se résume à une substitution ϕ, que l'on peut appliquer à Γ et à σ. De ce fait, faire explicitement figurer une contrainte dans les jugements du système de types de Damas et Milner ne procure aucune expressivité supplémentaire au système. Cependant, lorsqu'on généralise le langage de contraintes, cette propriété est en général perdue. C'est pourquoi il est alors nécessaire d'expliciter la présence d'une contrainte dans les jugements, et, par suite, dans les schémas de types.

Les systèmes de types dont la spécification même fait apparaître la notion de contrainte sont nombreux [10, 35, 41, 47–59]. Une version que l'on peut considérer comme définitive, car suffisamment élégante d'un point de vue logique, est le système HM(X) [35], dont la figure 2.7 propose une définition.

Les six premières règles sont celles du système de types de Damas et Milner, généralisées pour admettre une hypothèse C. La septième règle autorise l'emploi du sous-typage, et est donc communément appelée règle SUB ou règle de sous-typage. Le symbole $\Vdash$ y dénote l'implication de contraintes (*entailment*). La dernière règle permet l'emploi de la quantification existentielle pour expliciter le fait que certaines variables de types $\bar{\alpha}$ sont locales à une sous-dérivation.

$$\frac{\Gamma(x) = \sigma \qquad C \Vdash \exists\sigma}{C, \Gamma \vdash x : \sigma} \qquad \frac{C, \Gamma; x : \tau_1 \vdash e : \tau_2}{C, \Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad \frac{C, \Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad C, \Gamma \vdash e_2 : \tau_1}{C, \Gamma \vdash e_1\, e_2 : \tau_2}$$

$$\frac{C, \Gamma \vdash e_1 : \sigma \qquad C, \Gamma; x : \sigma \vdash e_2 : \tau}{C, \Gamma \vdash \textbf{\textit{let}}\ x = e_1\ \textbf{\textit{in}}\ e_2 : \tau} \qquad \frac{C \wedge D, \Gamma \vdash e : \tau \qquad \bar{\alpha}\ \#\ \mathrm{ftv}(C, \Gamma)}{C \wedge \exists\bar{\alpha}.D, \Gamma \vdash e : \forall\bar{\alpha}[D].\tau}$$

$$\frac{C, \Gamma \vdash e : \forall\bar{\alpha}[D].\tau}{C \wedge D, \Gamma \vdash e : \tau} \qquad \frac{C, \Gamma \vdash e : \tau \qquad C \Vdash \tau \leq \tau'}{C, \Gamma \vdash e : \tau'} \qquad \frac{C, \Gamma \vdash e : \sigma \qquad \bar{\alpha}\ \#\ \mathrm{ftv}(\Gamma, \sigma)}{\exists\bar{\alpha}.C, \Gamma \vdash e : \sigma}$$

FIG. 2.7 – Le système HM(X)

La règle de sous-typage est l'unique règle où la présence d'une contrainte dans les jugements est exploitée de façon essentielle : la contrainte C est ici l'hypothèse qui affirme que τ est effectivement sous-type de τ'. (Ces deux types pouvant contenir des variables de types, l'assertion $\tau \leq \tau'$ n'aurait guère de sens dans l'absolu.) Les autres règles se contentent de transmettre cette hypothèse. Dans la règle de généralisation, le fragment D de la contrainte courante qui concerne les variables généralisées $\bar{\alpha}$ devient partie intégrante du schéma de types $\forall\bar{\alpha}[D].\tau$ nouvellement créé. La manœuvre inverse est effectuée lors de l'instanciation d'un schéma de types.

Un pré-jugement $C, \Gamma \vdash e : \sigma$ est un jugement si et seulement si il est dérivable d'après les règles de la figure 2.7 et C est satisfiable. Une expression est *bien typée* dans l'environnement Γ si et seulement si il existe un jugement de la forme $C, \Gamma \vdash e : \sigma$, ou bien – c'est équivalent – si et seulement si il existe un jugement de la forme $C, \Gamma \vdash e : \tau$.

La définition originale de HM(X) [35] suppose que le langage de contraintes inclut les contraintes de sous-typage, la conjonction et la quantification existentielle. Toutefois, rien n'empêche d'y ajouter les contraintes ***let*** que j'ai définies plus haut. Il est alors possible, si on le souhaite, de reformuler les règles de typage de telle sorte que l'environnement Γ n'est plus nécessaire. Cette reformulation est effectuée dans la version longue de [1], non encore publiée, et est ici laissée en exercice au lecteur intéressé.

On peut démontrer que les règles de génération de contraintes évoquées plus haut sont correctes et complètes vis-à-vis de la spécification du système HM(X). L'énoncé, dont la démonstration n'est pas difficile, est le suivant :

Théorème 2.9 *$C, \Gamma \vdash e : \tau$ est équivalent à $C \Vdash$ **def** Γ **in** $[\![e : \tau]\!]$.* ◇

Il découle de ceci qu'une expression close e est bien typée si et seulement si la contrainte $\exists\alpha.[\![e : \alpha]\!]$ est satisfiable.

L'intérêt principal du système HM(X) est son indépendance vis-à-vis de la syntaxe et de l'interprétation des contraintes, dénotés par le paramètre formel X. La définition du système, sa preuve de sûreté, et la preuve de correction de l'algorithme de génération de contraintes en sont indépendants, modulo quelques hypothèses concernant par exemple l'interaction entre relation de sous-typage et constructeur de types « flèche. » La notation HM(X) représente donc en réalité une *famille* de systèmes de types, partageant une définition et quelques propriétés élémentaires communes. Celles-ci constituent un cadre théorique général dans lequel on peut situer bon nombre de développements concrets. Les travaux décrits dans les prochains chapitres de ce mémoire seront situés dans le cadre de HM(X) ou bien en constitueront des variations ou extensions.

On vérifie aisément que tout membre de la famille HM(X) est une extension du système de types de Damas et Milner.

Le membre le plus simple de cette famille est obtenu lorsque le sous-typage est interprété comme l'égalité entre types, et lorsqu'aucun autre prédicat de base est disponible. Je note HM$(=)$ le système ainsi spécialisé. Alors, les contraintes sont des problèmes d'unification, admettent des unificateurs principaux, et on peut démontrer [1] que le système HM$(=)$ coïncide avec le système de types de Damas et Milner. En d'autres termes, lorsqu'on ne manipule que des contraintes d'égalité, le système HM(X) n'est qu'une reformulation du système de types de Damas et Milner, où l'accent est mis sur la notion

de contrainte, et non sur la notion de substitution. Une substitution n'est en fait qu'une représentation particulière de la forme résolue d'une contrainte satisfiable. On peut argumenter qu'il est plus facile de raisonner en termes de contraintes, et de leur combinaison par conjonction et par quantification existentielle, plutôt qu'en termes de substitutions, et de leur combinaison par composition et par restriction. De plus, les multi-équations [14, 60] permettent de modéliser l'état de l'algorithme d'unification comme un graphe, et non comme un terme, ce qui est nécessaire pour une implantation efficace.

Quelques systèmes plus complexes sont obtenus en conservant l'interprétation du sous-typage en tant qu'égalité, mais en adoptant une notion d'égalité plus riche. Par exemple, interpréter les types dans un modèle constitué d'arbres infinis donne lieu à une extension du système de types de Damas et Milner par des types *équirécursifs*, selon la terminologie d'Abadi et Fiore [61]. D'autres systèmes intéressants peuvent être obtenus en interprétant les types dans un modèle qui valide une théorie équationnelle non triviale, par exemple celle des rangées (§3.1.3) ou celles des dimensions physiques [62–64].

Enfin, on peut obtenir des systèmes beaucoup plus complexes en interprétant la relation de sous-typage de façon non triviale ou en introduisant des prédicats de base autres que la relation de sous-typage. Une relation de sous-typage *non structurelle* sera utilisée au chapitre 3, en combinaison avec des rangées et une forme simple de *contraintes conditionnelles*. Des relations de sous-typage *structurelles* apparaîtront aux chapitres 4 et 5, accompagnées, au chapitre 5, de prédicats auxiliaires. Enfin, au chapitre 6, je présenterai une extension de HM(X) où les contraintes font intervenir quantification universelle et implication. Ainsi, une pléthore de systèmes de types « cousins » appartiennent à la famille HM(X) et bénéficient de ce cadre théorique commun.

2.4 Sûreté du typage pour HM(X)

Je termine ce chapitre par quelques mots de la preuve de sûreté du typage pour HM(X). Prouver qu'un programme bien typé dans HM(X) ne peut provoquer aucune erreur à l'exécution n'est pas particulièrement difficile, pourvu qu'on ait une expérience suffisante de ces questions. (En particulier, il est vital de ne pas se laisser importuner par des questions d'α-conversion. On ne peut que recommander à ce sujet la lecture des travaux de Gabbay et Pitts [15, 65, 66].) Néanmoins, il est intéressant de comparer les diverses techniques de preuve existantes. Ci-dessous, j'en énumère quatre. Chacune d'elles présente des avantages et inconvénients distincts, et si ces différences ne sont guère significatives dans le cas d'un langage aussi réduit que le λ-calcul étendu avec la construction ***let***, elles peuvent prendre de l'ampleur lorsqu'on s'intéresse à des langages plus riches et syntaxiquement plus lourds. Je pense, par exemple, au join-calcul [67, 68], pour lequel j'ai proposé un système de types à base de contraintes, en collaboration avec Sylvain Conchon [3].

2.4.1 Sûreté dénotationnelle

La première preuve de sûreté pour HM(X), réalisée par Odersky, Sulzmann et Wehr [35, 69], est relative à une sémantique dénotationnelle du λ-calcul, où les types sont interprétés par des idéaux d'un domaine de Scott [70], c'est-à-dire, en première approximation, par des ensembles de valeurs. Le principal énoncé, simplifié au cas où l'expression e considérée est close, est alors le suivant :

Théorème 2.10 (Sûreté dénotationnelle) *Soit $C, \varnothing \vdash e : \tau$ un jugement. Soit ϕ une solution de C. Alors la dénotation de e appartient à la dénotation de $\phi\tau$.* ◇

Par construction, la dénotation d'un type ne peut contenir l'élément **W**, qui représente une erreur à l'exécution. On en déduit donc qu'aucune expression close bien typée n'a **W** pour dénotation.

La preuve du théorème 2.10, généralisé au cas des expressions non closes, se fait par induction structurelle sur la dérivation d'un jugement de typage. Elle est donc très directe. En particulier, chaque règle de typage, dirigée par la syntaxe ou non, donne lieu à un cas de preuve distinct et indépendant. Comme la présentation de HM(X) comporte quatre règles non dirigées par la syntaxe, ce fait est appréciable.

L'inconvénient de cette approche est de reposer sur une sémantique dénotationnelle, dont la définition n'est pas toujours aisée. De nombreux langages ne sont dotés que d'une sémantique opérationnelle, dont la définition et la compréhension sont plus faciles. Par ailleurs, l'interprétation des types en tant

qu'idéaux est fragile, au sens où elle doit être considérablement modifiée si l'on souhaite introduire, par exemple, des types récursifs [71].

2.4.2 Sûreté syntaxique

Une seconde méthode, tout aussi classique, consiste à suivre l'approche syntaxique de Wright et Felleisen [72]. On adopte alors une sémantique opérationnelle à petits pas, munie d'une distinction entre *valeurs*, expressions *réductibles* et expressions *bloquées*. On démontre d'une part que toute expression close bien typée est soit réductible soit une valeur, d'autre part que la réduction préserve le typage. Ces deux propriétés sont connues sous les noms de *progrès* et *auto-réduction*. La seconde est celle dont la preuve est en général la plus délicate et la plus intéressante. L'énoncé en est le suivant :

Théorème 2.11 (Auto-réduction) *Si $C, \Gamma \vdash e : \sigma$ est dérivable et si e se réduit en e', alors $C, \Gamma \vdash e' : \sigma$ est dérivable.* ◇

La preuve se fait par induction structurelle sur la deuxième hypothèse, à savoir $e \to e'$. Elle présente donc un cas pour chacune des règles qui définissent la relation de réduction $\to$. Dans chaque cas, on doit analyser la structure du membre gauche e, en déduire la structure de la dérivation de $C, \Gamma \vdash e : \sigma$, puis utiliser l'ensemble des hypothèses ainsi obtenues pour construire une dérivation de $C, \Gamma \vdash e' : \sigma$. Lorsque les règles de réduction mettent en jeu un membre gauche e complexe, et lorsque le système de types comporte de nombreuses règles non dirigées par la syntaxe, ce processus peut être lourd. J'en ai fait l'expérience en écrivant une preuve d'auto-réduction (non publiée) pour le système JOIN(X) [3]. La preuve ne comporte essentiellement qu'un cas, car le join-calcul n'a qu'une règle de réduction, hormis les règles de passage au contexte. Étant donnée la complexité des termes e et e' mis en jeu, la déconstruction de l'assertion de typage initiale et la construction d'une nouvelle assertion se font en une bonne cinquantaine d'étapes ! De plus, elles exigent plusieurs lemmes de normalisation, dont l'objet est de contrôler l'emploi des règles de typage non dirigées par la syntaxe. Cette preuve est donc lourde et non modulaire. Elle illustre, dans une certaine mesure, les limitations de l'approche purement syntaxique, lorsque le langage et le système de types concernés sont complexes.

Dans le cas de HM(X), néanmoins, l'approche syntaxique ne présente pas de difficulté. La propriété d'auto-réduction pour HM(X), laissée par Sulzmann [69] à l'état de conjecture, a été vérifiée par Christian Skalka et moi-même [4].

Il faut noter que la propriété d'auto-réduction est intéressante en soi, et non seulement comme un outil pour établir la propriété de sûreté. En effet, elle donne au programmeur ou au compilateur la possibilité d'effectuer statiquement certaines réductions (technique connue sous le nom d'*inlining*) sans affecter le type des expressions concernées. On peut donc considérer les approches qui permettent d'établir auto-réduction et sûreté comme supérieures à celles qui ne permettent d'obtenir que la seconde propriété.

2.4.3 Sûreté semi-syntaxique

La troisième méthode, que j'appelle approche *semi-syntaxique*, a été développée pour tenter d'allier les avantages des deux précédentes. Je l'ai d'abord appliquée, à titre d'exercice, à HM(X) [2], puis à un système plus complexe, JOIN(X) [3].

Le join-calcul n'ayant à ma connaissance aucune sémantique dénotationnelle, la première approche n'était pas applicable. Par ailleurs, comme je l'ai expliqué plus haut, l'approche syntaxique se révèle particulièrement lourde dans le cas de JOIN(X).

À partir d'une suggestion d'Alexandre Frey, qu'il a lui-même récemment développée [59], j'ai donc proposé une approche hybride. L'idée est de procéder en deux temps. Dans un premier temps, on emploie l'approche syntaxique pour démontrer la correction d'un système dit *algébrique*, selon la terminologie de Frey. Les *monotypes* t n'y sont pas nécessairement des termes mais des éléments d'un espace mathématique abstrait. Le polymorphisme y est *extensionnel*, c'est-à-dire que les *polytypes* s – l'équivalent des schémas de types – sont définis simplement comme des ensembles non vides (et, en présence de sous-typage, clos vers le haut) de monotypes. Les règles de généralisation et d'instanciation sont

alors élémentaires. Par exemple, dans le cas où le langage considéré est le λ-calcul, elles s'écrivent comme suit :

$$\frac{\forall t \in s \quad \Gamma \vdash e : t}{\Gamma \vdash e : s} \qquad\qquad \frac{\Gamma \vdash e : s \qquad t \in s}{\Gamma \vdash e : t}$$

Il faut noter que la règle de généralisation peut avoir un nombre infini de prémisses, si le polytype s est lui-même infini. L'intérêt de ce système est son extrême simplicité : ses règles de typage ne sont guère plus complexes que celles du λ-calcul simplement typé, et il est très facile de démontrer, par la méthode syntaxique, qu'il est sûr.

Alors, dans un second temps, on revient au système dont on souhaite ultimement démontrer la sûreté, ici HM(X), et on démontre que chacun de ses jugements peut être interprété comme un ensemble non vide de jugements du système algébrique. Cette propriété peut s'énoncer comme suit :

Théorème 2.12 (Sûreté semi-syntaxique) *Soit $C, \Gamma \vdash e : \sigma$ un jugement. Soit ϕ une solution de C. Alors $\phi\Gamma \vdash e : \phi\sigma$ est un jugement valide dans le système algébrique.* ◇

Ce résultat implique, en particulier, que tout programme bien typé dans HM(X) est également bien typé dans le système algébrique. La sûreté de HM(X) découle alors de la sûreté de ce dernier. La preuve de ce théorème se fait par induction structurelle sur la dérivation de $C, \Gamma \vdash e : \sigma$. Comme dans le cas de la première approche, donc, la preuve comporte un cas pour chacune des règles de typage de HM(X). Ainsi, elle est équilibrée et modulaire.

Cette approche a été appliquée pour prouver la sûreté de JOIN(X) [3]. Elle a également permis d'alléger sensiblement la preuve de sûreté de l'analyse de flots d'information que j'ai développée en collaboration avec Vincent Simonet (chapitre 5). Enfin, elle a été récemment détaillée par Frey [59]. Celui-ci la pousse plus avant en démontrant que, sous certaines hypothèses, le système algébrique et le système syntaxique *coïncident*. Cela implique que le premier peut être considéré comme une spécification, et le second comme sa réalisation. En particulier, la propriété d'auto-réduction du système algébrique s'applique alors également au système syntaxique.

2.4.4 Sûreté syntaxique à base de contraintes

La dernière méthode que je souhaite décrire est une variante de la méthode syntaxique. J'ai mentionné plus haut que la démonstration classique de la propriété d'auto-réduction exige plusieurs lemmes de normalisation, dont l'objet est de contrôler l'emploi des règles de typage non dirigées par la syntaxe et de faciliter ainsi l'analyse de la dérivation de typage associée à l'expression initiale e. Or, ces mêmes lemmes et cette même analyse apparaissent également lorsqu'on établit la complétude des règles de génération de contraintes, c'est-à-dire lorsqu'on montre que si e est bien typée, alors $\exists\alpha.[\![e : \alpha]\!]$ est satisfiable (théorème 2.8). Il y a donc une certaine duplication de l'effort.

Une solution naturelle est alors d'établir la propriété d'auto-réduction non pas en termes des règles de typage, mais en termes des règles de génération de contraintes, puisque ces dernières, par construction, sont équivalentes aux premières, mais présentent l'avantage d'être dirigées par la syntaxe. L'énoncé de la propriété d'auto-réduction devient alors :

Théorème 2.13 (Auto-réduction en termes de contraintes) *Si e se réduit en e', alors on a $[\![e : \tau]\!] \Vdash [\![e' : \tau]\!]$.* ◇

Autrement dit, l'assertion « si e admet le type τ, alors e' admet le type τ, » exprimée de façon logique comme une assertion d'implication de contraintes, est vraie.

L'intérêt de cette approche réside dans le fait que la preuve de cet énoncé est d'une nature très calculatoire. En effet, puisque la forme des expressions e et e' est connue, les contraintes $[\![e : \tau]\!]$ et $[\![e' : \tau]\!]$ sont le résultat d'un simple calcul. De plus, pour prouver que la première entraîne la seconde, il suffit habituellement d'appliquer des lois d'équivalence ou d'implication de contraintes prises parmi un ensemble relativement restreint. On pourrait donc, en principe, tenter l'automatisation de ces preuves. C'est là un axe de recherche difficile, mais intéressant. Jusqu'ici, cette approche n'a été appliquée que de façon manuelle [1].

2.4.5 Discussion

Pour conclure cet aperçu des diverses preuves de sûreté de HM(X), on pourrait espérer qu'étant donné l'effort consacré à prouver la sûreté du typage dans ce cadre général, la question soit aujourd'hui close, et qu'on n'ait plus à l'avenir à établir la sûreté de telle ou telle analyse à base de contraintes. Or, c'est en partie vrai, mais en partie seulement.

Les systèmes de types présentés au chapitre 3, par exemple, sont des *instances* de HM(X). Pour en démontrer la sûreté, il suffira de vérifier que les schémas de types attribués aux opérations primitives sont sûrs, ce qui revient à effectuer quelques fragments de la preuve d'auto-réduction, et que le type d'une valeur permet d'en prédire la forme, ce qui revient à effectuer quelques fragments de la preuve de progrès. Le gros des preuves est donc réutilisé.

Cependant, de nombreuses analyses à base de contraintes ne constituent pas directement des instances de HM(X), mais plutôt des *variantes*. Cela est dû au fait que le système HM(X) ne vise qu'à garantir l'absence d'erreurs à l'exécution, tandis qu'une analyse de programmes typique s'intéresse à des aspects plus précis du comportement d'un programme : flots de données, flots d'information, nombre d'utilisations de chaque valeur ou expression, etc. Si les techniques de preuve restent inchangées, il peut alors être difficile de s'appuyer sur la propriété de sûreté de HM(X). Le chapitre 4 suggère une technique générale, basée sur un codage des programmes, qui permet cela dans certains cas, typiquement lorsque la propriété que l'on cherche à garantir est une *propriété de sûreté*.

A contrario, le chapitre 5 illustre une analyse dont la définition a de nombreux points communs avec celle de HM(X), mais pour laquelle aucune preuve n'a pu être directement réutilisée.

Enfin, malgré sa généralité, le système HM(X) ne constitue pas le dernier mot en matière de typage à base de contraintes. Au chapitre 6, j'en présenterai une *extension*, qui nécessite naturellement une nouvelle preuve de sûreté. Dans ces deux cas, les techniques de preuve décrites ci-dessus restent bien sûr pertinentes.

Chapitre 3

Sous-typage non structurel, contraintes conditionnelles et rangées

Ce chapitre commente une partie du matériau publié dans [42, 73–76].

J'AI JUSQU'ICI décrit le typage à base de contraintes de façon abstraite, sans choisir un langage de contraintes ni un domaine d'application particuliers. Dans ce chapitre, j'illustre ces idées de façon plus concrète, en m'intéressant à un langage de contraintes particulier, lequel combine trois ingrédients, à savoir (i) une notion de *sous-typage non structurel*, (ii) un nouveau prédicat de base sur les types, connu sous le nom de *contrainte de sous-typage conditionnelle*, et (iii) la notion de *rangée*. Ce langage expressif permet d'attribuer des schémas de types précis à de nombreuses opérations sur les *enregistrements*, sur leurs cousins les *objets*, ainsi que sur les *sommes* ou *variantes*. J'illustre cela en prenant pour exemple l'opération de *concaténation* des enregistrements.

3.1 Un langage de contraintes

Je décris d'abord le langage de contraintes utilisé dans ce chapitre. Ma description restera volontairement informelle, car une définition détaillée serait trop lourde ; le lecteur désireux d'en savoir plus pourra consulter [75] ou [76].

3.1.1 Sous-typage non structurel

On peut distinguer deux approches sensiblement différentes de la notion de sous-typage. Dans l'approche *basée sur les noms*, les types sont déclarés par l'utilisateur, qui leur attribue un nom, et la relation de sous-typage est elle aussi déclarée. Dans cette approche, un type est le plus souvent un atome – un nom. C'est l'approche adoptée par de nombreux langages de programmation orientés objets, comme par exemple Java [77]. Dans l'approche *basée sur la structure*, à l'inverse, les types comme la relation de sous-typage pré-existent. Dans cette approche, un type est le plus souvent un arbre – un terme. La relation de sous-typage est alors donnée par une série de règles fixées, qui portent sur la structure des deux types concernés. Cette approche est adoptée par de nombreux articles théoriques.

Le sous-typage dit « non structurel » appartient à la *seconde* approche, et n'est donc pas nommé de façon très judicieuse. Je conserve néanmoins cette dénomination, qui me semble standard. Le sous-typage non structurel est ainsi nommé par opposition au sous-typage dit « structurel » ou « atomique. » Dans les deux cas, les types sont des termes, c'est-à-dire des arbres, finis ou infinis selon les travaux. Une relation d'ordre entre termes est dite « structurelle » si deux termes comparables ont nécessairement la même structure en tant qu'arbres, c'est-à-dire s'ils définissent les mêmes chemins. Elle est dite « non structurelle » dans le cas contraire.

On obtient une relation de sous-typage structurelle lorsqu'on prend pour point de départ un langage de types dénué de sous-typage et lorsqu'on décore ces types par des *atomes* sur lesquels existe une

relation d'ordre, d'où l'appellation de sous-typage « atomique. » Ce procédé est utilisé dans la définition de nombreuses analyses de programmes. Les chapitres 4 et 5 en donneront quelques exemples.

On obtient une relation de sous-typage non structurelle lorsqu'on adopte des axiomes qui rendent comparables deux types de formes distinctes. L'exemple le plus simple consiste à ajouter au λ-calcul simplement typé deux types $\bot$ et $\top$. Le premier n'est attribuable à aucune valeur, et ne peut donc être attribué à une expression que si elle ne termine pas. Le second est attribuable à toutes les valeurs. La grammaire des types est alors donnée par

$$\tau ::= \bot \mid \tau \to \tau \mid \top$$

et la relation de sous-typage par

$$\bot \leq \tau \qquad\qquad \tau \leq \top \qquad\qquad \frac{\tau_1' \leq \tau_1 \qquad \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}$$

Ces définitions peuvent être interprétées de façon inductive ou co-inductive, donnant naissance à des espaces d'arbres finis ou infinis. Le point important est la présence des axiomes $\bot \leq \tau$ et $\tau \leq \top$, où le type τ est arbitraire, ce qui signifie que deux types de formes différentes peuvent être comparables. On notera que la relation de sous-typage organise les types en un *treillis*. Ce point est important, car il facilite la résolution et la simplification des contraintes de sous-typage [42].

L'ajout de $\bot$ et $\top$ est probablement l'emploi le plus simple, mais pas le plus intéressant pragmatiquement, du sous-typage non structurel. En réalité, l'introduction du sous-typage non structurel a été initialement motivée par le typage des langages de programmation dotés d'enregistrements. Cardelli [78] a proposé d'employer la notion de sous-typage pour formaliser l'idée, déjà bien connue dans la communauté orientée objets, qu'un enregistrement doté de champs en surnombre peut être fourni dans un contexte où un enregistrement doté de champs moins nombreux est attendu. En termes plus formels, cela signifie que si un type enregistrement est formé simplement d'une fonction partielle des étiquettes de champs vers les types :

$$\tau ::= \ldots \mid \{\ell : \tau_\ell\}_{\ell \in L}$$

alors on peut adopter la règle de sous-typage suivante :

$$\frac{L \supseteq L' \qquad \forall \ell \in L' \quad \tau_\ell \leq \tau_\ell'}{\{\ell : \tau_\ell\}_{\ell \in L} \leq \{\ell : \tau_\ell'\}_{\ell \in L'}}$$

Cette règle donne naissance à une relation de sous-typage non structurelle, puisqu'elle rend comparables des types enregistrements n'ayant pas les mêmes champs. Elle peut sembler beaucoup plus complexe que celle qui découle de la simple présence de $\top$ et $\bot$ dans le système, mais en réalité, tel n'est pas le cas. Pour s'en persuader, on peut imaginer un codage des types enregistrements, dans lequel un champ absent serait considéré comme présent mais de type $\top$. Les types enregistrements deviendraient alors des fonctions *totales* des étiquettes de champs vers les types, c'est-à-dire des produits infinis. Ce codage est correct au sens où l'axiome $\tau \leq \top$ et l'axiome de covariance du produit donnent bien lieu à la relation de sous-typage attendue entre types enregistrements. Son intérêt n'est d'ailleurs pas purement théorique : à condition de se doter d'un moyen d'exprimer les produits infinis, il conduit effectivement à une simplification. Je reviendrai sur ce point lors de l'introduction des rangées (§3.1.3).

Le sous-typage non structurel a donc été introduit pour permettre le sous-typage entre types enregistrements, qui lui-même était sensé avoir un lien avec la relation d'héritage entre classes dans les langages orientés objets. Malheureusement, la théorie du sous-typage non structurel est complexe : la résolution de contraintes semble exiger un temps cubique, même dans l'hypothèse simplificatrice où la relation de sous-typage forme un treillis ; les formes résolues sont difficiles à déchiffrer pour le programmeur ; enfin, on ne sait toujours pas si l'implication de contraintes est décidable [41, 42, 79–84]. De plus, l'emploi de sous-typage n'est pas indispensable pour les applications illustrées dans ce chapitre et dans [75] : il ne permet qu'un léger gain d'expressivité vis-à-vis d'une approche basée uniquement sur les contraintes (d'égalité) conditionnelles et les rangées. Malgré ces points négatifs, il s'agit d'une théorie simple et élégante, qui mérite d'être connue.

3.1.2 Contraintes conditionnelles

Les contraintes conditionnelles ont été inventées par Reynolds [85] puis utilisées, sous diverses formes, dans des travaux plus récents [57, 86, 87].

D'un point de vue logique, on peut considérer les contraintes conditionnelles comme une forme restreinte du connecteur d'implication $\Rightarrow$, lequel est puissant, mais coûteux. En effet, si le langage de contraintes contient l'implication, alors il contient la négation $\neg$, donc en fait toute la logique du premier ordre. Si les types sont interprétés de façon standard par des termes, finis ou infinis, le problème de la satisfiabilité est alors de complexité non élémentaire [88]. On peut également considérer les contraintes conditionnelles comme une forme restreinte du connecteur de disjonction $\vee$, dont l'introduction confère au problème de satisfiabilité un coût exponentiel. Ces idées sont développées de façon informelle ci-dessous.

Dans un langage doté d'une sémantique à appel par valeur, si une expression e_2 diverge, alors toute application de la forme $e_1\, e_2$ diverge également. Par conséquent, si e_2 admet le type $\bot$, alors il est sûr d'attribuer à $e_1\, e_2$ ce même type $\bot$. En d'autres termes, si on peut prouver que la fonction dénotée par e_1 ne sera jamais appelée, alors on peut ignorer le type de son résultat. Cette idée peut être matérialisée par une règle de typage, qui vient s'ajouter à la règle de typage habituelle des λ-abstractions (figure 2.7) :

$$\frac{C, \Gamma; x : \bot \vdash e : \tau}{C, \Gamma \vdash \lambda x.e : \bot \to \bot}$$

Cette règle indique que toutes les fonctions *bien typées* admettent le type $\bot \to \bot$. Il est facile de démontrer que l'ajout de cette règle ne compromet pas la sûreté du typage. Dans le cadre de la preuve du théorème d'auto-réduction, on doit vérifier que, si $(\lambda x.e)\, v$ admet un type, alors $[v/x]e$ admet le même type. Or, puisqu'aucune valeur n'admet le type $\bot$, la dérivation de typage associée à $\lambda x.e$ ne peut pas se terminer par une instance de la règle ci-dessus : elle se termine donc nécessairement par une instance de la règle de typage habituelle des λ-abstractions. Le reste de la preuve est inchangé.

Comment effectuer l'inférence de types pour un système de types doté d'une telle règle ? Le typage de chaque λ-abstraction nécessite à présent un choix entre la règle habituelle et la nouvelle règle. L'algorithme de génération de contraintes (figure 2.6) pourrait être modifié comme suit :

$$\begin{array}{rcl} [\![\lambda x.e : \tau]\!] & = & \exists \alpha_1 \alpha_2 \alpha_3.(\textbf{\textit{def}}\ x : \alpha_1\ \textbf{\textit{in}}\ [\![e : \alpha_2]\!] \wedge \alpha_1 \to \alpha_3 \leq \tau \wedge \\ & & \qquad (\alpha_1 = \bot \vee \alpha_2 \leq \alpha_3)) \end{array}$$

La principale nouveauté réside dans le fait que le type de e_2, représenté par α_2, et le codomaine du type flèche que l'on construit, représenté par α_3, ne sont plus nécessairement égaux. Au lieu de cela, une disjonction apparaît : soit α_1 vaut $\bot$, auquel cas α_3 peut prendre la valeur $\bot$ ou bien, par subsomption, n'importe quelle valeur ; soit, comme auparavant, α_3 doit être égal à (ou, ce qui revient au même, supertype de) α_2.

Cette spécification présente l'inconvénient d'employer la disjonction, qui dans le cas général est coûteuse. L'idée est alors de se contenter d'une forme moins puissante de disjonction. Ici, par exemple, on peut se limiter aux disjonctions dont le membre gauche est de la forme $\alpha_1 = \bot$. Cette forme restreinte est, en effet, strictement moins expressive. La disjonction $\alpha_1 = \bot \vee \alpha_2 \leq \alpha_3$ est souvent notée $\alpha_1 \neq \bot \Rightarrow \alpha_2 \leq \alpha_3$, d'où l'appellation de « contrainte conditionnelle. » On se gardera bien d'introduire la forme symétrique $\alpha_1 = \bot \Rightarrow \alpha_2 \leq \alpha_3$, car on pourrait alors coder la disjonction $C_1 \vee C_2$ sous la forme $\exists \alpha_1.(\alpha_1 = \bot \Rightarrow C_1 \wedge \alpha_1 \neq \bot \Rightarrow C_2)$.

On peut aller plus loin et affirmer que toutes les fonctions admettent le type $\bot \to \bot$, quel que soit leur contenu. On ajoute alors au système de types la règle de typage suivante :

$$C, \Gamma \vdash \lambda x.e : \bot \to \bot$$

Certes, cette règle semble trop flexible pour être mise en œuvre dans un langage de programmation. En effet, elle permet d'écrire des fragments de code dénués de tout sens, à la condition qu'on ne les utilise pas. Elle retarde ainsi la détection des erreurs, plus encore que la règle précédente. Néanmoins, elle est sûre, pour la même raison que la précédente, et peut avoir un intérêt dans le cadre d'une analyse de programmes.

L'inférence de types, pour un système doté d'une telle règle, serait effectuée de la façon suivante :

$$[\![\lambda x.e : \tau]\!] \quad = \quad \exists \alpha_1 \alpha_2 . (\alpha_1 \to \alpha_2 \leq \tau \wedge (\alpha_1 = \bot \vee \textbf{\textit{def}}\ x : \alpha_1\ \textbf{\textit{in}}\ [\![e : \alpha_2]\!]))$$

Ici, il n'est nécessaire d'examiner le corps de la fonction e que si l'équation $\alpha_1 = \bot$ ne peut être satisfaite, ce qui a lieu lorsqu'on impose que τ soit un type flèche de domaine autre que $\bot$. Encore une fois, on emploie ici une contrainte conditionnelle, c'est-à-dire une disjonction dont le membre gauche est de la forme $\alpha_1 = \bot$.

Le lecteur se demande peut-être pourquoi je m'intéresse à ces règles de typage non standard. D'une part, vouloir donner à toutes les fonctions le type $\bot \to \bot$ peut sembler une idée saugrenue. D'autre part, l'ajout de nouvelles règles de typage nous place hors du cadre de HM(X). En réalité, ces règles ne m'intéressent pas pour elles-mêmes, mais en tant qu'illustration du pouvoir d'expression des contraintes conditionnelles. En bref, ces contraintes permettent de *ne pas prendre en compte le résultat d'une expression, si on peut prouver que celle-ci ne sera pas exécutée*. Cette capacité se révèle particulièrement intéressante lorsqu'on souhaite attribuer un type précis à une expression contenant un *branchement* (construction ***if***, fonction définie par cas, etc.), ou bien à une opération primitive dont la sémantique fait intervenir un branchement. Les contraintes conditionnelles permettent alors d'ignorer la contribution des branches mortes et de ne prendre en compte le type que des branches jugées susceptibles d'être exécutées. Cette idée, due à Aiken *et al.* [87], est employée dans mon traitement de la concaténation des enregistrements (§3.2), qui se situe bel et bien *dans* le cadre de HM(X).

Les contraintes conditionnelles auxquelles je m'intéresse ici n'ajoutent rien à la complexité du problème de satisfiabilité : celle-ci reste cubique. Elles permettent donc effectivement d'éviter le coût lié à la disjonction. Malheureusement, les contraintes conditionnelles ne sont pas pour autant « gratuites : » en leur présence, le problème de la comparaison entre schémas de types déclarés et schémas de types inférés, que l'on peut comprendre en termes logiques comme la vérification d'une formule de préfixe $\forall\exists$, devient difficile [89]. D'un point de vue pratique, cela les rend plus intéressantes pour l'analyse entièrement automatique de programmes, où le programmeur n'a pas la possibilité d'annoter une expression par un schéma de types contraint, que pour la conception d'un langage de programmation typé où les contraintes seraient accessibles au programmeur.

3.1.3 Rangées

Les rangées, imaginées par Wand [90] et Rémy [91], constituent une syntaxe finie pour dénoter des familles infinies de types, c'est-à-dire des fonctions totales ou presque totales d'un ensemble dénombrable d'étiquettes vers les types. Le lecteur en trouvera un exposé récent et relativement complet dans [1].

J'emploie le mot *rangée* informellement pour désigner aussi bien l'objet syntaxique que la fonction presque totale qu'il dénote. J'appelle *presque totale* une fonction dont le domaine est un ensemble cofini, c'est-à-dire de complémentaire fini. Les fonctions presque totales sont utilisées comme éléments dans la construction de fonctions totales : en effet, une fonction totale pourra s'exprimer comme l'extension d'une fonction presque totale en un nombre fini de points.

Les rangées se présentent habituellement sous la forme d'une extension de la grammaire des types :

$$\tau ::= \ldots \mid (\ell : \tau ; \tau) \mid \partial\tau$$

La construction $(\ell : \tau_1 ; \tau_2)$ dénote la rangée qui à l'étiquette ℓ associe le type τ_1 et qui en tout autre point coïncide avec la rangée τ_2. La construction $\partial\tau$ dénote une rangée qui à toute étiquette élément de son domaine associe le type τ. (Cette construction, introduite par Rémy [92], n'est pas toujours indispensable : par exemple, elle ne sera pas employée en §3.2.) Enfin, tout constructeur de types ordinaire pourra dorénavant jouer simultanément le rôle d'un constructeur de rangées. Par exemple, on pourra appliquer le constructeur de types $\to$ à deux rangées τ_1 et τ_2, pour obtenir une rangée qui à toute étiquette élément ℓ de son domaine associe le type $\tau_1(\ell) \to \tau_2(\ell)$.

L'interprétation logique des rangées est construite de façon à valider ces affirmations informelles.

De ce fait, elle valide les axiomes suivants, qui constituent la théorie équationnelle des rangées :

$$\begin{aligned}(\ell_1 : \tau_1; \ell_2 : \tau_2; \tau) &= (\ell_2 : \tau_2; \ell_1 : \tau_1; \tau)\\ (\ell : \tau; \partial\tau) &= \partial\tau\\ (\ell : \tau_1; \tau_2) \to (\ell : \tau_1'; \tau_2') &= (\ell : \tau_1 \to \tau_1'; \tau_2 \to \tau_2')\\ \partial\tau_1 \to \partial\tau_2 &= \partial(\tau_1 \to \tau_2)\end{aligned}$$

Les deux derniers axiomes concernent le cas du constructeur de types $\to$, mais sont également valides pour tout autre constructeur de types « ordinaire. »

Les contraintes de sous-typage, conditionnelles ou non, peuvent être étendues aux rangées, et sont alors interprétées point par point. Par exemple, si τ_1, τ_2 et τ_3 sont des rangées de même domaine, alors la contrainte $\tau_1 \neq \bot \Rightarrow \tau_2 \leq \tau_3$ sera interprétée comme la conjonction infinie des contraintes $\tau_1(\ell) \neq \bot \Rightarrow \tau_2(\ell) \leq \tau_3(\ell)$, pour ℓ parcourant le domaine de τ_1, τ_2 et τ_3.

Ainsi, toutes les commodités dont on dispose au niveau des types ordinaires (constructeurs, contraintes) sont également disponibles au niveau des rangées, et interprétées point par point. De ce fait, si le langage de types permet de décrire une transformation de *champs*, alors il permet également de décrire l'application de cette transformation, point par point, à des *enregistrements*. Une illustration de cette idée est fournie en §3.2.

Le discours informel ci-dessus effectue une distinction entre « types » et « rangées, » bien que tous soient dénotés par la méta-variable τ. D'un point de vue formel, on impose aux types une discipline de *sortes*, qui permet de distinguer types ordinaires et rangées, et qui de plus permet de garder trace du domaine de chaque rangée. On se restreint dès lors aux types bien sortés, aux substitutions et valuations préservant les sortes, etc. L'emploi d'une discipline de sortes exige en principe une extension du cadre théorique dans lequel on se situe, c'est-à-dire ici du système HM(X). Celle-ci ne pose aucune difficulté, et je la passe donc sous silence.

J'ai décrit les rangées comme dénotant des fonctions des étiquettes vers les types, et considéré implicitement les étiquettes comme des atomes n'ayant d'autre propriété que leur identité. Il est intéressant de relaxer cette hypothèse et de s'intéresser à des étiquettes plus structurées. Si les étiquettes sont des k-uplets d'atomes, par exemple, on obtient ce que Rémy appelle *rangées à k dimensions.* J'ai également étudié, dans un travail resté à l'état d'ébauche, le cas où les étiquettes sont atomiques, mais sont organisées en une hiérarchie arborescente. Enfin, dans [76], j'ai suggéré une vision plus abstraite des rangées, où les étiquettes ne sont pas nécessairement atomiques, et où les ensembles d'étiquettes manipulés ne sont pas nécessairement finis ou cofinis, mais peuvent appartenir à toute classe d'ensembles préservée par les opérations booléennes et où la vacuité est décidable. Les applications de ces idées à l'inférence de types sont jusqu'ici restées à l'état de suggestions. Néanmoins, ces généralisations potentielles montrent que les rangées nous réservent peut-être encore du nouveau.

3.1.4 Résolution de contraintes

Bien que ma description du langage de contraintes et de son interprétation soit restée informelle, j'espère que le lecteur aura pu s'en constituer une idée. En bref, la grammaire des types permet de construire des types ordinaires et des rangées, tandis que le langage des contraintes permet d'imposer à ces types et rangées des contraintes, conditionnelles ou non, de sous-typage. L'interprétation de la relation de sous-typage est, en général, non structurelle. Comment détermine-t-on, dans un tel langage, si une contrainte est ou non satisfiable ?

Commençons par considérer uniquement les contraintes de sous-typage entre types ordinaires, et supposons, pour simplifier, que les seuls constructeurs de types sont $\bot$, $\top$ et $\to$ (§3.1.1). Dans ce cas, l'algorithme standard [54, 81] consiste à effectuer une *clôture* de la contrainte par transitivité et par

décomposition structurelle :

$$\begin{array}{rcl} \tau_1 \leq \alpha \wedge \alpha \leq \tau_2 & \longrightarrow & \tau_1 \leq \tau_2 \\ \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2' & \longrightarrow & \tau_1' \leq \tau_1 \wedge \tau_2 \leq \tau_2' \\ \top \leq \bot & \longrightarrow & \textbf{\textit{false}} \\ \top \leq \tau_1 \to \tau_2 & \longrightarrow & \textbf{\textit{false}} \\ \tau_1 \to \tau_2 \leq \bot & \longrightarrow & \textbf{\textit{false}} \end{array}$$

On démontre sans difficulté que la réécriture préserve l'interprétation de la contrainte et que toute forme normale ne contenant pas ***false*** est satisfiable. Par ailleurs, une analyse de complexité dans le style de McAllester [38] montre que la forme normale est obtenue en temps $O(n^3)$ au pire.

L'ajout des contraintes conditionnelles ne complique guère la situation. L'idée est d'employer un algorithme paresseux, en considérant toute contrainte conditionnelle comme satisfaite tant que sa condition est réfutable. Cela peut être traduit par les règles de clôture suivantes :

$$\begin{array}{rcl} \tau_1 \to \tau_2 \leq \alpha \wedge \alpha \neq \bot \Rightarrow \tau_2 \leq \tau_3 & \longrightarrow & \tau_2 \leq \tau_3 \\ \top \leq \alpha \wedge \alpha \neq \bot \Rightarrow \tau_2 \leq \tau_3 & \longrightarrow & \tau_2 \leq \tau_3 \end{array}$$

Il n'est pas difficile de vérifier que les propriétés énoncées ci-dessus sont préservées par cette extension.

L'ajout des rangées pose plus de difficultés, au moins d'un point de vue théorique. Si l'unification des rangées a été décrite de façon précise [1, 91], la complexité de l'algorithme existant n'est pas connue. J'ai pu construire des contraintes pathologiques de taille n dont la résolution par cet algorithme exige un temps $O(n^2)$ et même $O(n^3)$! Cela semble dû en partie au fait que l'algorithme engendre de nouvelles variables au cours du calcul, et en partie au fait qu'il ne mémorise pas toutes les équations qu'il découvre, ce qui peut le mener à redécouvrir plusieurs fois un même fait.

Cet algorithme d'unification des rangées peut être combiné avec l'algorithme de résolution de contraintes de sous-typage par clôture. C'est ce j'ai réalisé dans [73]. Des algorithmes similaires ont été mis au point par Fähndrich [57]. Néanmoins, je ne crois pas que la complexité théorique de ces deux réalisations soit connue avec précision. L'accent y est mis sur l'efficacité *pragmatique*, ce qui exige l'emploi d'algorithmes complexes de simplification de contraintes, que je ne décrirai pas ici.

Ces lacunes théoriques pourraient ne pas sembler bien graves, mais certains auteurs malicieux – au sens français du mot ! – en ont récemment tiré parti pour prétendre que la résolution de contraintes combinant sous-typage non structurel et rangées n'était pas *à leur connaissance* de complexité polynomiale [93]. C'est pourquoi j'ai été amené à étudier formellement la complexité du problème [76]. Cela n'aura pas suffi à empêcher mes amis de récidiver [94]... mais, cette étude s'étant révélée fructueuse, je ne leur en tiendrai pas rigueur.

L'approche adoptée dans [76] consiste à contourner la difficulté en abandonnant la formulation classique des rangées, telle que présentée en §3.1.3, en faveur d'une formulation purement à base de contraintes. Je n'ai pas démontré formellement l'équivalence entre les deux ; j'en fais ici la conjecture. Dans la nouvelle présentation, la construction $(\ell : \tau; \tau)$ disparaît de la grammaire des *types* : une rangée doit donc être de la forme α ou $\partial\tau$. Pour compenser cela, j'enrichis la grammaire des *contraintes*, lorsqu'elles portent sur des rangées, en les annotant par un *filtre* L, c'est-à-dire, dans le cas le plus simple, un ensemble fini ou cofini d'étiquettes. Par exemple, si τ_1 et τ_2 sont deux variables de rangées, la contrainte $L : \tau_1 \leq \tau_2$ sera interprétée comme la conjonction des $\tau_1(\ell) \leq \tau_2(\ell)$, *pour* ℓ *parcourant* L.

L'avantage de cette formulation est de se prêter aisément à une généralisation de l'algorithme de clôture esquissé plus haut. La règle de clôture par transitivité, par exemple, devient, dans les cas des contraintes entre rangées,

$$L_1 : \tau_1 \leq \alpha \wedge L_2 : \alpha \leq \tau_2 \quad \longrightarrow \quad L_1 \cap L_2 : \tau_1 \leq \tau_2$$

La règle de propagation doit également être adaptée au cas de types dont certains fils sont des rangées, et, réciproquement, au cas des rangées de la forme $\partial\tau$, dont les fils sont des types ; on consultera [76] pour plus de détails.

L'analyse de complexité se fait alors sans difficulté, et on constate que le coût de la résolution est $O(n^3 m \log m)$, où n mesure la taille de la contrainte et m le nombre d'étiquettes distinctes qui y figurent. On aurait peut-être pu espérer mieux, mais du moins la complexité n'est-elle pas exponentielle, comme le laissaient entendre les mauvaises langues...

Le type de certaines opérations élémentaires sur les enregistrements s'exprime de façon légèrement différente dans l'approche classique des rangées [91] et dans l'approche à base de contraintes [76]. Le type de l'opération d'*extension* à l'étiquette ℓ, par exemple, est habituellement

$$\forall \alpha \psi \varphi . \{\ell : \psi; \varphi\} \to \alpha \to \{\ell : \textbf{\textit{pre}}\, \alpha; \varphi\}$$

et devient, dans l'approche à base de contraintes,

$$\forall \alpha \varphi_1 \varphi_2 [\{\ell\} : \partial(\textbf{\textit{pre}}\, \alpha) \leq \varphi_2 \wedge (\mathcal{L} \setminus \{\ell\}) : \varphi_1 \leq \varphi_2] . \{\varphi_1\} \to \alpha \to \{\varphi_2\}$$

Ici, la première contrainte emploie le filtre singleton $\{\ell\}$ pour indiquer que le champ ℓ sera présent, avec le type α, dans l'enregistrement créé. La seconde contrainte emploie le filtre cosingleton $\mathcal{L} \setminus \{\ell\}$ pour indiquer que tous les champs autres que ℓ conservent le même statut dans l'enregistrement créé que dans l'enregistrement initial. Ce schéma de types polymorphe contraint est certes moins lisible que le précédent ; aussi, il n'est pas dit que mon idée ait un impact pratique immédiat. Son intérêt réside d'une part en une analyse de complexité facilitée, comme je l'ai expliqué ci-dessus, et d'autre part en une série de généralisations, obtenues en changeant la structure des filtres L, comme je l'ai suggéré en §3.1.3.

Les opérations primitives sur les enregistrements qui traitent toutes les étiquettes de façon uniforme se verront attribuer le même type dans les deux approches. C'est le cas de l'opération de concaténation, que je décris à présent.

3.2 Application au typage de la concaténation des enregistrements

3.2.1 Sémantique

On pourrait voir un enregistrement comme une fonction partielle, de domaine fini L, des étiquettes vers les valeurs, qu'on noterait $\{\ell : v_\ell\}_{\ell \in L}$. Je préfère adopter une notation légèrement différente, mais équivalente : un enregistrement sera considéré comme une fonction totale des étiquettes vers les *champs*, qu'on notera $\{\ell : w_\ell\}_{\ell \in \mathcal{L}}$, où les champs sont donnés par $w ::= \textbf{\textit{Abs}} \mid \textbf{\textit{Pre}}\, v$. La *concaténation* de deux champs w_1 et w_2, notée $w_1 + w_2$, est alors définie par les deux règles suivantes :

$$\begin{aligned} w + \textbf{\textit{Pre}}\, v &= \textbf{\textit{Pre}}\, v \\ w + \textbf{\textit{Abs}} &= w \end{aligned}$$

Je définis ici une version *asymétrique* de la concaténation, où le second champ a précédence s'il est présent, et où le premier champ est consulté dans le cas où le second est absent. Notons que la définition de la concaténation s'apparente à une définition par cas : il y a branchement suivant la forme des arguments. La concaténation des enregistrements est alors définie point par point :

$$\{\ell : w_\ell\}_{\ell \in \mathcal{L}} + \{\ell : w'_\ell\}_{\ell \in \mathcal{L}} \quad \longrightarrow \quad \{\ell : w_\ell + w'_\ell\}_{\ell \in \mathcal{L}}$$

Le problème est d'attribuer un schéma de types satisfaisant à cette opération. La difficulté vient en partie de ce que la concaténation de *champs* est définie par cas, ce qui rend difficile la prédiction statique de la forme de son résultat, et en partie de ce que la concaténation d'*enregistrements* est définie point par point en termes de la concaténation de champs.

3.2.2 Typage

Commençons par fixer la manière dont on attribue des types aux champs. Les champs w forment une classe syntaxique distincte des valeurs v, donc on leur attribuera des types d'une *sorte* distincte

de celle des types de valeurs. Notre système de sortes distinguera donc d'une part types ordinaires et rangées, comme suggéré en §3.1.3, d'autre part types de champs et types de valeurs.

Par définition, un champ est de la forme soit ***Abs***, soit ***Pre*** v. Il semble donc naturel d'introduire deux constructeurs de types de champs, nommés ***abs*** et ***pre*** $\cdot$, d'arités respectives 0 et 1. Le champ ***Abs*** aura le type ***abs***, et le champ ***Pre*** v aura le type ***pre*** τ si v a le type τ, où τ est un type de valeurs. Je suis ici l'approche de Rémy [91].

En présence de sous-typage non structurel, il est naturel d'introduire également deux types de champs $\bot$ et $\top$. Aucun champ w ne sera de type $\bot$; tout champ w sera de type $\top$. On pose $\bot \leq \tau \leq \top$ pour tout type de champ τ. L'introduction de ces deux éléments n'est pas anodine. D'abord, elle permet aux types de champs de former un treillis (§3.1.1). Ensuite, la présence de $\bot$ est essentielle pour garantir que les contraintes conditionnelles employées ci-dessous ne permettent pas le codage de la disjonction, donc que la résolution de contraintes conditionnelles reste de complexité polynomiale [75].

Enfin, il est possible, quoique facultatif, d'introduire un constructeur de types de champs ***maybe*** $\cdot$, accompagné des axiomes ***abs*** $\leq$ ***maybe*** τ et ***pre*** $\tau \leq$ ***maybe*** τ. Les types de champs forment alors toujours un treillis, où ***maybe*** τ est la borne supérieure de ***abs*** et ***pre*** τ. En d'autres termes, un champ w admettra le type ***maybe*** τ s'il est soit ***Abs***, soit ***Pre*** v, où v admet le type τ. La présence de ce constructeur de types permet d'attribuer un type légèrement plus précis à l'opération de concaténation, sans augmenter la complexité du système. Je l'adopte donc ici.

Les constructeurs de types ***pre*** $\cdot$ et ***maybe*** $\cdot$ sont covariants, ce qui est naturel, car je suppose que les champs d'un enregistrement ne sont accessibles qu'en lecture, non en écriture.

La définition du treillis des types de champs est à présent terminée. Avant de continuer, on peut formuler deux remarques. D'abord, je n'ai pas posé ***pre*** $\tau \leq$ ***abs***, comme cela est courant. En effet, si j'avais fait cela, ***abs*** serait devenu essentiellement synonyme de $\top$, puisque tout champ w aurait admis le type ***abs***. Or, le typage de la concaténation exige que l'on puisse exprimer l'absence *certaine*, et non *possible*, d'un champ. Ce problème est expliqué en détail par Cardelli et Mitchell [95]. Ensuite, on pourra remarquer que l'on dispose d'une certaine latitude dans la définition du treillis des types de champs. Rémy [96] illustre ce fait, dans le cadre d'un langage à objets, en proposant une hiérarchie particulièrement riche.

On peut maintenant passer à l'étape suivante : exprimer un schéma de types satisfaisant pour l'opération de concaténation des champs. Le tout est d'imaginer une contrainte qui relie de façon satisfaisante les types des champs w_1 et w_2 au type de $w_1 + w_2$, pour tous champs w_1 et w_2. Cette contrainte doit refléter de façon sûre le comportement de l'opération $+$, tel qu'il a été défini plus haut (§3.2.1). Puisque ce comportement est défini par la conjonction de deux règles, dont le membre gauche fait intervenir des motifs non triviaux, il est naturel de formuler cette contrainte comme la conjonction de deux sous-contraintes, lesquelles feront intervenir des contraintes conditionnelles :

$$\begin{array}{rcll} w + \textbf{\textit{Pre}}\, v & = & \textbf{\textit{Pre}}\, v & \exists \alpha_2.((\varphi_2 \leq \textbf{\textit{maybe}}\, \alpha_2) \wedge (\textbf{\textit{pre}} \leq \varphi_2 \Rightarrow \textbf{\textit{pre}}\, \alpha_2 \leq \varphi_3)) \\ w + \textbf{\textit{Abs}} & = & w & (\textbf{\textit{abs}} \leq \varphi_2 \Rightarrow \varphi_1 \leq \varphi_3) \end{array}$$

Je suppose ici que les variables de types φ_1, φ_2 et φ_3 sont associées à w_1, w_2 et $w_1 + w_2$, respectivement.

La contrainte $\varphi_2 \leq$ ***maybe*** α_2 fait en sorte que α_2 représente le type de la valeur v. Une façon de s'en persuader est de constater que, si, par hypothèse, ***Pre*** v est de type φ_2, et si la contrainte $\varphi_2 \leq$ ***maybe*** α_2 est satisfaite, alors par sous-typage ***Pre*** v admet le type ***maybe*** α_2, ce qui par construction implique que v admet le type α_2.

La contrainte conditionnelle ***pre*** $\leq \varphi_2 \Rightarrow$ ***pre*** $\alpha_2 \leq \varphi_3$ est d'une forme un peu particulière : ***pre*** n'est pas un type, mais un constructeur de types. On peut considérer qu'il s'agit là d'un sucre syntaxique pour $(\exists \alpha.$***pre*** $\alpha \leq \varphi_2) \Rightarrow$ ***pre*** $\alpha_2 \leq \varphi_3$. (Les contraintes conditionnelles employées ici sont donc d'une forme légèrement plus générale que celle présentée en §3.1.2.) Cette contrainte indique que, si φ_2 dépasse ***pre***, donc si le champ w_2 risque d'être présent, alors le résultat de l'opération risque d'être ***Pre*** v, donc le type φ_3 du résultat doit être au-dessus du type ***pre*** α_2 de ***Pre*** v.

La contrainte $\exists \alpha_2.((\varphi_2 \leq$ ***maybe*** $\alpha_2) \wedge ($***pre*** $\leq \varphi_2 \Rightarrow$ ***pre*** $\alpha_2 \leq \varphi_3))$ pourrait être formulée, de façon équivalente, $\forall \alpha_2.($***pre*** $\alpha_2 \leq \varphi_2 \Rightarrow$ ***pre*** $\alpha_2 \leq \varphi_3)$. Cette formulation semble plus lisible et reflète plus visiblement la règle $w +$ ***Pre*** $v =$ ***Pre*** v. Néanmoins, je n'ai pas étudié la résolution de cette forme de contrainte conditionnelle. Je présume que cela ne poserait pas de difficulté, mais ceci reste à vérifier.

La contrainte conditionnelle $\textbf{\textit{abs}} \leq \varphi_2 \Rightarrow \varphi_1 \leq \varphi_3$ indique que, si φ_2 dépasse ***abs***, donc si w_2 risque d'être absent, alors le résultat de l'opération risque d'être w_1, donc le type φ_3 du résultat doit être au-dessus du type φ_1 de w_1. Cette contrainte reflète directement la règle $w + \textbf{\textit{Abs}} = w$.

Il est important de comprendre que les contraintes ci-dessus ne constituent qu'une approximation du comportement dynamique de l'opération de concaténation. En particulier, lorsqu'on évalue une expression de la forme $w_1 + w_2$, une et une seule des deux règles qui définissent la concaténation des champs est utilisée. Lorsque l'on résoud statiquement les contraintes ci-dessus, au contraire, il se peut que *les deux* contraintes conditionnelles entrent en jeu. En effet, si φ_2 est de la forme $\textbf{\textit{maybe}}\,\tau_2$, alors les conditions $\textbf{\textit{pre}} \leq \varphi_2$ et $\textbf{\textit{abs}} \leq \varphi_2$ sont toutes deux satisfaites. Le solveur devra alors résoudre la contrainte $\exists\alpha_2.(\textbf{\textit{maybe}}\,\tau_2 \leq \textbf{\textit{maybe}}\,\alpha_2 \wedge \textbf{\textit{pre}}\,\alpha_2 \leq \varphi_3 \wedge \varphi_1 \leq \varphi_3)$, qui est équivalente à $(\textbf{\textit{pre}}\,\tau_2) \sqcup \varphi_1 \leq \varphi_3$. En d'autres termes, si la présence du second champ n'est pas certaine, alors le type du résultat est obtenu en combinant les types des premier et second champs. On pourra remarquer que, si φ_1 est lui-même de la forme $\textbf{\textit{pre}}\,\tau_1$, alors la contrainte devient $\textbf{\textit{pre}}\,(\tau_1 \sqcup \tau_2) \leq \varphi_3$: autrement dit, pourvu que le premier champ soit présent avec certitude, le champ résultat sera également présent avec certitude, même si le second champ risque d'être absent. (Le lecteur pourra vérifier que l'affirmation symétrique est vraie également.) La contrainte ci-dessus décrit donc le comportement de la concaténation de façon relativement précise.

Le schéma de types contraint que j'associe à l'opération de concaténation des champs est donc :

$$\forall\varphi_1\varphi_2\varphi_3[\exists\alpha_2.((\varphi_2 \leq \textbf{\textit{maybe}}\,\alpha_2) \wedge (\textbf{\textit{pre}} \leq \varphi_2 \Rightarrow \textbf{\textit{pre}}\,\alpha_2 \leq \varphi_3)) \\ \wedge (\textbf{\textit{abs}} \leq \varphi_2 \Rightarrow \varphi_1 \leq \varphi_3)].\varphi_1 \to \varphi_2 \to \varphi_3$$

Ici, φ_1, φ_2 et φ_3 sont des variables de types de champs, tandis que α_2 est une variable de types de valeurs. Toutes quatre sont des variables de types ordinaires, par opposition à des variables de rangées.

Passons à présent à la dernière étape : attribuer un schéma de types à l'opération de concaténation des *enregistrements*, laquelle a été définie point par point en termes de la concaténation des champs. Le procédé est si simple que je n'ai presque rien à dire : il suffit de conserver la même contrainte, mais de changer φ_1, φ_2, φ_3 et α_2 en *variables de rangées*. Parce que les contraintes portant sur les rangées sont interprétées point par point, la contrainte ainsi modifiée prendra exactement la signification souhaitée. Le schéma de types contraint que j'associe à l'opération de concaténation des enregistrements est donc :

$$\forall\varphi_1\varphi_2\varphi_3[\exists\alpha_2.((\varphi_2 \leq \textbf{\textit{maybe}}\,\alpha_2) \wedge (\textbf{\textit{pre}} \leq \varphi_2 \Rightarrow \textbf{\textit{pre}}\,\alpha_2 \leq \varphi_3)) \\ \wedge (\textbf{\textit{abs}} \leq \varphi_2 \Rightarrow \varphi_1 \leq \varphi_3)].\{\varphi_1\} \to \{\varphi_2\} \to \{\varphi_3\}$$

Toutes les contraintes ci-dessus sont entre rangées. Le constructeur de types $\{\cdot\}$ construit un type enregistrement à partir d'une rangée.

Cette solution peut paraître surprenante : où est donc passée la difficulté ? En réalité, elle réside maintenant dans le solveur de contraintes, qui doit décider la satisfiabilité de contraintes entre rangées. J'ai dit quelques mots de ce problème en §3.1.4. Le lecteur désireux d'en savoir plus pourra consulter [1], pour l'unification des rangées, [75], pour la résolution de contraintes conditionnelles de sous-typage, et [76] pour une combinaison des rangées et des contraintes conditionnelles de sous-typage.

En conclusion, cette approche à base de contraintes est sophistiquée et élégante. Elle est, à ma connaissance, la seule à permettre l'inférence de types en temps polynomial pour toutes les opérations sur les enregistrements, y compris (diverses formes de) la concaténation. Ses principaux aspects négatifs sont, d'une part, la complexité des schémas de types contraints, qui rendent les types inférés difficiles à déchiffrer par le programmeur ; d'autre part, la complexité du problème de la comparaison entre schémas de types déclarés et schémas de types inférés [89], qui rend problématique son emploi dans un langage de programmation modulaire.

Pour ces deux raisons et d'autres encore, l'opération de concaténation et l'inférence de types ne coexistent dans aucun langage de programmation implémenté. Pourtant, la combinaison est tentante : elle pourrait servir, par exemple, à la définition d'un langage doté d'inférence de types et de *mixins* [97–99] de première classe.

3.2.3 Travaux reliés

Dans certains travaux précédents [95], la concaténation d'enregistrements n'a pu recevoir aucun type satisfaisant. Dans d'autres [100–102], elle admet plusieurs types, mais pas de type principal. De

tels systèmes sont sûrs, mais n'ont pas d'algorithme d'inférence de types.

Wand [90] attribue à la concaténation des enregistrements un schéma de types contraint principal, mais emploie pour cela une disjonction logique, ce qui donne à la résolution de contraintes un coût exponentiel, et n'est donc guère satisfaisant.

Ohori et Buneman [103, 104] s'intéressent à la « jointure naturelle » des bases de données, que l'on peut comprendre comme une variante récursive de la concaténation des enregistrements. Ils lui attribuent un schéma de types contraint, à l'aide de contraintes *ad hoc*. Malheureusement, ces contraintes permettent le codage de la disjonction – en fait, on peut réduire 3-SAT au problème de la satisfiabilité de contraintes, ce qui montre que ce dernier est NP-complet. Cette approche est donc essentiellement de même nature que celle de Wand. Toutefois, Ohori et Buneman suggèrent qu'il est possible, « en pratique, » de retarder la résolution des contraintes portant sur des variables non encore instanciées. Cette remarque porte en germe, de façon très informelle, l'idée d'une approche paresseuse de la résolution de contraintes, laquelle est permise, de façon formelle, par les contraintes conditionnelles que j'ai présentées.

De façon fort similaire, Palsberg et Zhao [94] attribuent un schéma de types principal à la concaténation des enregistrements à l'aide de contraintes *ad hoc*. Ces contraintes permettent également le codage de la disjonction, et, par une réduction similaire, on peut montrer que le problème de la satisfiabilité est NP-complet. Il n'y a donc ici aucun progrès.

Il faut souligner que la propriété qui permet une résolution « paresseuse, » et empêche le codage de la disjonction, est la présence du type de champs $\bot$ dans le modèle. En son absence, un type de champs ne peut être que « présent ou absent. » Dans ce cas, la contrainte C définie comme $\exists\alpha.(\textbf{\textit{abs}} \leq \alpha \Rightarrow C_1 \wedge \textbf{\textit{pre}} \leq \alpha \Rightarrow C_2)$ est équivalente à la disjonction $C_1 \vee C_2$, car α doit nécessairement satisfaire une des deux conditions, et toute résolution paresseuse est impossible. En sa présence, au contraire, cette équivalence ne tient pas, car il suffit d'attribuer à α la valeur $\bot$ pour satisfaire les deux contraintes conditionnelles, ce qui signifie que C est équivalente à ***true***. De façon générale, lorsque le solveur rencontre une contrainte conditionnelle portant sur une variable α, il peut supposer que α vaut $\bot$, et ignorer la contrainte conditionnelle, tant que rien ne vient contredire cette hypothèse. C'est ce que Ohori et Buneman appellent « retarder la résolution des contraintes portant sur des variables non encore instanciées. » Ainsi, le simple ajout de l'élément $\bot$ au modèle logique affecte dramatiquement la complexité de la résolution de contraintes.

Sulzmann [69] propose également une solution basée sur des contraintes *ad hoc*. Malheureusement, le modèle logique dans lequel elles sont interprétées n'est pas défini de façon claire. L'algorithme de résolution de contraintes semble être de nature paresseuse. Sa correction n'est pas prouvée.

Rémy [105] propose un schéma de types principal pour la concaténation, à l'aide de rangées et de contraintes *ad hoc*. Les contraintes ne reçoivent aucune interprétation logique. Elles sont seulement munies d'une notion syntaxique de clôture, sur laquelle la preuve de sûreté du typage s'appuie directement ; cette technique (à mon avis peu viable) est inspirée par Eifrig *et al.* [54]. Les règles de clôture reflètent directement, au niveau des types, la sémantique de l'opération de concaténation des champs, et sont de nature paresseuse. La solution que je propose ici est inspirée de [105], mais emploie des contraintes conditionnelles plus standard.

Rémy [106] propose un codage de la concaténation en termes de λ-abstraction et d'extension d'enregistrements, pour lesquels l'inférence de types est possible. L'idée est intéressante, et illustre la technique de « typage à travers un codage » à laquelle le chapitre 4 est consacré. Malheureusement, le type principal d'un enregistrement, dans le système de types ainsi dérivé, est polymorphe vis-à-vis d'une variable de rangée. Comme l'argument d'une λ-abstraction ne peut pas, en ML, être polymorphe, une fonction qui attend un enregistrement et l'utilise dans plusieurs contextes différents peut être mal typée dans ce système de types. Cela diminue sensiblement l'intérêt pratique de cette approche.

Enfin, certains chercheurs ont utilisé, pour l'inférence de types dans des langages de programmation dotés d'enregistrements ou d'objets, la théorie logique des *feature trees* [107, 108], des arbres dont les arêtes sont étiquetées et non numérotées, et qui présentent donc des points communs avec les rangées. Müller et Nishimura [109, 110] appliquent cette approche à l'inférence de types pour un langage dotés d'objets et d'une notion de message de première classe. Une approche à base de rangées est également possible [75].

Chapitre 4

Typage à travers un codage

Ce chapitre commente une partie du matériau publié dans [111–113].

J'AI DÉCRIT en §2.4 comment établir la sûreté du typage pour un système à base de contraintes tel que HM(X). Mais comment établir la correction d'une analyse de programmes à base de contraintes, si celle-ci n'est pas une *instance*, mais une *variante* de HM(X) ? Doit-on écrire une nouvelle preuve de sûreté ?

Je rappelle, dans ce chapitre, une technique permettant de réutiliser la preuve de sûreté de HM(X), et d'obtenir presque *gratuitement* une preuve de correction pour une analyse de programmes à base de contraintes. L'idée, qui n'est pas nouvelle [114], est de formuler l'analyse comme la composition d'une transformation de programmes, ou *codage*, et d'un système de types standard, comme HM(X). La correction de l'analyse découle alors de la correction du codage, d'une part, et de la sûreté de HM(X), d'autre part.

En fait, cette technique permet de modulariser non seulement la *preuve* de l'analyse, mais sa *définition* même. On peut en effet prendre le codage pour point de départ, le composer avec le système de types HM(X), et en déduire, de façon quasi automatique, une analyse à base de contraintes. J'appelle ce procédé « typage à travers un codage. »

Une instance de cette technique appartient au folklore : il s'agit du codage des exceptions en termes de sommes [115–117]. Si l'on compose ce codage avec un système de types standard, doté de sommes binaires anonymes, on obtient un système de types *dérivé* dans lequel toute fonction admet un type de la forme $\tau_1 \to \tau_2 + \tau_3$. Ici, τ_2 est le type des valeurs que la fonction est susceptible de renvoyer, et τ_3 est le type des exceptions qu'elle est susceptible de lever. On peut écrire $\tau_1 \xrightarrow{\tau_3} \tau_2$ au lieu de $\tau_1 \to \tau_2 + \tau_3$, pour retrouver la notation classique des systèmes d'analyse des exceptions. On dira alors, pour employer la terminologie classique, que τ_3 est un *effet*. Si de plus on restreint syntaxiquement le langage source pour que les exceptions soient des variantes, c'est-à-dire des valeurs construites par application d'un constructeur de données, et si le système de types du langage cible est doté d'un constructeur de types variantes paramétré par une rangée, alors, dans l'analyse obtenue par composition, les effets sont des rangées indexées par les étiquettes d'exceptions. On retrouve ainsi certaines analyses classiques [118, 119].

Une autre illustration de cette technique est fournie par Rémy [106], qui, grâce à un codage simple, montre que, si l'on dispose d'un langage de programmation typé doté d'une opération d'extension des enregistrements, alors on peut mécaniquement en dériver un système de types permettant non seulement l'extension, mais aussi la concaténation des enregistrements. Rémy emploie principalement une extension du système de types de Damas et Milner avec des rangées comme système de types cible, mais évoque rapidement la combinaison du même codage avec d'autres systèmes de types.

Ce chapitre décrit brièvement le principe de la technique de « typage à travers un codage » (§4.1), puis en donne deux applications nouvelles. Toutes deux visent à définir, à moindre coût, des analyses de programmes à base de contraintes, et à en démontrer la correction. La première (§4.2) est une analyse de flots d'information pour un langage de programmation purement fonctionnel, et est basée sur un codage dans lequel les valeurs portent des étiquettes qui en indiquent la provenance. La seconde (§4.3) vise

à garantir l'absence d'erreurs à l'exécution pour un langage de programmation doté d'un mécanisme « d'inspection de pile, » et est basée sur un codage « security-passing style. »

4.1 Généralités

Commençons par exposer l'idée dans le cas général. Elle est d'une grande simplicité. D'abord, on suppose donnés deux langages de programmation, initialement non typés, que l'on qualifie respectivement de « source » et « cible. » Leurs expressions sont respectivement notées e et f. On les suppose tous deux munis d'une sémantique opérationnelle à petits pas, c'est-à-dire d'une relation de réduction déterministe. Ces deux relations sont notées $\longrightarrow$. Ensuite, on suppose donné un codage du langage source vers le langage cible, noté $[\![\cdot]\!]$. Ce codage doit être correct :

Hypothèse 4.1 (Correction du codage) *Si e se réduit en e', alors $[\![e]\!]$ se réduit en $[\![e']\!]$. Si e est une valeur, alors $[\![e']\!]$ se réduit en une valeur. Si e est bloquée, alors $[\![e]\!]$ se réduit en un terme bloqué.* ◇

La terminologie « se réduit » représente ici la relation $\longrightarrow^\star$, c'est-à-dire la réduction en un nombre arbitraire d'étapes. L'hypothèse de correction est formulée de façon assez rigide, par souci de simplicité. On peut, si besoin, la relaxer légèrement en introduisant une relation de réduction « administrative. » J'omets ces détails secondaires. L'hypothèse de correction implique que e et $[\![e]\!]$ ont la même sémantique : en particulier, e provoque une erreur à l'exécution si et seulement si $[\![e]\!]$ fait de même. Il est important de noter que l'hypothèse concerne des programmes non typés. Sa preuve est donc purement syntaxique, et indépendante de tout système de types.

Supposons maintenant donné un système de types pour le langage cible. On supposera que ce système se présente sous la forme d'un prédicat binaire $\cdot : \cdot$ dont les arguments sont une expression f et un typage T. Toute expression admettant un typage est dite bien typée. Ce système de types doit être sûr, au sens de Wright et Felleisen [72] :

Hypothèse 4.2 (Sûreté du typage cible) *Si $f : T$ et $f \longrightarrow f'$, alors $f' : T$. De plus, aucune expression f close bloquée n'est bien typée.* ◇

La construction du système de types « dérivé » est alors immédiate, par simple composition. On pose qu'une expression e admet le typage T si et seulement si $[\![e]\!]$ admet le typage T. Ceci définit un système de types pour le langage source, dont les typages sont les mêmes que ceux du système de types du langage cible. À l'aide des deux hypothèses ci-dessus, il est immédiat de vérifier que le système de types dérivé est sûr :

Théorème 4.3 (Sûreté du typage dérivé) *Si $e : T$ et $e \longrightarrow e'$, alors $e' : T$. De plus, aucune expression e close bloquée n'est bien typée.* ◇

Tout ceci peut sembler bien élémentaire. C'est justement parce que cette approche est simple qu'il ne faut pas se priver de l'employer lorsque cela est possible !

4.2 Un codage à base d'étiquettes

L'analyse de flots d'information est un problème de sécurité ancien [120] pour lesquels des techniques d'analyse statique, présentées de façon informelle, sont connues depuis longtemps [121, 122]. L'objectif de l'analyse est d'établir une propriété de *non-interférence*, c'est-à-dire d'absence de dépendance entre certaines des entrées d'un programme, ou fragment de programme, et certaines de ses sorties.

Ce problème a récemment connu un regain d'intérêt, et on a voulu formaliser ces techniques ainsi que leur preuve de correction. Certains chercheurs ont proposé de formuler ces analyses sous forme de systèmes de types. Cependant, la plupart des systèmes de types qui ont alors été avancés [123–130] étaient très rudimentaires : par exemple, tous étaient dénués de polymorphisme, ce qui en pratique les rendait inutilisables. De plus, leur preuve de correction dupliquait souvent certains aspects de la preuve de sûreté d'un système de types standard.

Pour remédier à ces deux problèmes, j'ai proposé d'approcher l'analyse de flots d'information, dans le cas d'un langage de programmation purement fonctionnel, à travers un codage [111]. Le but était d'obtenir d'un seul coup toute une famille d'analyses, par composition avec différents systèmes de types pour le langage cible, et de ne devoir prouver une fois pour toutes que la correction du codage.

En quoi doit consister un codage permettant d'effectuer une analyse de flots d'information, c'est-à-dire une analyse de dépendance ? La réponse est simple : l'effet du codage doit être d'instrumenter le programme de façon à ce qu'une analyse de dépendance *dynamique* soit incorporée au programme modifié. Ensuite, par composition avec le système de types du langage cible, on obtient une analyse de dépendance *statique*.

Ici, la tâche est facilitée par le fait qu'une analyse de dépendance dynamique pour un langage purement fonctionnel a déjà été définie par Abadi, Lampson et Lévy [131]. Leur idée est d'ajouter à chaque valeur une étiquette qui résume, de façon approximative, l'ensemble des informations dont elle dépend. Leur analyse est présentée sous forme d'un calcul où toute expression peut porter une étiquette :

$$e ::= \ldots \mid \ell : e$$

et où la sémantique opérationnelle contient, outre les règles classiques, une règle permettant la propagation des étiquettes :

$$(\ell : e_1)\, e_2 \longrightarrow \ell : (e_1\, e_2) \qquad \textit{(lift)}$$

On peut comprendre cette règle, d'un point de vue mécanique, comme une façon de déplacer l'étiquette ℓ pour faire apparaître un β-redex. D'un point de vue plus abstrait, la règle a pour effet de déplacer l'étiquette de la fonction vers son résultat, donc permet de mémoriser le fait que le résultat de l'application *dépend* de la fonction. Parce que le résultat de la fonction ne dépend pas nécessairement de son argument, qui peut être ignoré, la règle symétrique n'est pas utilisée.

On peut prouver que la façon dont les étiquettes sont propagées constitue effectivement une analyse de flots d'information. Cette propriété, dénommée « stabilité » par Abadi *et al.* [131], est énoncée de façon constructive dans [111], et prouvée de façon élémentaire :

Théorème 4.4 (Stabilité) $e_1 \longrightarrow^\star e_2$ *et* $\lfloor e_2 \rfloor_L = e_2$ *impliquent* $\lfloor e_1 \rfloor_L \longrightarrow^\star e_2$. ◇

Ici, L représente un ensemble quelconque d'étiquettes. L'expression $\lfloor e \rfloor_L$ est obtenue en effaçant tous les sous-termes de e portant une étiquette hors de L, c'est-à-dire en les remplaçant par un « trou » qui bloque l'évaluation lorsqu'il parvient en tête. Ainsi, l'énoncé indique que si l'évaluation de e_1 produit un résultat e_2, alors les sous-termes de e_1 portant des étiquettes qui n'apparaissent plus dans e_2 n'ont en fait pas servi au calcul.

Le calcul d'Abadi *et al.* [131] forme mon langage source. Mon but est à présent de définir un système de types pour ce langage. Pour réutiliser un système de types existant, je dois éliminer la construction $\ell : e$ et la règle de réduction *(lift)*, qui sont non standard. Le rôle du codage est précisément de les simuler à l'aide de constructions standard. Ici, toute valeur du langage source sera codée, dans le langage cible, par une paire d'une valeur et d'une étiquette. Je supposerai donc que, outre un noyau purement fonctionnel doté de paires, le langage cible contient une constante ℓ pour chaque étiquette $\ell \in \mathcal{L}$. Le langage source permet l'imbrication des étiquettes : on peut écrire $\ell_1 : \ell_2 : e$. Or, pour des raisons de simplicité et de régularité, on préfère que, dans un programme codé, chaque valeur porte exactement une étiquette. Le codage sera donc approximatif. Sa définition exigera la présence dans le langage cible d'une opération binaire $\sqcup$, qui à deux étiquettes associe leur borne supérieure. À partir d'ici, donc, je suppose que $\mathcal{L}$ forme un treillis. J'omets la définition exacte du codage et sa preuve de correction [111].

Le langage cible est donc un λ-calcul standard, doté d'une opération primitive inhabituelle, à savoir $\sqcup$. Ainsi, le codage permet de passer d'un langage source doté d'une sémantique inhabituelle à un langage cible doté seulement d'une opération primitive supplémentaire. C'est un gain, car on n'aura pas de difficulté à exhiber des systèmes de types sûrs pour le langage cible : il suffit de choisir un système de types existant pour le λ-calcul avec paires et de démontrer que les types attribués aux constantes ℓ et à l'opération $\sqcup$ sont cohérents.

Qu'attend-on précisément du système de types cible ? Comme indiqué précédemment (§4.1), il doit être sûr. Par ailleurs, une hypothèse supplémentaire est nécessaire :

Hypothèse 4.5 *$e_1 \sqcup (e_2 \sqcup e_3)$ et $(e_1 \sqcup e_2) \sqcup e_3$ ont les mêmes types. $(\bot \sqcup e)$ et e ont les mêmes types. Enfin, toute étiquette ℓ est un type, et si la valeur ℓ_1 admet le type ℓ_2, alors on a $\ell_1 \leq \ell_2$.* ◇

La façon la plus évidente de satisfaire cette hypothèse est d'adopter un système de types cible doté d'une relation de sous-typage, laquelle étend le treillis des étiquettes, de poser que la constante ℓ a le type ℓ, et d'attribuer à l'opération $\sqcup$ le schéma de types contraint $\forall\alpha[\alpha \leq \top].\alpha \to \alpha \to \alpha$.

Parce que toute valeur est codée par une paire d'une valeur et d'une étiquette, son type, à travers le codage, sera une paire d'un type « habituel » et d'une étiquette, par exemple $int \times \ell$, que nous écrirons int^ℓ, pour retrouver la notation habituelle des analyses de flots d'information. À l'aide des diverses hypothèses que nous avons formulées, à l'aide de la correction du codage, et à l'aide de la propriété de stabilité du langage source (théorème 4.4), on peut démontrer la propriété suivante :

Théorème 4.6 (Non-interférence) *Supposons $e_1, e_2 : int^\ell$ et $\lfloor e_1 \rfloor_{\downarrow\ell} = \lfloor e_2 \rfloor_{\downarrow\ell}$. Alors, soient e_1 et e_2 divergent, soit e_1 et e_2 convergent et produisent la même valeur.* ◇

Cet énoncé indique que, si e_1 et e_2 ont le type « entier public » et s'ils ne diffèrent que par des sous-termes étiquetés comme « secrets, » alors ils ont la même sémantique. En d'autres termes, le système de types garantit qu'aucun « secret » ne peut contribuer au calcul d'un résultat de type « public. »

Ce résultat est valide vis-à-vis d'une sémantique en appel par nom. On peut en déduire que, vis-à-vis d'une sémantique en appel par valeur, si e_1 et e_2 convergent, alors ils produisent la même valeur. Cette variante est connue sous le nom de « non-interférence faible. »

Ici, le typage à travers un codage permet de reconstruire des systèmes de types très similaires à ceux qui existaient auparavant [127–130]. Les types sont annotés par des étiquettes, et les règles de typage sont essentiellement les mêmes. L'intérêt de l'approche est de montrer que ces systèmes sont en fait obtenus par simple composition d'une analyse de dépendance dynamique et d'un système de types préexistants. De plus, nous obtenons, sans effort, une preuve de correction, et la capacité d'employer, si on le souhaite, polymorphisme, types récursifs, ou tout autre caractéristique connue.

Il est intéressant de noter que la composition d'un codage similaire avec une analyse de programmes existante a été proposée indépendamment par Ross et Sagiv [132].

4.3 Un codage « security-passing style »

Le langage Java [77] est doté d'un mécanisme particulier de contrôle d'accès [133], destiné à empêcher l'accès par un fragment de code considéré comme peu fiable (téléchargé depuis Internet, par exemple) à une ressource sensible (le contenu du système de fichiers, par exemple).

Le point de départ, classique [134], est la donnée d'un ensemble de *principaux* $\mathcal{P}$, chaque fragment de code étant *signé* par un principal, d'un ensemble de *ressources* $\mathcal{R}$, et d'une *matrice de droits d'accès* $\mathcal{A}$ reliant principaux et ressources. Pour simplifier la présentation, je supposerai qu'un principal *est* un ensemble de ressources, ce qui permet de faire abstraction de $\mathcal{A}$.

Le langage de programmation contient une instruction, `checkPermission`, permettant de signaler qu'on accède à une ressource r. Cette opération n'a aucun effet si l'accès est considéré comme autorisé, mais échoue dans le cas contraire. Cette instruction est utilisée par le code qui contrôle la ressource r : évidemment, un fragment de code adverse n'a aucun intérêt à l'employer ! Par conséquent, pour déterminer si l'accès doit être ou non autorisé, le système doit considérer non pas seulement le signataire du fragment de code en cours d'exécution, mais aussi l'historique de l'exécution : la requête est-elle provoquée, directement ou indirectement, par un fragment de code adverse ? Pour répondre à cette question, l'approche adoptée par Java est d'étudier l'historique des appels de méthode qui ont conduit à l'instruction `checkPermission` considérée. Cette technique est connue sous le nom d'inspection de pile. En première approximation, il faut vérifier que tous les principaux associés aux fragments de code figurant sur la pile sont dignes de confiance. Cependant, cette condition est trop restrictive. Un fragment de code non fiable peut légitimement provoquer un accès indirect au système de fichiers, si par exemple il demande l'affichage d'une chaîne de caractères dans une police non encore disponible. C'est alors le fragment de code (fiable) chargé de l'affichage qui prend la responsabilité de cet accès. Java fournit pour cela une second instruction, `doPrivileged`. Son effet est de stopper l'inspection de pile : si

un principal digne de confiance prend la responsabilité de l'accès à une ressource, on ne vérifie pas le bien-fondé de cette décision, donc on n'examine pas par quoi elle a été provoquée.

La description originale du système de contrôle d'accès de Java [135, 136] est très opérationnelle : elle décrit de façon informelle l'ensemble des instructions offertes au programmeur et leur comportement. Cet état de fait a été critiqué, et certains chercheurs ont tenté d'en étudier plus profondément les propriétés. Wallach et Felten [137, 138] en donnent une description semi-formelle en termes de la logique ABLP et, plus intéressant, en proposent une implémentation, dite « security-passing style, » où aucune inspection de pile n'est nécessaire. Au lieu de cela, l'ensemble des droits d'accès disponibles est maintenu à jour en permanence, ce qui demande un calcul à chaque appel de méthode, mais permet d'implémenter `checkPermission` en temps constant. Cette approche est asymptotiquement plus efficace et, surtout, va servir de base à mon analyse statique. Fournet et Gordon [139] proposent une étude théorique poussée des propriétés garanties par l'inspection de pile.

Malgré ces travaux, il reste difficile de prédire le comportement d'un programme truffé d'instructions `doPrivileged` et `checkPermission`. Le problème réside dans le fait que toute méthode peut maintenant échouer, c'est-à-dire lever `SecurityException`, si elle effectue, directement ou indirectement, un appel à `checkPermission`. Or l'ensemble des droits (ou permissions, ou ressources) dont une méthode doit disposer pour pouvoir être exécutée normalement n'est nulle part spécifié explicitement. (Il peut bien sûr figurer dans la documentation de la méthode, mais, cette spécification n'étant pas vérifiable automatiquement, elle n'a que peu de valeur.) On se trouve ainsi en situation de programmer « au petit bonheur, » sans trop savoir dans quelles conditions le code que l'on écrit risque d'échouer ni quand on doit prendre la responsabilité de l'accès à une ressource à l'aide de `doPrivileged`.

Ces limitations ont été soulignées par un travail de Skalka et Smith [140], lequel a éveillé mon intérêt pour cette question. Cet article propose un système de types pour un λ-calcul doté de constructions similaires à `checkPermission` et `doPrivileged`. Le type de chaque fonction indique explicitement l'ensemble des ressources auxquelles elle est susceptible d'accéder, ce qui permet de garantir statiquement que `checkPermission` ne peut échouer. En fait, les constructions `checkPermission` et `doPrivileged` n'influencent plus la sémantique des programmes bien typés, et peuvent être considérées comme de simples annotations de types. Le principal intérêt de l'approche est de permettre une spécification *vérifiable* des droits d'accès requis par chaque fragment de code. Accessoirement, elle permet également un gain d'efficacité, puisque le contrôle d'accès est en principe effectué à la compilation. Ses limitations résident dans plusieurs hypothèses simplificatrices (les principaux et les ressources sont atomiques, connus à la compilation, et ne sont reliés par aucune relation d'ordre ; les ressources ne sont pas des valeurs de première classe) qui rendent problématique une véritable application à Java.

Le travail que j'ai réalisé avec Christian Skalka et Scott Smith ne résoud pas ces limitations, mais propose une reformulation des résultats initiaux de Skalka et Smith [140] à l'aide de la technique du typage à travers un codage, et, grâce à cela, en augmente la puissance de façon significative. L'idée est simple : pour obtenir un système *statique* de contrôle d'accès, il suffit de composer une technique *dynamique* de contrôle d'accès, comme l'implémentation « security-passing style » de Wallach [138], avec un système de types standard, donc *statique*.

Donnons un aperçu rapide du langage source et de son codage. Le langage source est une extension du noyau de ML par trois constructions : $r.e$, qui correspond à `doPrivileged`, prend la responsabilité pour tout accès à r durant l'exécution de e ; $r\,!\,e$, qui correspond à `checkPermission`, signale un accès à r, et échoue si celui-ci n'est pas autorisé ; $p.e$ représente une expression e signée par le principal p. On exige que le corps de toute fonction soit signé. La syntaxe du langage source est donc la suivante :

$$e ::= x \mid \lambda x.p.e \mid e\,e \mid \textbf{\textit{let}}\ x = e\ \textbf{\textit{in}}\ e \mid r.e \mid r\,!\,e \mid p.e$$

La sémantique opérationnelle du langage source reflète de façon fidèle la définition de l'inspection de pile. Les constructions $p.e$ et $r.e$ n'ont aucun effet en soi, mais sont mémorisées pendant l'évaluation de l'expression e. Elles sont supprimées lorsque cette évaluation est terminée. Cela est reflété par la syntaxe des contextes d'évaluation :

$$E ::= \ldots \mid p.E \mid r.E$$

$$
\begin{array}{rcl}
[\![x]\!]_\star & = & x \\
[\![\lambda x.p.e]\!]_\star & = & \lambda x.\lambda s.[\![p.e]\!]_\star \\
[\![e_1\ e_2]\!]_p & = & [\![e_1]\!]_p\ [\![e_2]\!]_p\ s \\
[\![\textbf{\textit{let}}\ x = e_1\ \textbf{\textit{in}}\ e_2]\!]_p & = & \textbf{\textit{let}}\ x = [\![e_1]\!]_p\ \textbf{\textit{in}}\ [\![e_2]\!]_p \\
[\![r.e]\!]_p & = & \textbf{\textit{let}}\ s = s \cup (\{r\} \cap p)\ \textbf{\textit{in}}\ [\![e]\!]_p \\
[\![r\,!\,e]\!]_p & = & r \in s; [\![e]\!]_p \\
[\![p.e]\!]_\star & = & \textbf{\textit{let}}\ s = s \cap p\ \textbf{\textit{in}}\ [\![e]\!]_p
\end{array}
$$

FIG. 4.1 – Le codage « security-passing style »

ainsi que par les deux règles de réduction suivantes :

$$
\begin{array}{rcl}
r.v & \longrightarrow & v \\
p.v & \longrightarrow & v
\end{array}
$$

Les informations contenues dans la pile pendant l'exécution d'un programme Java sont donc ici contenues dans le contexte d'évaluation. (N'oublions pas qu'un contexte d'évaluation a une structure de liste, donc de pile !) Cela nous permet d'exprimer l'inspection de pile comme une inspection du contexte d'évaluation :

$$E[r\,!\,e] \longrightarrow E[e] \qquad \text{si } E \vdash r$$

La condition $E \vdash r$, que l'on peut lire « l'état de la pile E autorise l'accès à la ressource r, » est définie de façon relativement simple : il faut que la séquence de marqueurs p et r qui apparaissent sur la pile E appartiennent à un certain langage régulier, dont je ne donne pas ici la définition.

Le codage « security-passing style, » qui compile le langage source et sa sémantique non standard vers un langage cible standard, est donné par la figure 4.1. Le codage est indexé par un principal p, qui représente le signataire du fragment de code à traduire. Ce principal est remplacé par $\star$ lorsqu'il est sans importance. J'emploie la variable distinguée s pour représenter l'ensemble des ressources accessibles à chaque instant. Je suppose donc le langage cible muni d'une notion d'ensemble et doté des opérations suivantes : (i) union et intersection entre un ensemble variable et un ensemble fixe ; (ii) test d'appartenance d'une ressource fixe à un ensemble variable, avec échec à l'exécution en cas de non-appartenance. On constate que le codage de $r.e$ accroît en général l'ensemble s, tandis que celui de $p.e$ le restreint. On constate par ailleurs que s devient un argument supplémentaire de chaque fonction.

Reste à se donner un système de types pour le langage cible, et à effectuer la composition du codage et de ce système. On peut par exemple adopter le système de types de Damas et Milner, étendu à l'aide de rangées [1], lesquelles permettent un typage fin des ensembles. (Un ensemble n'est qu'un enregistrement particulier, où les champs n'ont pas de contenu, et où seule leur présence ou absence compte.) Parce que l'image de toute fonction à travers le codage est une fonction dotée d'un argument supplémentaire s, le type de toute fonction fait apparaître un argument supplémentaire, à savoir un type ensemble, constitué d'une rangée, qui indique quelles sont les ressources auxquelles la fonction est susceptible d'accéder. C'est exactement ce que nous souhaitions. De plus, nous avons gagné, outre une preuve de correction simple, le polymorphisme et le mécanisme d'inférence de types du système de types de Damas et Milner.

On peut, si on le souhaite, aller plus loin, par exemple en se basant sur une instance de HM(X) dotée de sous-typage, ou encore en employant des contraintes conditionnelles pour modéliser une construction supplémentaire, notée $r\,?\,e : e$, qui permet de tester dynamiquement si une ressource donnée est ou non accessible. Ces idées sont détaillées dans [112, 113].

Chapitre 5

Analyse de flots d'information

Ce chapitre commente une partie du matériau publié dans [141–143].

J'AI SUGGÉRÉ, dans le chapitre précédent, comment appliquer la technique de typage à travers un codage pour définir une analyse de flots d'information [111]. Ce résultat est intéressant, mais ne s'adresse qu'à un langage de programmation purement fonctionnel, comme les travaux précédents [128–130]. Pour obtenir un résultat susceptible d'avoir un impact pratique, il faut étendre l'analyse à un langage de programmation plus riche. Myers [144] a défini et implémenté une analyse pour l'ensemble du langage Java, ce qui représente un travail important, mais celle-ci n'est accompagnée d'aucune preuve. En collaboration avec Vincent Simonet, j'ai donc défini, implémenté et prouvé la correction d'une analyse de flots d'information pour le langage ML. Au noyau fonctionnel pur, nous avons ajouté principalement le traitement des effets de bord, c'est-à-dire références et exceptions. D'autres traits du langage, par exemple les types algébriques et le langage de modules, ont été implémentés, mais n'ont posé que peu de problèmes théoriques.

Naturellement, j'ai initialement voulu conserver la méthode de typage à travers un codage. En ce qui concerne l'ajout des références, je pensais pouvoir étendre la sémantique étiquetée d'Abadi *et al.* [131] et définir ainsi une analyse dynamique de flots d'information. Par ailleurs, pour ajouter les exceptions, je pensais m'appuyer sur le codage classique des exceptions en termes de sommes, que j'ai mentionné au chapitre 4. Or, ces deux idées se sont révélées impossibles ou difficiles à mettre en application. Nous avons donc dû procéder différemment.

5.1 Références

5.1.1 Échec du typage à travers un codage

Denning [122] l'avait affirmé : l'analyse *dynamique* de flots d'information n'est pas possible si le langage de programmation considéré est doté de variables modifiables ou de références. Le cœur du problème est le suivant : un flot d'information peut être provoqué non seulement par la modification du contenu d'une référence, mais aussi par sa *non*-modification. Le fragment de code suivant, qui emploie la syntaxe de ML, en donne un exemple :

$$\textbf{\textit{if}}\ !x = 1\ \textbf{\textit{then}}\ y := 1$$

Si, après exécution, y admet une valeur autre que 1, alors on peut en déduire qu'il en va de même de x. Donc, dans ce cas, l'exécution de ce fragment de code a transféré de l'information de x vers y, bien que la valeur de y n'ait pas été modifiée ! Denning, s'appuyant sur cette remarque, propose un fragment de

code plus élaboré :

$$
\begin{array}{l}
\textbf{\textit{let}}\ \textit{copy}\ x = \\
\quad \textbf{\textit{let}}\ y = \textit{ref}\ 0\ \textbf{\textit{in}} \\
\quad \textbf{\textit{let}}\ z = \textit{ref}\ 0\ \textbf{\textit{in}} \\
\quad \textbf{\textit{if}}\ !x = 0\ \textbf{\textit{then}} \\
\qquad y := 1; \\
\quad \textbf{\textit{if}}\ !y = 0\ \textbf{\textit{then}} \\
\qquad z := 1; \\
\quad !z
\end{array}
$$

Il est facile de vérifier que, si x vaut 0 ou 1, alors *copy* x vaut x. Il y a donc flot d'information de x vers z. Il semble impossible pour une analyse dynamique de flots d'information de détecter cela. En effet, si x vaut 0, alors, parce que y est modifiée après qu'on ait consulté x, on détecte un flot de x vers y. Parce que z n'est pas modifiée, aucun flot vers z n'est détecté. Symétriquement, si x vaut 1, alors on détecte un flot de y vers z, et aucun flot vers y. Pour détecter l'existence d'un flot de x vers z, il semble nécessaire de combiner ces deux observations, ce que seule une analyse *statique* peut réaliser.

Une analyse dynamique obtenue en ajoutant la gestion des références, de façon naïve, à l'analyse d'Abadi *et al.* [131], est donc incorrecte. Pourtant, on pourrait espérer que l'analyse statique obtenue en la composant avec un système de types classique, selon la méthode exposée au chapitre 4, soit correcte. En effet, le propre d'un système de types est d'examiner *toutes* les exécutions possibles du programme qu'on lui soumet, et de proposer un type qui satisfait simultanément toutes les contraintes qui en résultent. Dans le cas de *copy*, le système détecterait l'existence d'un flot d'information de x vers y et d'un autre de y vers z, et, parce qu'il résoud simultanément les contraintes qui en découlent, attribuerait correctement à z un niveau d'information (supérieur ou) égal à celui de x. Cette remarque m'a poussé à tenter de conserver la technique de typage à travers un codage, quitte à lui apporter des adaptations significatives, puisque le codage, pris isolément, n'est plus correct.

Mais, après quelques temps, il est apparu un autre problème, qui nous a conduit à abandonner totalement toute approche basée sur l'ajout d'étiquettes à la sémantique, dans le style d'Abadi *et al.* [131]. La propriété à établir, à savoir la non-interférence, exige de comparer deux exécutions d'un même programme, à partir de données différentes, et de démontrer qu'elle produisent le même résultat. Que le langage de programmation soit doté de sa sémantique standard ou d'une sémantique étiquetée, chacune de ces exécutions constitue un objet mathématique distinct. Les références allouées au cours de la réduction sont des entités *internes* à ces objets, sans identité externe. Il est donc impossible de garder trace de la correspondance entre références allouées au cours de la première et deuxième exécutions, respectivement – ce qui serait pourtant nécessaire pour pouvoir donner un sens à un énoncé tel que « le contenu de la référence m est le même dans les deux exécutions. »

5.1.2 Sémantique de ML^2

On peut probablement imaginer plusieurs solutions à ce problème. Celle proposée par Vincent Simonet et moi-même permet aux *deux* exécutions considérées de partager un *unique* tas (*store*), de sorte qu'une adresse (*location*) donnée m a bien la même signification de part et d'autre. Pour cela, nous définissons une extension du langage de programmation considéré, ici le noyau de ML, et la dotons d'une sémantique *ad hoc*.

L'idée est de se doter d'un langage capable de représenter des *paires* d'expressions du langage de départ. Ce langage, nommé ML^2, contient donc une construction, notée $\langle e_1 \mid e_2 \rangle$, où e_1 et e_2 sont des expressions de ML. De plus, il contient également toutes les constructions habituelles de ML. La syntaxe de ML^2 est donc définie par

$$e ::= \ldots \mid \langle e \mid e \rangle$$

sous la condition que les crochets ne peuvent être emboîtés. Ceci permet de représenter de façon fine les points communs et les différences entre deux expressions de ML. Par exemple, les termes $(1, \langle 2 \mid 3 \rangle)$ et $\langle (1,2) \mid (1,3) \rangle$ ont les mêmes *projections*, c'est-à-dire qu'ils représentent tous deux la paire formée des expressions $(1,2)$ et $(1,3)$. Cependant, le premier de ces termes est plus précis : sa structure même met en évidence le fait que le contexte $(1, [])$ est partagé.

Ce mécanisme s'applique également aux tas. Un unique tas de ML^2, qui aux adresses associe des valeurs contenant éventuellement des crochets, représente la paire de deux tas de ML.

On peut doter ML^2 d'une sémantique opérationnelle. On y trouve d'abord les règles qui définissent la sémantique de ML. Celles-ci prennent une signification nouvelle : elles permettent à présent une réduction *synchrone* des deux expressions ML représentées par une unique expression ML^2, lorsque celles-ci présentent la même structure. Ensuite, vient une règle indiquant que la composante gauche d'un crochet peut se réduire indépendamment de sa composante droite, et vice-versa. Cette règle permet une réduction *indépendante* des deux expressions ML considérées, lorsque celles-ci n'ont pas du tout la même structure. Enfin, vient une série de règles applicables lorsque la structure nécessaire à la réduction est en partie partagée. Par exemple, on se donne la règle suivante :

$$\langle v_1 \mid v_2 \rangle\, v \longrightarrow \langle v_1 \lfloor v \rfloor_1 \mid v_2 \lfloor v \rfloor_2 \rangle$$

Cette règle s'applique lorsque les deux programmes s'apprêtent simultanément à effectuer un appel de fonction, avec un même argument v, et lorsque la fonction appelée n'est pas la même – v_1 d'un côté, v_2 de l'autre. La règle n'a aucun effet calculatoire : on peut vérifier que ses membres gauche et droit ont les mêmes projections, à savoir $v_1 \lfloor v \rfloor_1$ et $v_2 \lfloor v \rfloor_2$. Son seul effet est donc de réécrire le terme de gauche en un terme moins précis, au sens ci-dessus, c'est-à-dire un terme où une partie de l'information de partage a été perdue. Cette règle est analogue à *(lift)* (§4.2).

La sémantique de ML^2 est conçue de façon à satisfaire les propriétés suivantes :

Théorème 5.1 (Correction) *Les projections d'une réduction de ML^2 sont des réductions de ML.* ◇

Théorème 5.2 (Complétude) *Si les deux projections d'une expression de ML^2 se réduisent dans ML jusqu'à un résultat, alors celle-ci se réduit dans ML^2 jusqu'à un résultat.* ◇

La combinaison de ces deux propriétés signifie, en bref, que ML^2 est un outil adéquat pour comparer deux exécutions de ML, pourvu que celles-ci terminent : en effet, ces deux exécutions sont alors les projections d'une unique exécution de ML^2.

5.1.3 Typage de ML^2 et non-interférence

Parce que la technique de typage à travers un codage a été abandonnée, l'analyse statique de flots d'information doit être définie explicitement, sous forme d'un système de types pour ML. On étend en fait ce système de types à ML^2, de façon à pouvoir exprimer la préservation d'un invariant au cours de la réduction de *deux* programmes reliés.

La propriété de non-interférence à établir est la suivante. Soit e un programme de ML ayant une variable libre x. Supposons que, sous l'hypothèse que x est de type « secret, » e admet le type « entier public. » Alors, quelles que soient les valeurs v_1 et v_2 par lesquelles on remplace x, l'évaluation de e, si elle termine, produit le même résultat.

Pour démontrer ce théorème, on doit comparer les exécutions de $[v_1/x]e$ et de $[v_2/x]e$, dans l'hypothèse où elles terminent. Grâce aux théorèmes 5.1 et 5.2, il suffit de considérer l'expression de ML^2 $[\langle v_1 \mid v_2 \rangle / x]e$. Son exécution produit nécessairement un résultat, dont les projections sont les résultats des deux exécutions que l'on souhaite comparer. Pour établir que ceux-ci sont identiques, il suffit de démontrer que celui-là ne contient aucun crochet.

On voit que les crochets, dans ML^2, sont employés pour représenter les informations « secrètes. » En effet, ce que l'on considère initialement comme « secret » est la valeur de x, à savoir v_1 d'un côté, v_2 de l'autre. Dans ML^2, cette valeur est représentée par $\langle v_1 \mid v_2 \rangle$. À l'issue de l'évaluation, au contraire, on souhaite montrer que le résultat obtenu n'est pas un « secret, » et, pour cela, on montre que ce n'est pas un crochet. Il est donc naturel que le système de types de ML^2 attribue à tous les crochets le niveau d'information « secret. » Cela est spécifié par la règle de typage associée à la construction $\langle \cdot \mid \cdot \rangle$, que je ne donne pas ici.

En ce qui concerne toutes les autres constructions, le système de types de ML^2 est identique à celui que l'on a défini pour ML. (Notons bien qu'il ne s'agit pas là du système de types de Damas et Milner, mais de l'analyse de flots d'information dont nous souhaitons établir la correction.) Ses règles sont conçues de façon à satisfaire la propriété suivante :

Théorème 5.3 (Auto-réduction) *La réduction de ML^2 préserve le typage.* ◇

Je peux maintenant terminer l'esquisse de la preuve de non-interférence, que j'ai suspendue plus haut. D'abord, la valeur $\langle v_1 \mid v_2 \rangle$ admet le type « secret. » Ensuite, sous l'hypothèse que x est de type « secret, » e admet le type « entier public. » Par une propriété de substitution, vérifiée par construction dans la plupart des systèmes de types et dans celui de ML^2 en particulier, ces deux faits impliquent que $[\langle v_1 \mid v_2 \rangle / x]e$ admet le type « entier public. » D'après le théorème 5.3, son résultat admet également le type « entier public. » Or, un résultat entier est soit une constante k, soit un crochet $\langle k_1 \mid k_2 \rangle$. Ici, le second cas est impossible, parce qu'un crochet ne peut être de type « public. » Par conséquent, ce résultat est nécessairement de la forme k, et ne contient aucun crochet. C'est là ce que je souhaitais démontrer.

Je rappelle, pour mémoire, l'énoncé (simplifié) de non-interférence que l'on obtient ainsi :

Théorème 5.4 (Non-interférence) *Soit e un programme de ML ayant une variable libre x. Supposons que, sous l'hypothèse que x est de type « secret, » e admet le type « entier public. » Alors, quelles que soient les valeurs v_1 et v_2 par lesquelles on remplace x, l'évaluation de e, si elle termine, produit le même résultat.* ◇

Bien entendu, l'exposé ci-dessus est très informel, et ne donne pas les règles de typage qui définissent l'analyse de flots d'information. Cette omission ne me semble pas très importante, parce que ces règles ne sont pas difficiles à concevoir : elles généralisent celles de Volpano et Smith [125], et sont similaires à celles données par d'autres auteurs [129, 144–146]. Il était beaucoup plus difficile de structurer la preuve de non-interférence de façon élégante, et c'est pourquoi j'ai insisté ici sur l'architecture de cette preuve. Grâce à l'outil ML^2, la partie habituellement la plus délicate de cette preuve revient ici à établir une propriété d'auto-réduction (théorème 5.3), dont la démonstration ne présente pas de difficulté particulière.

Cette technique de preuve, baptisée *technique des crochets*, a été transportée par moi-même au π-calcul, à titre d'expérimentation [143]. Le résultat obtenu reste modeste, car le système de types étudié dans cet article est d'une expressivité très limitée. Il faudrait, pour l'améliorer, y incorporer une notion de *linéarité*, similaire à celles étudiées par Zdancewic et Myers [146] dans un cadre séquentiel et par Honda, Vasconcelos et Yoshida [147] et Honda et Yoshida [148] dans le cadre du π-calcul. Il n'est pas certain que la technique des crochets permette la preuve de correction d'une analyse dotée de types linéaires. Cette technique est également utilisée par Simonet [149], et a été reprise par d'autres auteurs, par exemple Zheng et Myers [150].

Il faut noter que, parce que le système de types est polymorphe et doit permettre l'inférence de types, il n'est pas possible de s'appuyer sur un langage et une sémantique explicitement typés. Chez Zdancewic et Myers [146], a contrario, chaque application de la primitive *ref* est annotée par un niveau d'information fixé, et la sémantique opérationnelle est définie de telle façon que la modification du contenu d'une référence provoque une erreur à l'exécution si l'on tente d'y introduire une information d'un niveau incompatible avec celui qui a été fixé. Cette approche simple est incompatible avec le polymorphisme. En effet, on ne dispose plus d'une unique primitive *ref* polymorphe, mais d'une famille de primitives ref_ℓ, indexée par les niveaux d'information ℓ. La technique des crochets présente l'avantage d'éviter cet écueil.

Enfin, notons que l'approche semi-syntaxique (§2.4.3) a été utilisée pour alléger sensiblement la preuve. Dans un premier temps, on démontre la correction d'une formulation « algébrique » du système de types, puis, dans un second temps, on montre que la version syntaxique, à base de contraintes, est correcte vis-à-vis de la première. La seconde étape est standard et ne présente aucune difficulté nouvelle vis-à-vis de son homologue dans HM(X), ce qui permet de se concentrer sur la première.

5.2 Exceptions

Dans un langage de programmation comme Java ou ML, les exceptions constituent un mécanisme de transmission d'information, et doivent donc être prises en compte par l'analyse de flots d'information. Myers [144] s'appuie sur le système de types de Java, qui exige que la liste des exceptions que chaque méthode est susceptible de lever soit explicitement fournie par le programmeur. Il enrichit cette liste

en exigeant que chaque exception soit accompagnée par le niveau de l'information potentiellement transmise lorsque cette exception est levée.

Le traitement des exceptions chez Myers est relativement précis, et il semble difficile d'aller plus loin sans augmenter considérablement la complexité du système. Aussi Vincent Simonet et moi-même n'avons pas cherché à définir une analyse plus expressive, mais simplement à prouver la correction d'une analyse de précision comparable, dans le cadre de ML. Conformément à la tradition de ML, cependant, et contrairement à l'analyse de Myers, notre système n'exige aucune annotation de la part du programmeur, et est polymorphe.

J'ai songé, une nouvelle fois, à adopter l'approche du typage à travers un codage. L'idée est de traduire un langage doté d'exceptions vers un langage dénué d'exceptions, mais doté de types sommes, et pour lequel une analyse de flots d'information est disponible. Le langage étudié plus haut (§5.1) convient pour cela, à condition d'y incorporer un traitement suffisamment fin des sommes. Le traitement classique des sommes [111, 129, 130], où chaque type somme porte une seule étiquette, n'est certainement pas assez précis. En effet, il est clair que chaque branche d'un type somme doit porter une étiquette, de façon à obtenir, à travers le codage, une analyse où chaque nom d'exception se voit associer une étiquette.

Ce traitement fin des sommes, ainsi que le typage des exceptions qui en découle à travers le codage, ont été étudiés par Simonet [149]. Cette étude, intéressante d'un point de vue théorique, nous a conduits à préférer formuler directement une analyse d'exceptions pour ML, car le détour par les types sommes nous a semblé trop long. Nous avons donc employé la technique des crochets pour prouver de façon directe la correction de notre analyse de flots d'information en présence d'exceptions [141].

La version finale de notre analyse [142] a été simplifiée grâce à une légère restriction de l'expressivité du langage : les exceptions n'y sont plus des valeurs de première classe. Il reste donc possible d'écrire *raise* $(E\,v)$, si E est un nom d'exception, mais l'idiome *raise* x est supprimé. Quelques constructions courantes, comme *try/finally* et *try/propagate*, sont ajoutées pour compenser partiellement cette perte d'expressivité. En termes de typage, cela permet d'éliminer les contraintes conditionnelles, auxquelles on recourait dans [141] pour typer *raise* x. Cette simplification était nécessaire pour permettre le développement d'un analyseur modulaire, puisque, comme on l'a dit au chapitre 3, la comparaison entre schémas de types inférés et déclarés est difficile en présence de contraintes conditionnelles.

5.3 Quelques mots de Flow Caml

Vincent Simonet a implanté l'analyse évoquée ci-dessus dans Flow Caml [151, 152], un sous-ensemble du langage Objective Caml doté d'un système de types plus exigeant, dont les types reflètent des flots d'information potentiels. Les programmes Flow Caml bien typés sont traduits, par simple effacement des informations superflues, vers Objective Caml.

Le système de types de Flow Caml est une variante de HM(X). Ce n'en est pas une instance, car les règles de typage ont été modifiées pour garder trace des flots d'information. Les contraintes de sous-typage y sont interprétées de façon structurelle : l'ordre sur les types est engendré par l'ordre sur les niveaux d'informations qui les décorent. Le fait que deux types comparables ont nécessairement la même structure implique que tout programme bien typé au sens de Flow Caml est également bien typé au sens d'Objective Caml.

Le sous-typage structurel permet également un affichage original des schémas de types contraints sous forme graphique, imaginé et implanté par Simonet [151]. En bref, les contraintes peuvent être représentées sous forme d'arcs reliant certains nœuds d'un squelette, où un « squelette » est grosso modo un type dénué de décorations, c'est-à-dire un type au sens d'Objective Caml.

Outre le prédicat de sous-typage, le langage de contraintes s'appuie sur deux prédicats auxiliaires, appelées *gardes* et notées $\lhd$ et $\blacktriangleleft$ dans [142]. Chacun de ces prédicats relie un niveau d'information et un type, en imposant une relation d'ordre entre le premier et certains des niveaux qui décorent le second. Parce que l'interprétation de ces prédicats dépend de la structure du type auquel ils s'appliquent (flèche, produit, somme, etc.), ils ne sont pas définissables en termes de la relation de sous-typage. Cette caractéristique originale vient compliquer légèrement le processus de résolution de contraintes, mais permet d'alléger la structure des types : en particulier, nous avons pu, grâce à cela, n'ajouter aucune

annotation au constructeur de types produit ($\times$). Ceci constitue une innovation vis-à-vis des travaux précédents, qui adoptent en général l'hypothèse simplificatrice selon laquelle tout constructeur de types porte une annotation.

Le solveur de contraintes de Flow Caml gère les contraintes de sous-typage structurel, les « gardes, » et les rangées. Il a été implanté et formalisé par Vincent Simonet [153]. Il s'agit, à ma connaissance, de la première implantation réaliste d'une variante de ML dotée de sous-typage et d'inférence de types.

Chapitre 6

Types de données algébriques gardés

Ce chapitre commente une partie du matériau publié dans [154] et contenu dans [155].

AU COURS DES CHAPITRES PRÉCÉDENTS, j'ai présenté différents systèmes de types basés par exemple sur les contraintes de sous-typage, structurel ou non, les contraintes conditionnelles, ainsi que sur certaines formes de contraintes plus exotiques, comme les « gardes » (chapitre 5). Or, si ces systèmes sont théoriquement au point, il semble difficile de les mettre en œuvre pour accroître l'expressivité d'un langage de programmation généraliste, et ce pour diverses raisons.

D'une part, leur formes résolues, même simplifiées, restent souvent difficiles à lire ; or, elles apparaissent au sein des schémas de types inférés. C'est le cas des contraintes de sous-typage non structurel ainsi que des diverses formes de contraintes conditionnelles. D'autre part, la présence de contraintes peut rendre difficile la comparaison automatique entre schémas de types déclarés et inférés. C'est le cas des contraintes de sous-typage non structurel, pour lesquelles le problème reste ouvert [84], ainsi que des contraintes conditionnelles, pour lesquelles il est coûteux [89]. Dans le cas du sous-typage structurel, on peut afficher les schémas de types contraints sous forme graphique (§5.3). Néanmoins, la lecture d'un tel graphe reste un exercice délicat. Dans le cas de Flow Caml [152], le surcroît de complexité du système de types est justifié par la finesse qu'exige l'analyse de flots d'information. Dans le cas d'un langage de programmation généraliste, cependant, l'emploi de contraintes de sous-typage structurel semble discutable, même si certains l'ont mis en œuvre avec succès [56, 58, 59].

Ces remarques pessimistes signifient-elles que l'inférence de types à base de contraintes n'est qu'un exercice de style, sans véritable espoir de parvenir à une utilisation pratique ? Je ne le pense pas. Elles doivent seulement nous conduire à étudier des langages de contraintes pour lesquels les principaux problèmes de décision (satisfiabilité, implication) sont algorithmiquement peu coûteux et surtout dont les formes résolues sont aussi simples que possible, de façon à ce que les schémas de types inférés puissent être présentés au programmeur sous une forme naturelle.

L'objet de ce chapitre est de décrire une telle situation, en prenant pour exemple l'ajout de *types de données algébriques gardés* [156] à ML. Le langage de contraintes nécessaire pour effectuer l'inférence de types reste alors basé uniquement sur les équations entre types, mais est enrichi par l'introduction d'une forme restreinte d'implication. Les formes résolues restent les mêmes que dans le cas de contraintes d'unification classiques, ce qui signifie que les schémas de types conservent la même forme qu'en ML : le formalisme des contraintes reste entièrement invisible pour le programmeur, et ne sert qu'à permettre une formalisation plus élégante et une meilleure compréhension du système de types.

6.1 Présentation

Je rappelle d'abord comment on définit un type de données algébrique, puis j'explique comment on parvient à la notion plus générale de type de données algébrique gardé.

Types de données algébriques Soit ϵ un constructeur de types de données algébriques, paramétré par un vecteur de variables de types distinctes $\bar{\alpha}$. Soit K l'un des constructeurs de données associés à ϵ. Le

schéma de types (clos) attribué à K, que l'on peut déduire de la déclaration de ϵ, doit être de la forme

$$K :: \forall\bar{\alpha}.\tau_1 \times \cdots \times \tau_n \to \epsilon(\bar{\alpha}), \tag{1}$$

où n est l'arité de K. Alors, la règle qui gouverne le typage du filtrage peut être résumée comme suit : si le motif $K\,(x_1, \ldots, x_n)$ filtre une valeur de type $\epsilon(\bar{\alpha})$, alors la variable x_i dénote une valeur de type τ_i.

Par exemple, un type de données algébrique $tree(\alpha)$, décrivant les arbres binaires dont les nœuds internes sont étiquetés par des valeurs de type α, pourrait être déclaré comme suit :

$$\begin{aligned} \mathit{Leaf} &:: \forall\alpha.tree(\alpha), \\ \mathit{Node} &:: \forall\alpha.tree(\alpha) \times \alpha \times tree(\alpha) \to tree(\alpha). \end{aligned}$$

Lorsque le motif Leaf filtre une valeur de type $tree(\alpha)$, aucune variable n'est liée. Lorsque le motif $Node(l, v, r)$ filtre une telle valeur, les variables l, v, et r sont liées à des valeurs de types respectifs $tree(\alpha)$, α, et $tree(\alpha)$.

Types existentiels à la Läufer et Odersky Il est possible d'imaginer des extensions de ML permettant des formes plus libérales de déclarations de types de données algébriques. Considérons par exemple la façon dont Läufer et Odersky [157] introduisent les types existentiels dans ML. Le schéma de types associé à un constructeur de données peut alors être de la forme

$$K :: \forall\bar{\alpha}\bar{\beta}.\tau_1 \times \cdots \times \tau_n \to \epsilon(\bar{\alpha}). \tag{2}$$

La nouveauté réside dans le fait que les types des arguments, à savoir $\tau_1, \ldots, \tau_n$, peuvent contenir des variables de types, à savoir $\bar{\beta}$, qui ne sont pas des paramètres du constructeur de types de données algébriques ϵ. Alors, la règle qui gouverne le typage du filtrage devient : si le motif $K\,(x_1, \ldots, x_n)$ filtre une valeur de type $\epsilon(\bar{\alpha})$, alors *il existe* des types inconnus $\bar{\beta}$ tels que la variable x_i dénote une valeur de type τ_i.

Par exemple, un type de données algébrique key, décrivant des paires d'une clé et d'une fonction des clés vers les entiers, où le type des clés reste abstrait, pourrait être déclaré comme suit :

$$Key :: \forall\beta.\beta \times (\beta \to int) \to key.$$

Les valeurs $Key\,(3, \lambda x.5)$ et $Key\,([1; 2; 3], length)$ admettent toutes deux le type key. Lorsque le motif $Key\,(v, f)$ filtre l'une quelconque de ces valeurs, les variables v et f sont liées à des valeurs de types respectifs β et $\beta \to int$, pour un β abstrait, ce qui permet, par exemple, d'évaluer $(f\,v)$, mais interdit de considérer v comme un entier ou comme une liste d'entiers, ce qui ne serait pas sûr.

Types de données algébriques gardés On peut aller plus loin en autorisant les constructeurs de données à recevoir un schéma de types *contraint* :

$$K :: \forall\bar{\alpha}\bar{\beta}[D].\tau_1 \times \cdots \times \tau_n \to \epsilon(\bar{\alpha}). \tag{3}$$

Ici, D est une contrainte. La valeur $K\,(v_1, \ldots, v_n)$, où chaque v_i est de type τ_i, est bien typée, et admet le type $\epsilon(\bar{\alpha})$, seulement si les variables de types $\bar{\alpha}\bar{\beta}$ satisfont la contrainte D. En contrepartie de cette règle de *construction* restreinte, la règle qui gouverne le typage de la *destruction*, c'est-à-dire du filtrage, devient plus flexible : si le motif $K\,(x_1, \ldots, x_n)$ filtre une valeur de type $\epsilon(\bar{\alpha})$, alors il existe des types inconnus $\bar{\beta}$ *satisfaisant* D tels que la variable x_i dénote une valeur de type τ_i. Ainsi, le succès d'un test *dynamique*, à savoir le filtrage, permet de glaner une information *statique* supplémentaire, exprimée par D, dans la portée d'une branche. J'emploie la terminologie « types de données algébriques gardés » parce que les schémas de types associés aux constructeurs sont gardés, c'est-à-dire contraints.

Dans le cas qui nous intéresse ici, à savoir une extension de ML, D doit être une contrainte d'unification, c'est-à-dire une conjonction d'équations entre types. Alors, on peut en fait associer aux constructeurs de données des schémas de types de la forme suivante :

$$K :: \forall\bar{\beta}.\tau_1 \times \cdots \times \tau_n \to \epsilon(\bar{\tau}). \tag{4}$$

Dans cette forme, aucune contrainte n'apparaît, mais le constructeur de types ϵ peut être appliqué à un vecteur de types arbitraires, plutôt qu'à un vecteur de variables de types distinctes $\bar{\alpha}$. Il n'est pas difficile de vérifier que les formes (3) et (4) sont équivalentes.

Je fais parfois référence aux types de données algébriques introduits par une déclaration de la forme (4) en tant que *types inductifs*, parce qu'ils sont très proches de ceux que l'on trouve dans le calcul des constructions inductives [158, 159]. La seule différence réside dans le fait que les véritables types inductifs sont soumis à une condition de positivité, qui garantit la cohérence logique, tandis que, dans le cadre d'un langage de programmation, cette restriction n'est pas nécessaire. Ces types sont également connus sous les noms de *constructeurs de types de données récursifs gardés* par Xi *et al.* [156] et de *types fantômes de première classe* par Cheney et Hinze [160]. Un cas particulier de ce mécanisme a été utilisé auparavant par Crary, Weirich et Morrisett [161].

6.2 Application à la défonctionalisation

À quoi peuvent bien servir les types de données algébriques gardés ? Leur trait distinctif est de permettre à un test *dynamique* de fournir une information *statique* supplémentaire. On peut en tirer parti de multiples façons.

Par exemple, Crary *et al.* [161] utilisent un type de données algébrique gardé particulier, noté R, pour traduire un langage doté d'*analyse de types intensionnelle*, c'est-à-dire d'une construction ***typecase***, vers un langage standard, doté d'une simple construction ***case***. L'idée est de représenter les types, pendant l'exécution, par des valeurs particulières, que l'on peut analyser de façon classique. Pour que les programmes qui emploient cette convention puissent être considérés comme bien typés, il faut que l'analyse de la *représentation dynamique* d'un type, à l'aide de la construction ***case***, permette d'obtenir des informations *statiques* sur le type *lui-même*. La définition du type R est conçue pour permettre cela. Elle est formulée de telle sorte que, pour tout type sans variables τ, le type $R\,\tau$ contient une seule valeur, laquelle est précisément la représentation dynamique du type τ. Pour cette raison, on dit que $R\,\tau$ est un *type singleton*.

Cheney et Hinze [162] tentent de simuler le type R dans un langage dénué de types de données algébriques gardés, en l'occurence Haskell, ce qui donne lieu à quelques acrobaties intéressantes. Les mêmes auteurs [160] donnent ensuite divers exemples d'utilisation des types de données algébriques gardés.

Xi *et al.* [156] généralisent les règles de typage associées à R et proposent la notion générale de type de données algébrique gardé. Ils donnent d'autres illustrations de l'utilité de cette notion. Une version préliminaire de leur article suggère en particulier que les types de données algébriques gardés autorisent un nouveau schéma de compilation des *type classes* de Haskell [28]. L'idée est de représenter un dictionnaire non pas par un enregistrement contenant une clôture pour chaque méthode, comme c'est le cas habituellement [44, 46, 163, 164], mais par une simple structure de données, laquelle est ensuite interprétée, à l'aide d'une construction ***case***, lorsqu'on souhaite appeler une des méthodes qu'elle représente. J'ai récemment étudié deux variantes de ce schéma de compilation original, et me propose de les publier d'ici peu.

Défonctionalisation De façon indépendante, j'ai démontré, en collaboration avec Nadji Gauthier, que les types de données algébriques gardés permettent de considérer la *défonctionalisation*, une transformation de programmes classique due à Reynolds [165, 166], comme produisant des programmes *bien typés*, y compris en présence de polymorphisme. Ce résultat est nouveau : l'effet de la défonctionalisation sur les types n'était jusqu'ici compris qu'en l'absence de polymorphisme [167, 168] ou bien, dans le cas d'un langage doté de polymorphisme à la Hindley et Milner, moyennant son élimination préalable par *monomorphisation* [169–172].

En quoi consiste la défonctionalisation ? Son but est d'éliminer l'emploi des fonctions en tant que valeurs de première classe. Pour cela, on attribue à chaque λ-abstraction – donc à chaque fragment de code – du programme source une étiquette distincte. On code alors chaque valeur fonctionnelle par une *clôture* composée d'une telle étiquette et d'un environnement. Lorsque l'on souhaite appliquer une

fonction, on examine l'étiquette et on en déduit vers quel fragment de code il faut sauter. La défonctionalisation est donc très similaire à la *closure conversion*, à la différence près que cette dernière emploie des pointeurs de code en guise d'étiquettes, ce qui permet de remplacer l'examen explicite de l'étiquette par un simple saut indirect. En termes de typage, cette différence est sensible : l'image d'un type fonctionnel à travers la défonctionalisation est un type somme ou un type de données algébrique, tandis que son image à travers la *closure conversion* emploie types fonctionnels et types existentiels [173–175].

Interaction avec le polymorphisme Pourquoi est-il difficile de définir la défonctionalisation pour un λ-calcul typé polymorphe, par exemple le système F [176, 177] ? Le problème réside dans la définition de $apply$, la fonction qui forme le cœur de tout programme défonctionalisé. Ses paramètres sont une clôture f et une valeur arg ; sa tâche est de simuler l'application de la fonction source codée par f à la valeur source codée par arg, et d'en renvoyer le résultat. En d'autres termes, si $[\![e]\!]$ dénote l'image d'une expression source e à travers la défonctionalisation, alors on souhaite définir $[\![e_1\, e_2]\!]$ comme $apply\, [\![e_1]\!]\, [\![e_2]\!]$. Supposons à présent que la défonctionalisation préserve le typage, et que $[\![\tau]\!]$ dénote l'image d'un type source τ à travers la défonctionalisation. Alors, si e_1 admet le type $\tau_1 \to \tau_2$ et si e_2 admet le type τ_1, nous constatons que, pour que $apply\, [\![e_1]\!]\, [\![e_2]\!]$ soit bien typée, $apply$ doit avoir le type $[\![\tau_1 \to \tau_2]\!] \to [\![\tau_1]\!] \to [\![\tau_2]\!]$. De plus, puisque e_1 est ici arbitraire, ceci doit valoir pour tous types τ_1 et τ_2. La façon la plus naturelle de satisfaire cette exigence est de faire en sorte que $apply$ ait le type $\forall\alpha_1\alpha_2.[\![\alpha_1 \to \alpha_2]\!] \to \alpha_1 \to \alpha_2$ et que $[\![\cdot]\!]$ commute avec toute substitution de variables de types par des types.

Considérons maintenant la définition de la fonction $apply$. Elle doit être de la forme

$$\Lambda\alpha_1.\Lambda\alpha_2.\lambda f : [\![\alpha_1 \to \alpha_2]\!].\lambda arg : \alpha_1.\mathsf{case}\ f\ \mathsf{of}\ \vec{c}$$

où $\vec{c}$ contient une clause pour chaque étiquette, c'est-à-dire pour chaque λ-abstraction du programme source. Le membre droit de chacune de ces clauses est le corps de la λ-abstraction associée, renommé de façon à ce que son paramètre formel soit arg. En guise d'illustration, supposons que le programme source contient les λ-abstractions $\lambda x.x + 1$ et $\lambda x.\mathsf{not}\, x$, dont les types sont $int \to int$ et $bool \to bool$, et dont les étiquettes sont *succ* et *not*, respectivement. Alors, $\vec{c}$ doit contenir les clauses suivantes :

$$\begin{array}{rcl} succ & \mapsto & arg + 1 \\ not & \mapsto & \mathsf{not}\ arg \end{array}$$

Cependant, dans le système F, ces clauses sont incompatibles : elles font des hypothèses différentes quant au type de arg, et produisent des valeurs de types différents. En fait, pour que $apply$ soit bien typée, toute λ-abstraction du programme source doit produire une valeur de type α_2, sous l'hypothèse que son argument est de type α_1. En l'absence de toute autre hypothèse à propos de α_1 et α_2, cela revient à exiger que toutes les λ-abstractions du programne source admettent le type $\forall\alpha_1\alpha_2.\alpha_1 \to \alpha_2$, ce qui ne peut pas être le cas en général ! Ceci explique pourquoi il ne semble pas possible de définir une notion de défonctionalisation qui préserve le typage dans le cadre du système F.

La (non-)solution usuelle La réponse communément apportée à ce problème dans le cas simplement typé [167–172] consiste à spécialiser $apply$. Au lieu de définir une unique fonction polymorphe, on introduit une famille de fonctions monomorphes, de type $[\![\tau_1 \to \tau_2]\!] \to [\![\tau_1]\!] \to [\![\tau_2]\!]$, où τ_1 et τ_2 parcourent les types sans variables. Du coup, la définition de $apply_{\tau_1 \to \tau_2}$ peut se contenter de ne traiter que les étiquettes dont les λ-abstractions associées sont de type $\tau_1 \to \tau_2$. Pour reprendre l'exemple précédent, la définition de $apply_{int \to int}$ doit contenir un cas pour *succ*, mais aucun pour *not*. Inversement, la définition de $apply_{bool \to bool}$ traite *not*, mais pas *succ*. Il est à présent facile de vérifier que toutes les clauses qui figurent dans la définition de $apply_{\tau_1 \to \tau_2}$ sont compatibles au sens du typage, de sorte que la fonction est bien typée. Enfin, en exploitant le fait que e_1 doit avoir un type *sans variables* de la forme $\tau_1 \to \tau_2$, on définit $[\![e_1\, e_2]\!]$ comme $apply_{\tau_1 \to \tau_2}\, [\![e_1]\!]\, [\![e_2]\!]$. Ainsi, la défonctionalisation dans un cadre simplement typé non seulement *préserve*, mais *est dirigée par* le typage. Notons que $[\![\cdot]\!]$ *ne commute pas* avec la substitution de variables de types par des types. En effet, tout type flèche distinct dans le programme source est codé par un type de données algébrique distinct dans le programme cible. Du coup, il n'existe aucune façon naturelle de traduire un type flèche qui contiendrait des variables de types. Ces remarques expliquent pourquoi cette approche échoue en présence de polymorphisme.

Une solution à base de types de données algébriques gardés Nadji Gauthier et moi-même avons proposé une nouvelle solution à ce problème [154]. On conserve une unique fonction $apply$, dont le type est $\forall\alpha_1\alpha_2.[\![\alpha_1 \to \alpha_2]\!] \to \alpha_1 \to \alpha_2$, comme initialement suggéré ci-dessus. On exige également que le codage des types commute avec les substitutions de types, ce qui implique que $[\![\tau_1 \to \tau_2]\!]$ doit s'écrire $Arrow\ [\![\tau_1]\!]\ [\![\tau_2]\!]$, pour un certain constructeur de types de données algébriques distingué binaire $Arrow$. Il reste à trouver une extension du système F dans laquelle la définition de $apply$ soit bien typée, c'est-à-dire où toutes les clauses produisent effectivement une valeur de type α_2, sous l'hypothèse que arg est de type α_1. La remarque clef est que, pour permettre cela, on doit acquérir des hypothèses supplémentaires à propos de α_1 et α_2. Par exemple, dans le cas de la branche *succ*, on pourrait raisonner comme suit. Si cette branche est empruntée, alors f est *succ*, donc *succ* admet le type $Arrow\,\alpha_1\,\alpha_2$. Cependant, nous savons que la λ-abstraction associée avec l'étiquette *succ*, à savoir $\lambda x.x + 1$, est de type $int \to int$, donc il est naturel d'attribuer le type $Arrow\ int\ int$ au constructeur de données *succ*. En combinant ces deux faits, nous constatons que, si la branche est empruntée, alors on doit avoir $Arrow\,\alpha_1\,\alpha_2 = Arrow\ int\ int$, c'est-à-dire $\alpha_1 = int$ et $\alpha_2 = int$. *Sous ces hypothèses de typage supplémentaires*, il est possible de prouver que, si arg est de type α_1, alors $arg + 1$ admet le type α_2. En analysant chaque clause de façon analogue, on conclut que $apply$ est bien typée.

Quels sont les ingrédients nécessaires pour cette solution ? D'abord, les constructeurs de données *succ* et *not*, qui sont associés au type de données algébrique $Arrow$, doivent recevoir les schémas de types $Arrow\ int\ int$ et $Arrow\ bool\ bool$, respectivement. Notons que, si $Arrow$ était un type de données algébrique ordinaire, alors ces constructeurs d'arité nulle recevraient nécessairement le schéma de types $\forall\alpha_1\alpha_2.Arrow\,\alpha_1\,\alpha_2$. Ensuite, lorsqu'on examine l'étiquette d'une valeur de type $Arrow\,\alpha_1\,\alpha_2$, il est nécessaire que la branche correspondant à *succ* (resp. *not*) soit typée sous l'hypothèse supplémentaire $Arrow\,\alpha_1\,\alpha_2 = Arrow\ int\ int$ (resp. $Arrow\,\alpha_1\,\alpha_2 = Arrow\ bool\ bool$). C'est précisément ce que permettent les règles de typage associées aux types de données algébriques gardés.

Moralité La défonctionalisation peut être vue non seulement comme une technique de compilation, mais aussi comme un outil qu'on peut employer manuellement pour raisonner à propos d'un programme et le transformer. Danvy et Nielsen [178] ont noté qu'elle constitue un inverse du codage de Church, ce qui signifie qu'elle permet de raisonner en termes de structures de données, au lieu de fonctions d'ordre supérieur. On peut illustrer ce point à l'aide de la fonction *sprintf*, connue pour être difficile à exprimer en ML, parce que la valeur de son premier argument dicte le nombre et les types des arguments suivants. Danvy [179] a suggéré une façon subtile d'exprimer *sprintf* en ML en codant les spécificateurs de format sous forme de fonctions de première classe. Plus récemment, Xi *et al.* [156] ont montré que *sprintf* peut être exprimée dans un style direct, où les spécificateurs sont des structures de données, à l'aide de types de données algébriques gardés. Notre étude permet de réaliser que la seconde approche n'est qu'une version défonctionalisée de la première. Ainsi, le fait que la défonctionalisation préserve le typage signifie que certains programmes existants, qui utilisent des ruses à base de continuations pour contourner les limitations du système de types de ML, peuvent être réécrits de façon plus naturelle en tant que programmes du premier ordre, manipulant des structures de données définies par des types de données algébriques gardés. Cette remarque théorique me semble un bon argument pour justifier l'ajout de types de données algébriques gardés aux langages de programmation de la famille ML, comme Objective Caml.

6.3 Inférence de types

En collaboration avec Vincent Simonet [155], j'ai donc étudié l'introduction de types de données algébriques gardés dans ML, et, plus généralement, dans le cadre général du typage à base de contraintes. Ainsi, en ajoutant à HM(X) la récursivité polymorphe, les types de données algébriques gardés et le filtrage, nous avons obtenu un système plus expressif, baptisé HMG(X).

La définition de HMG(X) généralise les travaux de Xi [180–182] et de Zenger [183, 184]. Tous deux s'intéressent à des systèmes de types qui *raffinent* ML, c'est-à-dire qui permettent d'exprimer des propriétés plus fines à propos d'un programme, mais ne permettent pas de considérer de nouveaux programmes comme bien typés. Il généralise également le travail de Xi *et al.* [156], qui introduit les

types de données algébriques gardés. Une généralisation similaire à la nôtre, mais où l'accent n'est pas mis sur l'inférence de types, a été effectuée indépendamment par Xi [185]. Techniquement, notre travail est très proche de celui de Zenger [184], mais ajoute le sous-typage, le filtrage à l'aide de motifs de profondeur arbitraire, et est à mon avis présenté de façon plus concise et élégante, surtout en ce qui concerne la génération de contraintes.

Je ne m'étendrai pas ici sur la définition de HMG(X), qui me semble relativement naturelle. Je présente simplement la règle de typage qui concerne les clauses. (Une clause $p.e$, où p est un motif et e une expression, permet de définir des fonctions par cas.) Elle s'écrit comme suit :

$$\frac{C \vdash p : \tau' \rightsquigarrow \exists\bar{\beta}[D]\Gamma' \qquad C \wedge D, \Gamma\Gamma' \vdash e : \tau \qquad \bar{\beta} \# \operatorname{ftv}(C, \Gamma, \tau)}{C, \Gamma \vdash p.e : \tau' \to \tau}$$

On peut lire cette règle de la façon suivante : « Supposons que, sous l'hypothèse C, il soit permis de soumettre une valeur de type τ' au motif p, et que, en cas de succès du filtrage, on puisse en déduire l'existence de types inconnus $\bar{\beta}$, satisfaisant D, et tels que les variables nouvellement liées par le motif p admettent les types décrits par Γ'. Supposons de plus que, sous les hypothèses C et D, et dans l'environnement obtenu en concaténant Γ et Γ', l'expression e admet le type τ. Alors, sous la seule hypothèse C et dans l'environnement Γ, la clause $p.e$ admet le type $\tau' \to \tau$. »

Lorsque le motif p ne fait intervenir que des types de données algébriques ordinaires, $\bar{\beta}$ est vide et D est ***true***. On retrouve alors la règle qui gouverne le filtrage en ML. Lorsque le motif fait intervenir un type existentiel au sens de Läufer et Odersky [157], $\bar{\beta}$ peut être non vide. On retrouve alors la règle qui gouverne l'élimination des types existentiels, connue sous le nom de *open*. La clause de fraîcheur concernant les variables de types $\bar{\beta}$ garantit que ces variables sont considérées comme abstraites. Elle exige en fait que la fonction « $\lambda\Gamma'.e$ » soit polymorphe vis-à-vis de ces variables de types. Enfin, dans le cas général, la contrainte D peut être non triviale. On note alors que l'expression e est typée sous une hypothèse $C \wedge D$ plus riche que l'hypothèse C disponible à l'extérieur de la clause. C'est par ce moyen qu'un test dynamique, à savoir le filtrage, fournit une information statique supplémentaire.

Le système HMG(X) autorise la récursivité polymorphe, pour laquelle le problème de l'inférence de types est indécidable en l'absence d'annotations explicites [186]. On exige donc que toutes les définitions récursives soient annotées par un schéma de types. Une fois cette restriction raisonnable adoptée, il est possible de démontrer que l'inférence de types pour HMG(X) se ramène à la résolution de contraintes. Dans le cas d'une clause $p.e$, la contrainte engendrée est de la forme $\forall\bar{\beta}.D \Rightarrow \ldots$, où $\bar{\beta}$ et D sont essentiellement ceux de la règle de typage ci-dessus. La quantification universelle reflète le fait que les variables de types $\bar{\beta}$ représentent des types abstraits, et apparaît déjà lorsque l'on effectue l'inférence de types pour le système de types de Läufer et Odersky [157]. L'implication est propre au filtrage en présence de types de données algébriques gardés, et reflète le fait qu'une hypothèse supplémentaire, représentée par la contrainte D, est localement disponible.

Le problème de l'inférence de types se ramène donc à la résolution de contraintes dans une théorie du premier ordre. Je veux dire par là que, outre les prédicats élémentaires que nous avons pu employer (égalité, sous-typage, et éventuellement autres), l'ensemble des connecteurs de la logique du premier ordre est mis à contribution, puisque l'implication contient la négation. La résolution de contraintes est alors décidable au moins dans certains cas, comme ceux de l'égalité [187, 188] et du sous-typage structurel [189, 190]. Néanmoins, la complexité du problème est, dans tous les cas, non élémentaire [88]. Même si des algorithmes de résolution concrets ont été proposés [191–195], il serait déraisonnable de vouloir emprunter cette route.

Par conséquent, il est nécessaire de reconsidérer le problème et de faire en sorte que les contraintes engendrées appartiennent à un fragment suffisamment restreint pour admettre un algorithme de résolution efficace. C'est ce que Vincent Simonet et moi-même [155] effectuons donc dans un second temps. Notre proposition est de se placer dans un fragment où les contraintes d'implication $\forall\bar{\beta}.D \Rightarrow \ldots$ satisfont la condition $\operatorname{ftv}(D) \subseteq \bar{\beta}$. Cette condition est forte : il est immédiatement apparent que l'implication générale n'est plus exprimable.

Dans le cas où le seul prédicat élémentaire est l'égalité, elle est suffisante pour garantir un algorithme de résolution efficace. En effet, la contrainte $\exists\bar{\beta}.D$ étant close, on peut décider, par unification, si elle satisfiable ou non. Si elle ne l'est pas, alors $\forall\bar{\beta}.D \Rightarrow \ldots$ est équivalente à ***true***. Si elle l'est, alors la simplicité de la structure de sa forme résolue permet de réécrire $\forall\bar{\beta}.D \Rightarrow \ldots$ en une contrainte de

la forme $\forall\bar{\beta}'\ldots.$, où l'implication a disparu. Le fait qu'il soit possible d'éliminer l'implication a une conséquence importante : les formes résolues, pour notre langage de contraintes, sont les mêmes que celles d'un algorithme d'unification ordinaire. Comme je l'ai expliqué plus tôt, cette propriété permet aux contraintes de rester invisibles aux yeux du programmeur. L'ajout de types de données algébriques gardés à ML respecte donc l'esprit du langage.

Le cas où le seul prédicat élémentaire est une relation de sous-typage structurel a été étudié par Simonet [196]. Dans ce cas également, une restriction de la forme des implications semble nécessaire pour parvenir à un algorithme de résolution efficace.

Pour que les contraintes engendrées puissent appartenir au langage ainsi restreint, il est nécessaire d'exiger quelques annotations supplémentaires de la part du programmeur. En bref, il est suffisant d'exiger que les constructions ***case*** et μ soient combinées et annotées par un schéma de types *clos*. La définition par cas de fonctions récursives devient donc une construction primitive du langage. La même solution est entrevue par d'autres auteurs, parmi lesquels Xi *et al.* [156], Cheney et Hinze [160], et Sheard [197].

Chapitre 7

Et ensuite ?

LA THÈSE que je défends ici est la suivante : les contraintes fournissent un formalisme *léger*, *déclaratif*, et *adapté au calcul*, pour la spécification des systèmes de types. En effet, on peut voir un algorithme de vérification ou d'inférence de types comme une simple fonction qui à un programme, ou fragment de programme, associe une contrainte (voir, entre autres, les figures 2.2, 2.3 et 2.6). Les typages admissibles par le programme sont alors en bijection avec les solutions de cette contrainte.

Pour arguer de la *légèreté* du formalisme, je m'appuie sur une observation empirique : même lorsque le langage de programmation étudié est riche, le langage de contraintes reste relativement restreint – il s'agit en général d'un fragment de la logique du premier ordre fondé sur un très petit nombre de prédicats de base.

L'aspect *déclaratif* du formalisme provient du fait que les contraintes sont dotées d'une interprétation logique dans un modèle, laquelle définit les notions de solution, de satisfiabilité, et d'équivalence de contraintes. On peut ainsi raisonner sur les contraintes, donc sur les programmes, de façon purement logique. Le difficile problème de la mise au point et de la description formelle d'un algorithme de résolution de contraintes efficace devient alors entièrement indépendant.

Enfin, les contraintes permettent des *calculs* que n'autorisent pas les présentations classiques à base de règles de déduction. Par exemple, pour savoir si un programme est bien typé, il suffit de déterminer si la contrainte qui lui correspond est satisfiable, ce que permet en général un calcul spécifié sous forme d'un système de réécriture de contraintes. Les contraintes servent donc non seulement à la spécification du problème, mais également à sa résolution, étape par étape. Par ailleurs, pour savoir si deux programmes admettent les mêmes typages, il suffit de déterminer si les contraintes qui leur correspondent sont équivalentes, ce qu'on peut réaliser par le calcul, si le problème de l'équivalence de contraintes est décidable. Une idée voisine peut être employée pour mécaniser partiellement les preuves d'autoréduction (§2.4.4).

Cette thèse n'est certes pas nouvelle, puisque la discipline du typage à base de contraintes est apparue dans les années 1980 et s'est développée au cours des années 1990. Tout au plus l'ai-je illustrée ici au moyen de quelques exemples concrets.

Quel est l'avenir de cette discipline ? La floraison de systèmes de types à base de contraintes plus ou moins exotiques, souvent dédiées à une application particulière, à laquelle on a assisté continuera probablement ; mais là n'est pas, à mon avis, le plus intéressant.

Dans ce qui suit, je suggère deux axes selon lesquels je souhaiterais orienter mes recherches futures. Il s'agit là d'idées assez générales, donc nécessairement vagues. Le premier axe consiste à revisiter en termes de contraintes certains domaines où celles-ci n'ont pas ou peu été employées. Le second consiste à concevoir des systèmes de types, à base de contraintes ou non, qui soient assez séduisants pour être adoptés par des langages de programmations réalistes, et ne pas rester à l'état de propositions théoriques.

Mieux comprendre certaines théories grâce aux contraintes Il reste aujourd'hui quelques familles de systèmes de types auxquelles les techniques à base de contraintes n'ont pas été appliquées. Or, expliquer certains de ces systèmes en termes de contraintes pourrait permettre d'abord de mieux les

comprendre, ensuite d'en découvrir d'éventuelles généralisations.

Je pense, par exemple, à l'extension du système de types de Damas et Milner connue sous le nom de « polymorphisme de rang arbitraire, » due à Odersky et Läufer [198] et mise en œuvre dans le compilateur GHC par Peyton Jones et Shields [199]. L'algorithme d'inférence de types proposé par Odersky et Läufer est à base de substitutions, donc relativement peu lisible. Celui décrit par Peyton Jones et Shields est présenté sous forme de code Haskell. Or, je prétends que l'inférence de types pour ce système peut être réduite à la composition d'une phase de propagation des annotations de types et d'une phase de génération de contraintes d'unification du premier ordre sous préfixe mixte. Cette présentation me semble faciliter la compréhension du système. De plus, elle permet de découvrir que certains des choix présentés comme décisifs par Peyton Jones et Shields n'ont en réalité aucun impact profond. Ce (modeste) résultat, non encore publié, a été obtenu en collaboration avec Didier Rémy.

Je pense par ailleurs à des systèmes de types plus riches, comme le système F ou certaines de ses restrictions dotées d'un problème d'inférence de types décidable. Par exemple, le système ML^{F} [200], dont l'expressivité semble très prometteuse, n'est pas présenté en termes de contraintes. Or, réduire l'inférence de types pour ML^{F} vers un problème de résolution de contraintes pourrait permettre d'une part de mieux comparer son expressivité à celle de ML, en évaluant la puissance du langage de contraintes nécessaire à une telle réduction ; et, d'autre part, de mieux comprendre quelles extensions on peut lui apporter : par exemple, l'ajout de types récursifs ou de types algébriques gardés serait-il possible ? En ce qui concerne le système F lui-même, l'inférence de types a été peu étudiée depuis que Wells [201] en a établi l'indécidabilité. Seul Raffalli [202, 203] semble avoir suivi une approche à base de contraintes. Or, il pourrait être intéressant de mieux comprendre ce problème, afin peut-être d'en isoler d'autres restrictions décidables. Par ailleurs, une réduction de l'inférence de types pour le système F vers un problème (indécidable) de résolution de contraintes pourrait faciliter les preuves de *non*-typabilité dans le système F, preuves dont la difficulté est notoire.

Enfin, les systèmes d'inférence de types locale [204, 205] n'ont jamais été expliqués en termes de contraintes. Quel serait un langage de contraintes approprié, et de quelle manière la notion aujourd'hui informelle de localité se manifesterait-elle ? La question est relativement théorique, mais néanmoins intrigante.

Faire progresser les langages de programmation On peut distinguer, parmi les systèmes de types, ceux conçus pour une analyse entièrement automatique, d'une part, et ceux destinés à être explicitement employés par le programmeur, d'autre part. On dispose, lors de la conception des premiers, d'une certaine marge de manœuvre, puisque les seuls impératifs sont une précision et une efficacité raisonnables. Or, en ce qui concerne les seconds, celle-ci est significativement réduite. En effet, l'enjeu est beaucoup plus important : le système de types d'un langage de programmation définit un univers conceptuel, une structure mentale, que l'on impose de fait au programmeur. Il est alors nécessaire de conjuguer non seulement expressivité et efficacité, mais aussi et surtout une grande simplicité. Force est malheureusement de constater que rares sont les systèmes de types à base de contraintes pouvant prétendre à ces trois qualités. Il y a donc là un défi à relever.

Le système de types $\mathrm{HMG}(=)$, c'est-à-dire le système obtenu en ajoutant à ML récursivité polymorphe et types de données algébriques gardés, est peut-être de ceux qui conjuguent ces qualités. Parce que les contraintes employées admettent des unificateurs principaux, il est possible de ne présenter au programmeur que des schémas de types non contraints, comme en ML. Par ailleurs, les types attribués aux programmes ML existants restent inchangés : la complexité supplémentaire ne se manifeste que lorsqu'on l'exploite effectivement. Enfin, je crois que le gain d'expressivité offert par cette extension est important. J'ai démontré qu'elle permet l'écriture de programmes en style « défonctionalisé, » ce qui représente toute une classe d'applications (§6.2). J'ai par ailleurs en tête quelques applications originales des types de données algébriques gardés, que j'espère bientôt publier.

D'autres auteurs sont aujourd'hui persuadés de l'intérêt de ces « types inductifs. » Xi et ses coauteurs en ont publié des applications dans divers domaines [206–208]. Cheney et Hinze [160] ont milité pour leur introduction dans Haskell, laquelle est actuellement étudiée par Peyton Jones, Washburn et Weirich [209]. L'enthousiaste Sheard [197] note qu'ils donnent au langage ML une partie du pouvoir d'expression d'un démonstrateur automatique tel que Coq, et prédit que les langages de programmation du futur en seront munis, ce qui réduira le fossé entre écriture et preuve de programmes.

Au-delà de l'effet de mode, je crois cette idée promise à un avenir durable. Sous ma direction, Yann Régis-Gianas étudie actuellement son implémentation, d'abord dans un prototype, puis au sein du compilateur Objective Caml, et il est probable qu'elle formera l'un des attraits des versions futures du langage.

Mon jugement est plus réservé en ce qui concerne les instances de $\mathrm{HMG}(X)$ dotées de langages de contraintes plus complexes. Par exemple, l'emploi de l'arithmétique de Presburger, soutenu par Xi [180, 210] et par Zenger [184], peut sembler tentant. Mais pourquoi privilégier ce fragment de l'arithmétique plutôt qu'une autre théorie logique ? Pour satisfaire les besoins de chacun, ne risque-t-on pas d'être amené à combiner diverses théories et d'obtenir ainsi un outil disparate ? Les formes résolues, qui sont présentées à l'utilisateur au sein des schémas de types inférés, ne risquent-elles pas d'être très complexes ? Si la combinaison de théories logiques rencontre un succès pratique certain dans le domaine de la démonstration entièrement automatisée, je ne suis pas convaincu qu'il soit bon d'introduire une telle complexité dans le système de types d'un langage de programmation généraliste. Se limiter aux contraintes d'égalité entre termes signifie certes renoncer à une certaine expressivité supplémentaire, mais semble relativement naturel, dans la mesure où ces contraintes sont déjà présentes en ML.

Peut-être, pour aller au-delà de la théorie de l'égalité, sans pour autant privilégier telle ou telle théorie particulières, faudrait-il autoriser l'utilisateur à définir lui-même le langage de contraintes qu'il souhaite exploiter, ainsi que la procédure de résolution associée. Un pas dans cette direction semble avoir été effectué par le langage expérimental Chameleon [211, 212]. La procédure de résolution des contraintes peut y être partiellement spécifiée par le programmeur, sous forme d'un système de réécriture confluent, exprimé dans le formalisme des *constraint handling rules* (CHR). Toutefois, pour garantir la sûreté du typage, une partie des règles de réécriture, suffisante pour définir l'interprétation logique des contraintes, est engendrée automatiquement par le système.

De façon générale, augmenter l'expressivité du système de types associé à un langage de programmation généraliste, sans en sacrifier la simplicité, reste un défi qu'on ne voit relever que fort rarement. Néanmoins, ces dernières années ont vu des progrès importants dans le domaine du typage et de notre compréhension de la structure des programmes – par exemple, aurait-on imaginé, il y a dix ans, pouvoir analyser, par le typage, le code d'un ramasseur de miettes, exprimé en langage machine ? Je suis heureux d'avoir pu observer cette évolution, ainsi qu'y participer modestement, et suis impatient d'apprendre ce que le futur nous réserve.

Chapitre 8

Bibliographie

[1] François Pottier et Didier Rémy. The essence of ML type inference. *In* Benjamin C. Pierce, éditeur. *Advanced Topics in Types and Programming Languages*, chapitre 10, pages 389–489. MIT Press, 2005.

[2] François Pottier. A semi-syntactic soundness proof for HM(X). Research Report 4150, INRIA, mars 2001.

[3] Sylvain Conchon et François Pottier. JOIN(X) : Constraint-based type inference for the join-calculus. *In European Symposium on Programming (ESOP)*, volume 2028 de *Lecture Notes in Computer Science*, pages 221–236. Springer Verlag, avril 2001.

[4] Christian Skalka et François Pottier. Syntactic type soundness for HM(X). *In Workshop on Types in Programming (TIP)*, volume 75 de *Electronic Notes in Theoretical Computer Science*, juillet 2002.

[5] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[6] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, décembre 1978.

[7] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1–2):11–49, avril 2000.

[8] François Pottier. *Synthèse de types en présence de sous-typage : de la théorie à la pratique*. Thèse de doctorat, Université Paris 7, juillet 1998.

[9] J. Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[10] John C. Mitchell. Coercion and type inference. *In ACM Symposium on Principles of Programming Languages (POPL)*, pages 175–185, janvier 1984.

[11] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux et Gilles Kahn. A simple applicative language : Mini-ML. *In ACM Symposium on Lisp and Functional Programming (LFP)*, pages 13–27, 1986.

[12] Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, 1987.

[13] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticæ*, 10:115–122, 1987.

[14] Jean-Pierre Jouannaud et Claude Kirchner. Solving equations in abstract algebras : a rule-based survey of unification. Rapport technique 561, Université Paris-Sud, avril 1990.

[15] Murdoch J. Gabbay et Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13(3–5):341–363, juillet 2002.

[16] Trevor Jim. What are principal typings and what are they good for ? Rapport technique MIT/LCS TM-532, Massachusetts Institute of Technology, août 1995.

[17] J. B. Wells. The essence of principal typings. *In International Colloquium on Automata, Languages and Programming*, volume 2380 de *Lecture Notes in Computer Science*, pages 913–925. Springer Verlag, 2002.

[18] Gérard Huet. *Résolution d'équations dans des langages d'ordre* $1, 2, \ldots, \omega$. Thèse de doctorat, Université Paris 7, septembre 1976.

[19] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, avril 1975.

[20] M. S. Paterson et M. N. Wegman. Linear unification. *In Annual ACM Symposium on Theory of Computing*, pages 181–186, 1976.

[21] Mario Coppo et Mariangiola Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame J. Formal Logic*, 21(4):685–693, 1980.

[22] Garrel Pottinger. A type assignment for the strongly normalizable λ-terms. *In* J. Roger Hindley et Jonathan P. Seldin, éditeurs. *To H. B. Curry : Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 561–577. Academic Press, 1980.

[23] Luis Damas et Robin Milner. Principal type-schemes for functional programs. *In ACM Symposium on Principles of Programming Languages (POPL)*, pages 207–212, 1982.

[24] Assaf J. Kfoury, Jerzy Tiuryn et Pawel Urzyczyn. ML typability is DEXPTIME-complete. *In Colloquium on Trees in Algebra and Programming*, volume 431 de *Lecture Notes in Computer Science*, pages 206–220. Springer Verlag, 1990.

[25] Harry G. Mairson, Paris C. Kanellakis et John C. Mitchell. Unification and ML type reconstruction. *In* J.-L. Lassez et G. Plotkin, éditeurs. *Computational Logic : Essays in Honor of Alan Robinson*, pages 444–478. MIT Press, 1991.

[26] Mark P. Jones. Typing Haskell in Haskell. *In Haskell workshop*, octobre 1999.

[27] Xavier Leroy et François Pottier. Notes du cours de DEA « typage et programmation », décembre 2002.

[28] Simon Peyton Jones, éditeur. *Haskell 98 Language and Libraries : The Revised Report*. Cambridge University Press, avril 2003.

[29] Luis Damas. *Type Assignment in Programming Languages*. Thèse de doctorat, University of Edinburgh, 1985.

[30] Xavier Leroy. *Typage polymorphe d'un langage algorithmique*. Thèse de doctorat, Université Paris 7, juin 1992.

[31] Catherine Dubois et Valérie Ménissier-Morain. Typage de ML : Spécification et preuve en Coq. *In Actes du GDR Programmation*, novembre 1997.

[32] Catherine Dubois et Valérie Ménissier-Morain. Certification of a type inference tool for ML : Damas-Milner within Coq. *Journal of Automated Reasoning*, 23(3–4):319–346, novembre 1999.

[33] Wolfgang Naraschewski et Tobias Nipkow. Type inference verified : Algorithm W in Isabelle/HOL. *Journal of Automated Reasoning*, 23:299–318, 1999.

[34] Oukseh Lee et Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):707–723, 1998.

[35] Martin Odersky, Martin Sulzmann et Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[36] Martin Müller. A constraint-based recast of ML-polymorphism. *In International Workshop on Unification*, juin 1994. Technical Report 94-R-243, CRIN, Nancy, France.

[37] Jörgen Gustavsson et Josef Svenningsson. Constraint abstractions. *In Symposium on Programs as Data Objects*, volume 2053 de *Lecture Notes in Computer Science*. Springer Verlag, mai 2001.

[38] David McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, juillet 2002.

[39] Didier Rémy. Extending ML type system with a sorted equational theory. Rapport technique 1766, INRIA, 1992.

[40] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[41] Valery Trifonov et Scott Smith. Subtyping constrained types. *In Static Analysis Symposium (SAS)*, volume 1145 de *Lecture Notes in Computer Science*, pages 349–365. Springer Verlag, septembre 1996.

[42] François Pottier. Simplifying subtyping constraints : a theory. *Information and Computation*, 170(2):153–183, novembre 2001.

[43] Olaf Chitil. Compositional explanation of types and algorithmic debugging of type errors. *In ACM International Conference on Functional Programming (ICFP)*, pages 193–204, septembre 2001.

[44] Philip Wadler et Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. *In ACM Symposium on Principles of Programming Languages (POPL)*, pages 60–76, janvier 1989.

[45] Cordelia Hall, Kevin Hammond, Simon Peyton Jones et Philip Wadler. Type classes in Haskell. *In* Donald Sannella, éditeur. *European Symposium on Programming (ESOP)*, volume 788 de *Lecture Notes in Computer Science*, pages 241–256. Springer Verlag, avril 1994.

[46] Karl-Filip Faxén. A static semantics for Haskell. *Journal of Functional Programming*, 12(4–5):295–357, juillet 2002.

[47] You-Chin Fuh et Prateek Mishra. Type inference with subtypes. *In European Symposium on Programming (ESOP)*, volume 300 de *Lecture Notes in Computer Science*, pages 94–114. Springer Verlag, 1988.

[48] Pavel Curtis. *Constrained Quantification in Polymorphic Type Analysis*. Thèse de doctorat, Cornell University, février 1990.

[49] Alexander S. Aiken et Edward L. Wimmers. Type inclusion constraints and type inference. *In Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 31–41. ACM Press, 1993.

[50] Mark P. Jones. A theory of qualified types. *In European Symposium on Programming (ESOP)*, volume 582 de *Lecture Notes in Computer Science*. Springer Verlag, février 1992.

[51] Mark P. Jones. *Qualified Types : Theory and Practice*. Cambridge University Press, novembre 1994.

[52] Geoffrey S. Smith. Polymorphic type inference with overloading and subtyping. *In* Marie-Claude Gaudel et Jean-Pierre Jouannaud, éditeurs. *International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, volume 668 de *Lecture Notes in Computer Science*, pages 671–685. Springer Verlag, avril 1993.

[53] Geoffrey S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23(2–3):197–226, décembre 1994.

[54] Jonathan Eifrig, Scott Smith et Valery Trifonov. Type inference for recursively constrained types and its application to OOP. *In Mathematical Foundations of Programming Semantics*, volume 1 de *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1995.

[55] Martin Odersky, Philip Wadler et Martin Wehr. A second look at overloading. *In Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 135–146, juin 1995.

[56] François Bourdoncle et Stephan Merz. Type checking higher-order polymorphic multi-methods. *In ACM Symposium on Principles of Programming Languages (POPL)*, pages 302–315, janvier 1997.

[57] Manuel Fähndrich. BANE *: A Library for Scalable Constraint-Based Program Analysis*. Thèse de doctorat, University of California at Berkeley, 1999.

[58] Daniel Bonniot. Type-checking multi-methods in ML (a modular approach). *In Workshop on Foundations of Object-Oriented Languages (FOOL)*, janvier 2002.

[59] Alexandre Frey. *Approche algébrique du typage d'un langage à la ML avec objets, sous-typage et multi-méthodes.* Thèse de doctorat, École des Mines de Paris, juin 2004.

[60] Alberto Martelli et Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, avril 1982.

[61] Martín Abadi et Marcelo P. Fiore. Syntactic considerations on recursive types. *In IEEE Symposium on Logic in Computer Science (LICS)*, pages 242–252, juillet 1996.

[62] Jean Goubault. Inférence d'unités physiques en ML. *In* Pierre Cointe, Christian Queinnec et Bernard Serpette, éditeurs. *Journées Françaises des Langages Applicatifs*, pages 3–20, 1994.

[63] Andrew Kennedy. Dimension types. *In European Symposium on Programming (ESOP)*, volume 788 de *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[64] Andrew Kennedy. Type inference and equational theories. Rapport technique LIX/RR/96/09, École Polytechnique, septembre 1996.

[65] Murdoch J. Gabbay. *A Theory of Inductive Definitions with α-Equivalence*. Thèse de doctorat, Cambridge University, 2001.

[66] Murdoch J. Gabbay. A general mathematics of names in syntax. Submitted for publication, mars 2004.

[67] Cédric Fournet et Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. *In ACM Symposium on Principles of Programming Languages (POPL)*, pages 372–385, 1996.

[68] Cédric Fournet, Luc Maranget, Cosimo Laneve et Didier Rémy. Implicit typing à la ML for the join-calculus. *In International Conference on Concurrency Theory (CONCUR)*, volume 1243 de *Lecture Notes in Computer Science*, pages 196–212. Springer Verlag, 1997.

[69] Martin Sulzmann. *A general framework for Hindley/Milner type systems with constraints*. Thèse de doctorat, Yale University, Department of Computer Science, mai 2000.

[70] David B. MacQueen, Gordon D. Plotkin et Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1–2):95–130, octobre–novembre 1986.

[71] Martín Abadi, Benjamin Pierce et Gordon Plotkin. Faithful ideal models for recursive polymorphic types. *International Journal of Foundations of Computer Science*, 2(1):1–21, mars 1991.

[72] Andrew K. Wright et Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, novembre 1994.

[73] François Pottier. Wallace : an efficient implementation of type inference with subtyping, février 2000.

[74] François Pottier. A 3-part type inference engine. *In European Symposium on Programming (ESOP)*, volume 1782 de *Lecture Notes in Computer Science*, pages 320–335. Springer Verlag, mars 2000.

[75] François Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4):312–347, novembre 2000.

[76] François Pottier. A constraint-based presentation and generalization of rows. *In IEEE Symposium on Logic in Computer Science (LICS)*, pages 331–340, juin 2003.

[77] James Gosling, Bill Joy, Guy Steele et Gilad Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.

[78] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, février 1988.

[79] Roberto M. Amadio et Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, septembre 1993.

[80] Dexter Kozen, Jens Palsberg et Michael I. Schwartzbach. Efficient recursive subtyping. *Mathematical Structures in Computer Science*, 5(1):113–125, 1995.

[81] Jens Palsberg, Mitchell Wand et Patrick M. O'Keefe. Type inference with non-structural subtyping. *Formal Aspects of Computing*, 9:49–67, 1997.

[82] Trevor Jim et Jens Palsberg. Type inference in systems of recursive types with subtyping. Manuscript, 1999.

[83] Zhendong Su, Alexander Aiken, Joachim Niehren, Tim Priesnitz et Ralf Treinen. The first-order theory of subtyping constraints. *In ACM Symposium on Principles of Programming Languages (POPL)*, pages 203–216, janvier 2002.

[84] Joachim Niehren et Tim Priesnitz. Non-structural subtype entailment in automata theory. *Information and Computation*, 186(2):319–354, 2003.

[85] John C. Reynolds. Automatic computation of data set definitions. *In Information Processing 68*, volume 1, pages 456–461. North Holland, 1969.

[86] Nevin Heintze. Set based analysis of ML programs. Rapport technique CMU-CS-93-193, Carnegie Mellon University, School of Computer Science, juillet 1993.

[87] Alexander S. Aiken, Edward L. Wimmers et T. K. Lakshman. Soft typing with conditional types. *In ACM Symposium on Principles of Programming Languages (POPL)*, pages 163–173, janvier 1994.

[88] Sergei G. Vorobyov. An improved lower bound for the elementary theories of trees. *In International Conference on Automated Deduction (CADE)*, volume 1104 de *Lecture Notes in Computer Science*, pages 275–287. Springer Verlag, 1996.

[89] Zhendong Su et Alexander Aiken. Entailment with conditional equality constraints. *In European Symposium on Programming (ESOP)*, volume 2028 de *Lecture Notes in Computer Science*, pages 170–189, avril 2001.

[90] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93(1):1–15, juillet 1991.

[91] Didier Rémy. Type inference for records in a natural extension of ML. *In* Carl A. Gunter et John C. Mitchell, éditeurs. *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1994.

[92] Didier Rémy. Projective ML. *In ACM Symposium on Lisp and Functional Programming (LFP)*, pages 66–75, 1992.

[93] Jens Palsberg et Tian Zhao. Efficient type inference for record concatenation and subtyping. *In IEEE Symposium on Logic in Computer Science (LICS)*, pages 125–136, juillet 2002.

[94] Jens Palsberg et Tian Zhao. Type inference for record concatenation and subtyping. *Information and Computation*, 189:54–86, 2004.

[95] Luca Cardelli et John Mitchell. Operations on records. *In* Carl A. Gunter et John C. Mitchell, éditeurs. *Theoretical Aspects of Object-Oriented Programming : Types, Semantics, and Language Design*. MIT Press, 1994.

[96] Didier Rémy. From classes to objects via subtyping. *In European Symposium on Programming (ESOP)*, volume 1381 de *Lecture Notes in Computer Science*, pages 200–220. Springer Verlag, mars 1998.

[97] Gilad Bracha et William Cook. Mixin-based inheritance. *In ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 303–311, 1990.

[98] Gilad Bracha et Gary Lindstrom. Modularity meets inheritance. Rapport technique UUCS-91-017, University of Utah, octobre 1991.

[99] Tom Hirschowitz et Xavier Leroy. Mixin modules in a call-by-value setting. *ACM Transactions on Programming Languages and Systems*, 2004. To appear.

[100] Robert Harper et Benjamin Pierce. A record calculus based on symmetric concatenation. *In ACM Symposium on Principles of Programming Languages (POPL)*, pages 131–142, janvier 1991.

[101] Hideki Tsuiki. On typed calculi with a merge operator. *In Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 880 de *Lecture Notes in Computer Science*, pages 101–112. Springer Verlag, 1994.

[102] Jan Zwanenburg. A type system for record concatenation and subtyping. Rapport technique, Eindhoven University of Technology, juillet 1997.

[103] Atsushi Ohori et Peter Buneman. Type inference in a database programming language. *In ACM Symposium on Lisp and Functional Programming (LFP)*, pages 174–183, 1988.

[104] Peter Buneman et Atsushi Ohori. Polymorphism and type inference in database programming. *ACM Transactions on Database Systems*, 21(1):30–76, 1996.

[105] Didier Rémy. A case study of typechecking with constrained types : Typing record concatenation. Workshop on Advances in Types for Computer Science, août 1995.

[106] Didier Rémy. Typing record concatenation for free. *In* Carl A. Gunter et John C. Mitchell, éditeurs. *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1994.

[107] Gert Smolka et Ralf Treinen. Records for logic programming. *Journal of Logic Programming*, 18(3):229–258, avril 1994.

[108] Martin Müller, Joachim Niehren et Ralf Treinen. The first-order theory of ordering constraints over feature trees. *Discrete Mathematics and Theoretical Computer Science*, 4(2):193–234, 2001.

[109] Martin Müller et Susumu Nishimura. Type inference for first-class messages with feature constraints. *In Asian Computer Science Conference (ASIAN)*, volume 1538 de *Lecture Notes in Computer Science*, pages 169–187. Springer Verlag, décembre 1998.

[110] Martin Müller et Susumu Nishimura. Type inference for first-class messages with feature constraints. *International Journal of Foundations of Computer Science*, 11(1):29–63, 2000.

[111] François Pottier et Sylvain Conchon. Information flow inference for free. *In ACM International Conference on Functional Programming (ICFP)*, pages 46–57, septembre 2000.

[112] François Pottier, Christian Skalka et Scott Smith. A systematic approach to static access control. *In European Symposium on Programming (ESOP)*, volume 2028 de *Lecture Notes in Computer Science*, pages 30–45. Springer Verlag, avril 2001.

[113] François Pottier, Christian Skalka et Scott Smith. A systematic approach to static access control. To appear in *ACM Transactions on Programming Languages and Systems*, octobre 2003.

[114] Mitchell Wand. Embedding type structure in semantics. *In ACM Symposium on Principles of Programming Languages (POPL)*, pages 1–6, janvier 1985.

[115] Philip L. Wadler. How to replace failure by a list of successes. *In Conference on Functional Programming Languages and Computer Architecture (FPCA)*, volume 201 de *Lecture Notes in Computer Science*, pages 113–128. Springer Verlag, septembre 1985.

[116] Eugenio Moggi. Computational λ-calculus and monads. *In IEEE Symposium on Logic in Computer Science (LICS)*, pages 14–23, juin 1989.

[117] Mike Spivey. A functional theory of exceptions. *Science of Computer Programming*, 14:25–42, 1990.

[118] Alexander S. Aiken et Manuel Fähndrich. Program analysis using mixed term and set constraints. *In Static Analysis Symposium (SAS)*, pages 114–126, septembre 1997.

[119] François Pessaux et Xavier Leroy. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, 2000.

[120] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16 (10):613–615, octobre 1973.

[121] Dorothy E. Denning et Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, juillet 1977.

[122] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[123] Dennis Volpano, Geoffrey Smith et Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[124] Dennis Volpano. Provably-secure programming languages for remote evaluation. *ACM SIGPLAN Notices*, 32(1):117–119, janvier 1997.

[125] Dennis Volpano et Geoffrey Smith. A type-based approach to program security. *Lecture Notes in Computer Science*, 1214:607–621, avril 1997.

[126] Dennis Volpano et Geoffrey Smith. Eliminating covert flows with minimum typings. *In IEEE Computer Security Foundations Workshop*, pages 156–168, juin 1997.

[127] Jens Palsberg et Peter Ørbæk. Trust in the λ-calculus. *In Static Analysis Symposium (SAS)*, volume 983 de *Lecture Notes in Computer Science*, pages 314–330, septembre 1995.

[128] Peter Ørbæk et Jens Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 7 (6):557–591, novembre 1997.

[129] Nevin Heintze et Jon G. Riecke. The SLam calculus : Programming with secrecy and integrity. *In ACM Symposium on Principles of Programming Languages (POPL)*, pages 365–377, janvier 1998.

[130] Martín Abadi, Anindya Banerjee, Nevin Heintze et Jon G. Riecke. A core calculus of dependency. *In ACM Symposium on Principles of Programming Languages (POPL)*, pages 147–160, janvier 1999.

[131] Martín Abadi, Butler Lampson et Jean-Jacques Lévy. Analysis and caching of dependencies. *In ACM International Conference on Functional Programming (ICFP)*, pages 83–91, mai 1996.

[132] John L. Ross et Mooly Sagiv. Building a bridge between pointer aliases and program dependences. *Nordic Journal of Computing*, 5(4):361–386, 1998.

[133] Li Gong, Gary Ellison et Mary Dageforde. *Inside Java 2 Platform Security, Second Edition*. Addison-Wesley, 2003.

[134] Michael A. Harrison, Walter L. Ruzzo et Jeffrey D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, août 1976.

[135] Li Gong, Marianne Mueller, Hemma Prafullchandra et Roland Schemers. Going beyond the sandbox : An overview of the new security architecture in the Java Development Kit 1.2. *In USENIX Symposium on Internet Technologies and Systems*, pages 103–112, décembre 1997.

[136] Li Gong et Roland Schemers. Implementing protection domains in the Java development kit 1.2. *In Internet Society Symposium on Network and Distributed System Security*, mars 1998.

[137] Dan S. Wallach et Edward Felten. Understanding Java stack inspection. *In IEEE Symposium on Security and Privacy (S&P)*, mai 1998.

[138] Dan S. Wallach. *A New Approach to Mobile Code Security*. Thèse de doctorat, Princeton University, janvier 1999.

[139] Cédric Fournet et Andrew D. Gordon. Stack inspection : Theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, mai 2003.

[140] Christian Skalka et Scott Smith. Static enforcement of security with types. *In ACM International Conference on Functional Programming (ICFP)*, pages 34–45, septembre 2000.

[141] François Pottier et Vincent Simonet. Information flow inference for ML. *In ACM Symposium on Principles of Programming Languages (POPL)*, pages 319–330, janvier 2002.

[142] François Pottier et Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, janvier 2003.

[143] François Pottier. A simple view of type-secure information flow in the π-calculus. *In IEEE Computer Security Foundations Workshop*, pages 320–330, juin 2002.

[144] Andrew C. Myers. *Mostly-Static Decentralized Information Flow Control*. Thèse de doctorat, Massachusetts Institute of Technology, janvier 1999. Technical Report MIT/LCS/TR-783.

[145] Steve Zdancewic et Andrew C. Myers. Secure information flow and CPS. *In European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science. Springer Verlag, avril 2001.

[146] Steve Zdancewic et Andrew C. Myers. Secure information flow via linear continuations. *Higher Order and Symbolic Computation*, 15(2–3):209–234, septembre 2002.

[147] Kohei Honda, Vasco Vasconcelos et Nobuko Yoshida. Secure information flow as typed process behaviour. *In European Symposium on Programming (ESOP)*, volume 1782 de *Lecture Notes in Computer Science*, pages 180–199. Springer Verlag, mars 2000.

[148] Kohei Honda et Nobuko Yoshida. A uniform type structure for secure information flow. *In ACM Symposium on Principles of Programming Languages (POPL)*, pages 81–92, janvier 2002.

[149] Vincent Simonet. Fine-grained information flow analysis for a λ-calculus with sum types. *In IEEE Computer Security Foundations Workshop*, pages 223–237, juin 2002.

[150] Lantian Zheng et Andrew C. Myers. Dynamic security labels and noninterference. Rapport technique 2004-1924, Cornell University, janvier 2004.

[151] Vincent Simonet. *Inférence de flots d'information pour ML : formalisation et implantation*. Thèse de doctorat, Université Paris 7, mars 2004.

[152] Vincent Simonet. The Flow Caml system : documentation and user's manual. Rapport technique 0282, INRIA, juillet 2003.

[153] Vincent Simonet. Type inference with structural subtyping : a faithful formalization of an efficient constraint solver. *In Asian Symposium on Programming Languages and Systems*, volume 2895 de *Lecture Notes in Computer Science*. Springer Verlag, novembre 2003.

[154] François Pottier et Nadji Gauthier. Polymorphic typed defunctionalization. *In ACM Symposium on Principles of Programming Languages (POPL)*, pages 89–98, janvier 2004.

[155] Vincent Simonet et François Pottier. Constraint-based type inference with guarded algebraic data types. Submitted, juin 2004.

[156] Hongwei Xi, Chiyan Chen et Gang Chen. Guarded recursive datatype constructors. *In ACM Symposium on Principles of Programming Languages (POPL)*, pages 224–235, janvier 2003.

[157] Konstantin Läufer et Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, septembre 1994.

[158] Christine Paulin-Mohring. Inductive definitions in the system Coq : rules and properties. Research Report RR1992-49, ENS Lyon, 1992.

[159] Benjamin Werner. *Une Théorie des Constructions Inductives*. Thèse de doctorat, Université Paris 7, 1994.

[160] James Cheney et Ralf Hinze. First-class phantom types. Rapport technique 1901, Cornell University, 2003.

[161] Karl Crary, Stephanie Weirich et Greg Morrisett. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming*, 12(6):567–600, novembre 2002.

[162] James Cheney et Ralf Hinze. A lightweight implementation of generics and dynamics. *In Haskell workshop*, 2002.

[163] Lennart Augustsson. Implementing Haskell overloading. *In Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 65–73, 1993.

[164] John Peterson et Mark P. Jones. Implementing type classes. *In ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 227–236, juin 1993.

[165] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, décembre 1998.

[166] John C. Reynolds. Definitional interpreters revisited. *Higher-Order and Symbolic Computation*, 11(4):355–361, décembre 1998.

[167] Lasse R. Nielsen. A denotational investigation of defunctionalization. Rapport technique RS-00-47, BRICS, décembre 2000.

[168] Anindya Banerjee, Nevin Heintze et Jon G. Riecke. Design and correctness of program transformations based on control-flow analysis. *In International Symposium on Theoretical Aspects of Computer Software (TACS)*, volume 2215 de *Lecture Notes in Computer Science*, pages 420–447. Springer Verlag, octobre 2001.

[169] Andrew Tolmach. Combining closure conversion with closure analysis using algebraic types. *In Workshop on Types in Compilation (TIC)*, juin 1997.

[170] Andrew Tolmach et Dino P. Oliva. From ML to Ada : Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, juillet 1998.

[171] Henry Cejtin, Suresh Jagannathan et Stephen Weeks. Flow-directed closure conversion for typed languages. *In European Symposium on Programming (ESOP)*, volume 1782 de *Lecture Notes in Computer Science*, pages 56–71. Springer Verlag, mars 2000.

[172] Jeffrey M. Bell, Françoise Bellegarde et James Hook. Type-driven defunctionalization. *In ACM International Conference on Functional Programming (ICFP)*, août 1997.

[173] Yasuhiko Minamide, Greg Morrisett et Robert Harper. Typed closure conversion. *In ACM Symposium on Principles of Programming Languages (POPL)*, pages 271–283, janvier 1996.

[174] Greg Morrisett et Robert Harper. Typed closure conversion for recursively-defined functions (extended abstract). *In International Workshop on Higher Order Operational Techniques in Semantics (HOOTS)*, volume 10 de *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1998.

[175] Greg Morrisett, David Walker, Karl Crary et Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, mai 1999.

[176] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'état, Université Paris 7, juin 1972.

[177] John C. Reynolds. Types, abstraction and parametric polymorphism. *In Information Processing 83*, pages 513–523. Elsevier Science, 1983.

[178] Olivier Danvy et Lasse R. Nielsen. Defunctionalization at work. *In ACM International Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 162–174, septembre 2001.

[179] Olivier Danvy. Functional unparsing. *Journal of Functional Programming*, 8(6):621–625, novembre 1998.

[180] Hongwei Xi. *Dependent Types in Practical Programming*. Thèse de doctorat, Carnegie Mellon University, décembre 1998.

[181] Hongwei Xi. Dead code elimination through dependent types. *In International Workshop on Practical Aspects of Declarative Languages (PADL)*, volume 1551 de *Lecture Notes in Computer Science*, pages 228–242. Springer Verlag, janvier 1999.

[182] Hongwei Xi. Dependently Typed Pattern Matching. *Journal of Universal Computer Science*, 9 (8):851–872, 2003.

[183] Christoph Zenger. Indexed types. *Theoretical Computer Science*, 187:147–165, 1997.

[184] Christoph Zenger. *Indizierte Typen*. Thèse de doctorat, Universität Karlsruhe, juillet 1998.

[185] Hongwei Xi. Applied type system. *In TYPES 2003*, volume 3085 de *Lecture Notes in Computer Science*, pages 394–408. Springer Verlag, février 2004.

[186] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, avril 1993.

[187] Michael J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. *In IEEE Symposium on Logic in Computer Science (LICS)*, pages 348–357, juillet 1988.

[188] Hubert Comon et Pierre Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7:371–425, 1989.

[189] Viktor Kuncak et Martin Rinard. Structural subtyping of non-recursive types is decidable. *In IEEE Symposium on Logic in Computer Science (LICS)*, juin 2003.

[190] Viktor Kuncak et Martin Rinard. On the theory of structural subtyping. Rapport technique 879, MIT Laboratory for Computer Science, janvier 2003.

[191] Alain Colmerauer. Equations and inequations on finite and infinite trees. *In International Conference on Fifth Generation Computer Systems (FGCS)*, pages 85–99, novembre 1984.

[192] Viswanath Ramachandran et Pascal Van Hentenryck. Incremental algorithms for constraint solving and entailment over rational trees. *In Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 205–217, 1993.

[193] Torbjörn Keisu. Finite and rational tree constraints. *Bulletin of the IGPL*, 2(2):167–204, 1994.

[194] Torbjörn Keisu. *Tree Constraints*. Thèse de doctorat, The Royal Institute of Technology (KTH), mai 1994.

[195] Thi Bich Hanh Dao. *Résolution de contraintes du premier ordre dans la théorie des arbres finis ou infinis*. Thèse de doctorat, Université de la Méditerranée, décembre 2000.

[196] Vincent Simonet. An extension of HM(X) with bounded existential and universal data-types. *In ACM International Conference on Functional Programming (ICFP)*, juin 2003.

[197] Tim Sheard. Languages of the future. *In ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 116–119, octobre 2004.

[198] Martin Odersky et Konstantin Läufer. Putting type annotations to work. *In ACM Symposium on Principles of Programming Languages (POPL)*, pages 54–67, janvier 1996.

[199] Simon Peyton Jones et Mark Shields. Practical type inference for arbitrary-rank types. Submitted, avril 2004.

[200] Didier Le Botlan et Didier Rémy. ML^{F} : Raising ML to the power of system F. *In ACM International Conference on Functional Programming (ICFP)*, pages 27–38, août 2003.

[201] J. B. Wells. Typability and type checking in system F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999.

[202] Christophe Raffalli. Type checking in system F^{η}. Prépublication 98-05a, LAMA, Université de Savoie, 1998.

[203] Christophe Raffalli. An optimized complete semi-algorithm for system F^{η}. Unpublished, 1999.

[204] Benjamin C. Pierce et David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, janvier 2000.

[205] Martin Odersky, Matthias Zenger et Christoph Zenger. Colored local type inference. *In ACM Symposium on Principles of Programming Languages (POPL)*, pages 41–53, 2001.

[206] Chiyan Chen et Hongwei Xi. Meta-programming through typeful code representation. *In ACM International Conference on Functional Programming (ICFP)*, pages 275–286, août 2003.

[207] Dengping Zhu et Hongwei Xi. A typeful and tagless representation for XML documents. *In Asian Symposium on Programming Languages and Systems*, volume 2895 de *Lecture Notes in Computer Science*, pages 89–104. Springer Verlag, novembre 2003.

[208] Chiyan Chen, Rui Shi et Hongwei Xi. A typeful approach to object-oriented programming with multiple inheritance. *In International Workshop on Practical Aspects of Declarative Languages (PADL)*, volume 3057 de *Lecture Notes in Computer Science*. Springer Verlag, juin 2004.

[209] Simon Peyton Jones, Geoffrey Washburn et Stephanie Weirich. Wobbly types : type inference for generalised algebraic data types. Draft, juillet 2004.

[210] Hongwei Xi. Dependent ML, 2001.

[211] Peter J. Stuckey et Martin Sulzmann. A theory of overloading. *In ACM International Conference on Functional Programming (ICFP)*, pages 167–178, 2002.

[212] Andreas Rossberg, Peter J. Stuckey, Martin Sulzmann et Jeremy Wazny. The Chameleon language.