T H È S E

présentée à

l'Université Paris 7 – Denis Diderot

pour obtenir le titre de

Docteur en Sciences

spécialité Informatique

Inférence de flots d'information pour ML : Formalisation et implantation

soutenue par

Vincent Simonet

le 16 mars 2004, devant le jury composé de :

Monsieur	Jean Goubault-Larrecq	*Président*
Messieurs	Guy Cousineau	
	François Pottier	*Directeurs de thèse*
Messieurs	Gérard Boudol	
	Thomas Jensen	*Rapporteurs*
Monsieur	Giuseppe Castagna	*Examinateur*

Résumé

Cette thèse décrit la conception d'un outil d'analyse de flots d'information pour un langage de la famille ML, de ses fondements théoriques aux aspects pratiques. La première partie du mémoire présente l'outil réalisé, Flow Caml, et illustre son fonctionnement sur des exemples concrets. La deuxième partie donne une formalisation du système de types utilisé, accompagné d'une preuve de sa correction. Il s'agit du premier système de types pour l'analyse de flots d'information dans un langage de programmation réaliste dont la correction ait été formellement prouvée. Enfin, la troisième partie est consacrée à l'étude d'un algorithme efficace pour l'inférence de types en présence de sous-typage structurel et de polymorphisme. Une instance de cet algorithme est utilisée pour la synthèse de types dans Flow Caml.

Abstract

This thesis describes the conception of an information flow analyser for a language of the ML family, from its theoretical foundation to the practical issues. The first part of the dissertation presents the tool that was implemented, Flow Caml, and illustrates its use on concrete examples. The second part gives a formalization of the type system featured by Flow Caml, together with a proof of its correctness. This is the first type system for information flow analysis in a realistic programming language that has been formally proved. Lastly, the third part is devoted to the formalization and the proof of an efficient algorithm for type inference in the presence of structural subtyping and polymorphism. An instance of this algorithm is used to synthesize types in Flow Caml.

Remerciements

Je dois beaucoup à François Pottier, qui a guidé mes recherches depuis mon stage de DEA. Toujours très disponible et attentif, il a accompagné chaque étape de ce travail, me permettant de tracer peu à peu mon propre chemin.

Gérard Boudol et Thomas Jensen m'ont témoigné de leur intérêt pour mon travail alors que je n'étais que jeune thésard, en m'invitant — à quelques mois d'intervalle — à présenter mes premiers résultats dans leurs équipes. Ils ont tous deux accepté spontanément la lourde tâche de rapporteur. Je leur en suis très reconnaissant.

Je suis très heureux que Guy Cousineau ait accepté de diriger « formellement » cette thèse. Elle s'inscrit autour du développement du langage Caml, dont il a été l'un des initiateurs.

Jean Goubault-Larrecq me fait l'honneur de présider le jury de cette thèse, malgré un emploi du temps chargé ; je l'en remercie. Toute ma gratitude va également à Giuseppe Castagna, qui a bien voulu faire partie de ce jury.

Je souhaite remercier les membres du projet Cristal — au sein duquel j'ai effectué cette thèse — et tout particulièrement Xavier Leroy, Michel Mauny et Didier Rémy.

Enfin, je tiens à saluer Matthieu Finiasz, Tom Hirschowitz et François Maurel, qui ont, chacun d'une manière différente, contribué à la réalisation de ce travail.

Table des matières

Introduction

Comment prévenir toute modification ou divulgation indésirable des données traitées par un système d'information, tout en permettant leur utilisation légitime ? Ce problème essentiel dans de nombreuses applications — médicales, financières ou militaires par exemple — est difficile : il faut non seulement effectuer un contrôle sur les *principaux* (individus, autres systèmes, etc.) qui ont accès au système mais également sur les *programmes* utilisés, puisque ceux-ci pourraient effectuer des opérations menaçant la confidentialité ou l'intégrité des données auxquelles ils ont accès. L'action illicite de tel ou tel programme peut de plus être causée par une erreur lors de sa réalisation, ou avoir été malicieusement introduite par son auteur : on parle alors de *Cheval de Troie*.

Une technique communément employée pour protéger les informations stockées dans un système est le *contrôle d'accès « à discrétion »* (*discretionary access control* en Anglais). Elle consiste à attacher à chaque donnée, ou, de manière plus générale, à chaque ressource, une annotation indiquant à quels utilisateurs ou programmes elle est légalement accessible, en lecture et en écriture. Un mécanisme d'identification permet ensuite de contrôler dynamiquement chaque accès, pour rejeter ceux qui ne sont pas autorisés. S'il répond à certains besoins élémentaires, un tel système n'offre cependant pas toujours une protection satisfaisante : une fois un droit d'accès offert, le détenteur des données ne peut restreindre ou contrôler leur utilisation. Les premiers travaux sur la sécurité des systèmes informatiques, en particulier ceux menés par Fenton [Fen73, Fen74], ou Bell et LaPadula [BL73], ont proposé un raffinement de ce procédé, appelé *contrôle d'accès « obligatoire »* (*mandatory access control* en Anglais). La propagation de l'information est régulée par le système en déterminant, de manière simultanée à l'exécution normale des programmes, les droits associés à chaque donnée. Outre le surcoût qu'elle induit à l'exécution, en temps de calcul et en occupation mémoire, cette méthode inscrite au *livre orange* du Ministère de la Défense américain [Dep85] se montre trop restrictive en pratique, empêchant presque tout usage légitime des données.

Analyse de flots d'information

Une troisième solution, plus prometteuse, consiste à analyser préalablement le système — ou, plus modestement, certains fragments de celui-ci —

programme auquel on souhaite conférer certains droits, de façon à déterminer quel usage il en fera. Si l'analyse statique garantit que son comportement sera acceptable, on peut lui accorder un accès sans restriction. Dans le cas contraire, son exécution sera refusée. Cette technique, appelée analyse de *flots d'information*, présente l'intérêt de ne reposer sur aucune notion de confiance – seule la correction de l'analyse importe et peut être prouvée formellement.

Ces difficultés ont suscité la recherche de techniques permettant de contrôler d'une manière plus précise la façon dont un programme dissémine l'information. L'analyse de flots d'information consiste à étudier, préalablement à son exécution, le code source d'un programme de manière à vérifier l'usage qu'il fera des droits qui lui sont accordés. Si l'analyse garantit que son comportement

sera acceptable, il peut être exécuté sans autre vérification. De part son caractère statique, cette approche présente deux avantages importants. Elle n'induit *a priori* aucun surcoût à l'exécution par rapport à un système qui n'effectuerait aucune vérification. De plus, elle peut prendre en compte *toutes* les traces d'exécution possibles du programme étudié, au lieu de la seule suivie dynamiquement par une exécution isolée, ce qui permet une analyse beaucoup plus fine de son comportement.

Cette idée apparaît à la fin des années 1970, dans la thèse de Denning [DD77, Den82]. Bien que les techniques d'analyse proposées soient manuelles et que leur correction ne soit pas prouvée, ces travaux sont intéressants ; plusieurs des idées introduites étant encore d'actualité. En particulier, pour décrire les flots d'information, Denning suggère d'adopter un *treillis de sécurité*, que je note $(\mathcal{L}, \leq_{\mathcal{L}})$, et d'affecter à chaque donnée ou ressource un *niveau d'information*, que je note ℓ, choisi parmi ce treillis. Un flot d'information est autorisé d'un niveau ℓ_1 vers un autre niveau ℓ_2 si et seulement si $\ell_1 \leq_{\mathcal{L}} \ell_2$. Ce treillis permet de décrire la *politique de sécurité* du système analysé. Dans les cas les plus simples, il peut être réduit à deux éléments, traditionnellement notés L (pour *low*) et H (pour *high*), tels que $\mathsf{L} \leq_{\mathcal{L}} \mathsf{H}$. Si on s'intéresse à la confidentialité, cela permet de distinguer les données *publiques* (de niveau L) des *secrètes* (de niveau H) ; dans le cas de l'intégrité, les données *sûres* de celles *non-sûres*. Certaines applications nécessitent de considérer des treillis de sécurité plus complexes : on peut naturellement penser aux classifications militaires (*secret*, *top-secret*, etc.) ou aux treillis des parties d'un ensemble de principaux, permettant de spécifier précisément qui peut lire ou produire chaque donnée. En considérant le *produit* de plusieurs de ces treillis, il est possible de composer des politiques de sécurité.

Les deux décennies suivantes voient se développer des recherches dans deux directions, de manière presque indépendante. D'une part, plusieurs chercheurs s'intéressent à la définition formelle des propriétés de sécurité : par exemple, les propriétés de *confidentialité* et d'*intégrité* qui nous intéressent sont des cas particuliers de la *non-interférence* introduite par Goguen et Meseguer [GM82]. On s'attache également à exprimer de telles propriétés en termes de sémantique formelle des programmes [Coh77, Coh78, McL92]. D'autre part, les travaux de Denning sont poursuivis, en faisant appel à différentes techniques pour analyser les programmes : systèmes logiques et preuves manuelles [AR80, BBL94], prouveurs [Fei80, MG85], vérificateurs de modèle (*model checkers* en Anglais) [FG95] et enfin systèmes de types [PØ95, ØP97].

Volpano, Smith et Irvine [VSI96] ont été les premiers, en 1996, à explicitement relier ces deux lignées de travaux, en traduisant une analyse dans le style de Denning pour un langage impératif élémentaire en un système de types puis en montrant que ce dernier vérifie une propriété de *non-interférence* directement exprimée en termes de la *sémantique* des programmes analysés. Cette voie a été poursuivie par différents chercheurs, qui ont considéré des traits de plus en plus élaborés, parmi ceux des langages de programmation modernes, et ont amélioré l'expressivité des systèmes de types utilisés [SM03]. Tout d'abord, Volpano et Smith [VS97b] ont augmenté leur premier langage de procédures et introduit une forme de polymorphisme pour les typer. Cette dernière est cependant très rudimentaire, puisqu'elle revient à *dupliquer* le code des procédures autant de fois qu'elles sont utilisées. Ils se sont également intéressés aux exceptions [VS97a], mais leur solution est trop restrictive pour être acceptable. Les premiers travaux concernant un langage fonctionnel sont dûs à Heintze et Riecke [HR98], qui ont défini le *SLam Calculus* (pour *Secure Lambda Calculus*), un λ-calcul doté d'un système de types pour l'analyse de flots d'information. (Ils ont également proposé une extension de ce langage avec effets de bord et concurrence, mais sans en prouver la correction.) Un système similaire est ensuite présenté par Abadi, Banerjee et les mêmes auteurs, mais d'une manière plus simple et plus générale [ABHR99]. Toujours dans le cadre d'un langage fonctionnel pur, Pottier et Conchon [PC00] ont montré comment il était possible d'extraire de tout système de type habituel (c'est-à-dire garantissant l'absence d'erreur à l'exécution) un système de types pour l'analyse de flots d'information. Ce procédé leur permet en particulier d'obtenir polymorphisme et inférence de types.

Sans pour autant minimiser leur intérêt, on peut dire que ces différents travaux ont essentiellement un caractère théorique : les calculs étudiés restent relativement éloignés de langages de programmation réalistes. En contraste franc, Myers [Mye99a, Mye99b] a considéré en 1999 l'ensemble du langage Java, incluant les objets, les exceptions et les classes paramétrées, mais sans montrer la correction de son système. Banerjee et Naumann [BN02] se sont également intéressés à Java, mais en donnant aux effets de bord une sémantique non-standard (et quelque peu surprenante).

C'est dans ce contexte que s'inscrit cette thèse. Elle a pour objet la formalisation et l'implantation d'une analyse de flot d'information pour le langage ML, langage fonctionnel doté de traits impératifs et d'un mécanisme d'exceptions. Le premier but — la formalisation — vise à combler l'écart entre les deux séries de travaux évoquées ci-avant, en donnant une preuve de correction pour une analyse de flots d'information portant sur un langage de programmation réaliste. Ce système doit être suffisamment simple et expressif pour être utilisable en pratique. Le deuxième objectif du travail — l'implémentation — me semble tout aussi important, tant le manque d'expérience concrète dans ce domaine est flagrant : lorsque j'ai débuté ce travail de thèse, à la fin de l'année 2000, aucun analyseur de flots d'information « réaliste » n'était à ma connaissance publiquement disponible. Il en existe aujourd'hui deux, qui ont été publiés à quelques mois d'écart : *Jif* pour le langage Java, développé à l'Université de Cornell par Myers et son équipe [MNZZ01], et *Flow Caml* [Sim], qui est décrit dans la première partie de cette thèse.

Une notion de flot d'information

J'ai jusqu'ici parlé de *flot d'information* dans les systèmes puis les programmes, sans pour autant avoir donné une définition exacte de cette notion. Toutes les analyses typées de flots d'information citées dans les paragraphes précédents s'intéressent à la *non-interférence*. Dans le cas d'un programme, cette propriété exprime l'indépendance entre certaines de ses entrées de niveau « haut » et certaines de ses sorties de niveau « bas », et correspond donc à l'*absence* de flot d'information. Comme cela a été remarqué par Abadi, Banerjee, Heintze et Riecke [ABHR99], une analyse de flots d'information n'est autre qu'une analyse de *dépendances*.

La définition de la non-interférence peut être rigoureusement exprimée en utilisant les outils habituels de la sémantique des langages de programmation. Considérons un langage de programmation arbitraire, dont les programmes, notés e, sont exécutés par une « machine » dont les états sont les éléments s d'un ensemble S. Dans ce modèle, l'exécution d'un programme e qui débute avec un état s peut soit produire un état s', soit diverger. Sa sémantique peut donc être décrite par une fonction $[\![e]\!]$ de S dans S, à condition de disposer d'un état distingué pour représenter la non-terminaison. La distinction entre les parties basses et hautes des états est donnée par une relation $\approx$ supposée réflexive et symétrique : on a $s_1 \approx s_2$ si et seulement si s_1 et s_2 coïncident sur leur parties basses. La relation $\approx$ correspond donc au pouvoir d'observation d'un attaquant de niveau « bas », c'est-à-dire à sa vision (partielle) de l'égalité entre états. On dit alors que le programme e vérifie la propriété de non-interférence pour $\approx$ si et seulement si

$$\forall s_1, s_2 \in S \;\; s_1 \approx s_2 \Rightarrow [\![e]\!](s_1) \approx [\![e]\!](s_2),$$

c'est-à-dire si deux exécutions indépendantes du programme ayant des états initiaux indistinguables produisent des états également indistinguables. Les propriétés de confidentialité et d'intégrité qui nous intéressent sont bien des cas particuliers de non-interférence : la confidentialité est obtenue en considérant les données publiques comme de niveau bas et les données secrètes de niveau haut. De manière duale, l'intégrité revient à voir les données sûres comme de niveau bas, et les données non-sûres de niveau haut.

La définition de la non-interférence que je viens de donner est *bipolaire* : elle introduit une dichotomie entre niveaux « bas » et « hauts », ce qui revient à effectuer une partition du treillis $\mathcal{L}$

en deux ensembles L et H respectivement clos inférieurement et supérieurement. Il serait possible de formuler une notion plus générale, séparant les niveaux en un nombre quelconque de classes. Cependant, cette propriété étendue pourrait être vue comme la superposition de plusieurs instances de la forme simple, chacune d'entre elles exprimant l'absence de flot d'information entre une paire d'un émetteur et d'un receveur.

La définition de la fonction $[\![\cdot]\!]$, comme celle de la relation $\approx$ à partir d'une partition (L, H) de $\mathcal{L}$, dépend naturellement du langage de programmation étudié. Il faut cependant garder à l'esprit qu'elles ne correspondent en général qu'à un *modèle* de l'exécution des programmes. En plus des entrées et sorties « normales », l'information peut transiter en utilisant d'autres voies, appelées *canaux cachés* (*covert channels* en Anglais) par Lampson [Lam73]. De manière non exhaustive, on peut citer :

- *La terminaison*, qui peut disséminer un bit d'information à partir de la terminaison ou de la non terminaison d'un programme. Par exemple, observer la terminaison du programme $\mathsf{if}\, x\, \mathsf{then}\, \mathit{loop}\, \mathsf{else}\, ()$ permet de déterminer le Booléen x.
- *Le temps d'exécution* peut divulguer de l'information à partir de la durée d'un calcul, ou de l'intervalle de temps séparant deux opérations particulières. Par exemple, une mesure précise du temps d'exécution de la boucle $\mathsf{for}\, i = 1\, \mathsf{to}\, x\, \mathsf{do}\, \mathit{wait}\, 1s$ donne de l'information sur sa borne x. (Notons que la terminaison peut être vue comme le cas particulier d'un temps d'exécution infini.)
- *La consommation d'une ressource partagée* comme par exemple la mémoire ou bien l'énergie nécessaire au fonctionnement du système.

L'étude des flots d'information engendrés par ces différents canaux est une question difficile, car elle dépend très étroitement de caractéristiques précises du système informatique sur lequel les programmes sont exécutés. Elles ne sont généralement pas prises en compte dans les modèles sémantiques usuels. Par exemple, Smith et Volpano [SV98] ont proposé un système visant à mesurer le temps d'exécution des programmes, en affectant un coût unitaire à chaque instruction. Ce modèle n'est cependant pas très réaliste. Par exemple, une analyse précise des temps d'accès à la mémoire nécessite sur la plupart des architectures de s'intéresser aux phénomènes de cache : le programme $\mathit{ignore}(\mathsf{if}\, x\, \mathsf{then}\, !\, r_1\, \mathsf{else}\, !\, r_2); !\, r_1$ effectue le même nombre d'instructions quelle que soit la valeur du Booléen x, à savoir deux lectures en mémoire. Cependant, la durée de leur réalisation est susceptible d'indiquer si les deux adresses consultées successivement sont identiques ou non.

Dans cette thèse, il a été choisi de ne prendre en considération aucun de ces canaux de communication, comme dans la quasi-totalité des travaux précédents. Ce choix de la simplicité paraît raisonnable pour une première expérimentation, tant les problèmes suscités par les canaux de communications « normaux » sont déjà importants. En particulier, je ne considérerai pas la terminaison des programmes — ce qui aurait de toutes façon peu de sens sans s'attaquer au problème plus général du temps d'exécution. Avec les définitions précédentes, cela revient à supposer que le résultat $\bot$ est indissociable de tout autre résultat, *i.e.* $\bot \approx s$ pour tout s. Un tel énoncé de non-interférence est dit *faible*.

Types annotés

La plupart des systèmes de types pour l'analyse de flots d'information qui ont été proposés depuis la fin des années 1990 sont construits, quel que soit le langage considéré, à partir d'un système de type « habituel » pour le même langage, en annotant ses types par des niveaux choisis dans le treillis $\mathcal{L}$. Par exemple, le type d'une valeur entière porte en général un niveau décrivant l'information qu'elle contient : ainsi, dans le treillis $\{\mathsf{L} \leq \mathsf{H}\}$, une valeur de type $\mathsf{int}\,\mathsf{L}$ est un entier public, tandis qu'un entier de type $\mathsf{int}\,\mathsf{H}$ est susceptible de porter une information secrète. On voit que les types décrivent à la fois la structure des valeurs et l'information qu'elles portent. Il n'est cependant pas possible — sauf dans certains cas particuliers — de séparer ces deux fonctionnalités, puisque les niveaux d'information doivent apparaître à l'*intérieur* des types pour permettre une

description suffisamment précise du contenu des valeurs : une paire d'entiers dont la première composante est secrète et la seconde publique peut ainsi recevoir un type de la forme $\mathsf{int}\,\mathsf{L} \times \mathsf{int}\,\mathsf{H}$. Grâce à aux annotations portées par son argument et son résultat, le type d'une fonction décrit précisément les flots d'information qu'elle engendre : par exemple, le type $\mathsf{int}\,\mathsf{L} \times \mathsf{int}\,\mathsf{H} \to \mathsf{int}\,\mathsf{L} \times \mathsf{int}\,\mathsf{H}$, qui s'applique à une fonction prenant deux arguments entiers et produisant également deux entiers, indique que le premier résultat ne dépend pas du deuxième argument, puisqu'un flot d'information du niveau H vers L n'est pas autorisé dans le treillis $\{\mathsf{L} \leq \mathsf{H}\}$.

Par ailleurs, ces systèmes sont généralement dotés de *sous-typage*, de manière à permettre une analyse raisonnablement fine des flots d'information, car *orientée*. L'ordre de sous-typage est obtenu à partir de celui sur les niveaux $\leq_{\mathcal{L}}$, en le « propageant » le long de la structure des types. On a par exemple $\mathsf{int}\,\mathsf{L} \leq_{\mathcal{L}} \mathsf{int}\,\mathsf{H}$, puisque $\mathsf{L} \leq_{\mathcal{L}} \mathsf{H}$: cette inégalité signifie qu'un entier *public* (*i.e.* de niveau L) peut être considéré comme *secret* (*i.e.* de niveau H), sans permettre une violation des règles de sécurité. Comme il est habituel, les types de fonctions sont contravariants à gauche et covariants à droite : par exemple, une fonction capable de traiter un entier secret peut prétendre qu'elle n'accepte que des entiers publics ; et, inversement, une fonction produisant un entier public peut prétendre produire un résultat secret. Ainsi, $\mathsf{int}\,\mathsf{L} \times \mathsf{int}\,\mathsf{H} \to \mathsf{int}\,\mathsf{L} \times \mathsf{int}\,\mathsf{H}$ est un sous-type de $\mathsf{int}\,\mathsf{L} \times \mathsf{int}\,\mathsf{L} \to \mathsf{int}\,\mathsf{H} \times \mathsf{int}\,\mathsf{H}$. Ce deuxième type donne cependant une description moins précise de la fonction que le premier, puisqu'il laisse penser que ses *deux* résultats sont susceptibles de dépendre de ses *deux* arguments.

Suivant la voie ouverte par Pottier et Conchon [PC00], il paraît nécessaire, pour obtenir un outil d'analyse utilisable, de s'intéresser à un système doté de *polymorphisme* et d'un algorithme d'*inférence* de types, en plus du sous-typage habituel. Pour motiver l'introduction de polymorphisme, considérons l'exemple simple de la fonction successeur. On souhaite pouvoir utiliser cette fonction avec des entiers de différents niveaux, et obtenir à chaque fois un résultat du même niveau que l'argument. Avec un système de types monomorphe, il est nécessaire de dupliquer la définition de la fonction en plusieurs exemplaires, chacun spécialisé pour un niveau particulier. Cela ne semble pas être une solution très réaliste à grande échelle, pour des raisons bien connues liées à l'entretien des programmes ou à la taille des exécutables générés, par exemple. Dans le système de Pottier et Conchon, le polymorphisme permet de donner un (schéma de) type à la fonction successeur qui est indépendant du ou des niveau(x) avec le(s)quel(s) elle est utilisée : $\forall \ell.\, \mathsf{int}\,\ell \to \mathsf{int}\,\ell$. La même fonction peut donc être utilisée avec des arguments de niveaux différents. Par ailleurs, l'inférence de types permet d'envisager la réalisation d'un vérificateur *automatique* des programmes, elle décharge le programmeur de la lourde tâche d'annoter manuellement par des niveaux d'information le code source du programme étudié.

Pourquoi ML ?

Avant de terminer cette introduction, indiquons brièvement pourquoi il a semblé pertinent de s'intéresser au langage ML pour accomplir les deux objectifs que nous nous sommes fixés. Tout d'abord, il paraît raisonnable de se restreindre, pour une première expérience pratique, à un langage de programmation séquentiel : en présence de traits concurrentiels, la terminaison d'un processus est observable par les autres processus. Ce canal supplémentaire de transmission de l'information nécessite *a priori*, pour être traité, un système plus restrictif ou plus complexe. Par ailleurs, la conception d'une analyse de flots d'information *typée* suggère de s'intéresser à un langage initialement doté d'un typage statique fort, comme c'est le cas de ML. De plus, pour les raisons évoquées ci-avant, je souhaite aboutir à un système de types doté de polymorphisme et d'inférence, deux propriétés que ML possède déjà. Enfin, les différents dialectes de ce langage peuvent être considérés comme de *vrais* langages de programmation : ils sont dotés de toutes les constructions habituelles — structures de données, valeurs mutables, exceptions, etc. Les différents compilateurs existants — pour SML [sml, mos] ou Caml [Ler, LDG+b] — sont aujourd'hui couramment utilisés pour réaliser des programmes complexes. Cependant, le noyau de ML reste relativement simple,

ce qui permet à sa sémantique d'être formalisée et bien comprise. Cela m'aidera à donner une définition précise de la propriété des programmes garantie par l'analyse de flots d'information directement reliée à leur sémantique, puis à prouver sa correction.

Inférence de types à base de contraintes

Je souhaite définir un système de types doté de sous-typage, de polymorphisme et d'un algorithme d'inférence de types. Ces trois requêtes m'amènent à considérer une forme particulièrement puissante de systèmes de types dits *à base de contraintes*. Ils sont apparus au début des années 1980 : à ma connaissance, le premier d'entre eux est dû à Mitchell [Mit84, Mit91]. Ces systèmes ont été largement étudiés depuis sous différentes formes, la formulation la plus élégante disponible à ce jour est certainement celle proposée par Odersky, Sulzmann et Wehr [OSW99] puis améliorée par Pottier et Rémy [PR03]. Elle est connue sous le nom de HM($\mathcal{X}$), car il s'agit d'une extension du système de types de Hindley–Milner (celui du langage ML) *paramétrée* par une logique du premier ordre $\mathcal{X}$. Ses formules, appelées *contraintes*, sont utilisées pour exprimer des hypothèses sur les variables de type. Leur syntaxe et leur sémantique ne sont pas totalement spécifiées par la définition de HM($\mathcal{X}$), plusieurs choix intéressants étant possibles. Ils donnent naissance à des systèmes de types différents. Par exemple, si la logique $\mathcal{X}$ permet d'exprimer des égalités entre types, le système obtenu — souvent appelé HM(=) — est équivalent au système de Hindley–Milner habituel [PR03]. En remplaçant ces égalités par des inégalités entre types, on obtient une extension de ce dernier dotée de sous-typage. Comme le système de Hindley–Milner, HM($\mathcal{X}$) offre du polymorphisme paramétrique : l'ensemble des types acceptables pour une expression est décrit par un *schéma de type*. Cependant, de manière à atteindre une expressivité suffisante, un tel schéma peut porter une contrainte qui borne la quantification universelle. Il s'écrit ainsi sous la forme

$$\forall \bar{\alpha}[C].\tau$$

où $\bar{\alpha}$ est un ensemble de variables de types universellement quantifiées, C une contrainte et τ un type. Il représente l'ensemble des types τ obtenus pour des instances des variables $\bar{\alpha}$ qui vérifient la contrainte C.

Outre sa grande généralité, un autre intérêt de HM($\mathcal{X}$) vient du fait qu'il permet un traitement particulièrement modulaire de l'inférence de types. En effet, celle-ci se ramène de manière systématique, c'est-à-dire indépendante de $\mathcal{X}$, à la résolution de contraintes dans cette logique. Plus précisément, étant donnée une expression e à typer, on dispose d'un algorithme très simple qui parcourt l'arbre syntaxique de e pour générer une contrainte C telle que e est bien typée si et seulement si C est satisfiable, *i.e.* admet une solution dans le modèle de la logique $\mathcal{X}$. De plus, si le résultat est positif, le schéma de type le plus général pour e est $\forall\alpha[C].\alpha$ où α est (par construction) l'unique variable libre de la contrainte C. Cette approche est intéressante, aussi bien pour l'étude formelle du processus d'inférence que son implémentation, car elle permet une présentation modulaire de l'algorithme, comme combinaison d'un générateur de contraintes et d'un solveur.

Le système de type que je présente dans cette thèse n'est pas une instance de HM($\mathcal{X}$), puisque ce dernier est un système « traditionnel » destiné à garantir l'absence d'erreur à l'exécution et ne s'intéresse donc pas aux flots d'information. Cependant, il suit très exactement les même règles de conception et, comme pour HM($\mathcal{X}$), l'inférence de types s'y ramène à la résolution de contraintes. Celles-ci font intervenir une forme de sous-typage dite *structurelle* : deux types comparables sont des arbres de même forme, qui ne diffèrent que par des annotations atomiques (ici, les niveaux de sécurité) portées par leurs feuilles.

Si la théorie du sous-typage structurel a été largement étudiée (voir l'introduction de la partie III (page 167) pour une présentation détaillée), mon système fait intervenir quelques formes de contraintes non-standard, nécessitant une extension des algorithmes classiques et de nouvelles preuves de correction. Par ailleurs, l'implantation efficace de ces algorithmes relève toujours du

folklore, c'est-à-dire que ses détails, s'ils sont connus de certains experts, restent délicats et non publiés. C'est pourquoi, en plus des problèmes directement relatifs à l'analyse de flots d'information, cette thèse s'intéresse également à la formalisation et l'implantation d'un algorithme de résolution de contraintes.

Plan

Le développement qui suit se décompose en trois parties. La première, intitulée **Le système Flow Caml**, (page 33) est une présentation de Flow Caml, un système d'analyse de flots d'information pour le langage Caml. Elle est construite à la manière d'un tutoriel : je décris progressivement les différentes caractéristiques de cet outil grâce à des exemples, en commençant par des fragments de code très simples et en terminant par un (petit) programme complet. Mon but est à la fois d'expliquer comment et pourquoi utiliser Flow Caml, et en même temps de donner quelques indications sur les problèmes rencontrés et les choix effectués lors de sa conception. Le lecteur pourrait être surpris de voir cette partie introduire ce mémoire, alors qu'elle décrit ce qui constitue d'une certaine manière l'achèvement de ce travail de thèse. Cependant, il m'a semblé qu'elle constituait une introduction intéressante et originale aux questions étudiées de manière plus abstraite dans les deux parties suivantes.

La deuxième partie, **Une analyse typée de flots d'information pour ML** (page 93), est consacrée à l'étude du système de types mis en œuvre par Flow Caml. Le langage étudié est réduit à son noyau, de manière à permettre un traitement mathématique rigoureux, mais toutes les fonctionnalités sont néanmoins représentées. Après une formalisation de la sémantique, je présente les règles de typage. Je donne ensuite la preuve de leur correction, qui est énoncée par un théorème de non-interférence. Je montre enfin comment l'inférence de types peut être ramenée à la résolution de contraintes — comme pour le système HM($\mathcal{X}$) — qui font intervenir du sous-typage structurel.

Enfin, dans la troisième partie, **Synthèse de types en présence de sous-typage structurel et de polymorphisme à la ML** (page 167), je m'intéresse à la seconde moitié du processus d'inférence, en donnant un algorithme de résolution pour les contraintes générées précédemment. Il est formalisé par des règles de réécriture sur les contraintes, ce qui permet à la fois d'expliquer son fonctionnement de manière précise et de prouver sa correction. Cet algorithme est utilisé dans le système Flow Caml pour effectuer la synthèse de types. Il pourrait cependant être utilisé dans tout autre système faisant appel à du sous-typage structurel.

Parcours de lecture

J'ai rédigé cette thèse de manière ce que chaque partie puisse être lue de manière indépendante des deux autres : chacune commence par une brève introduction, qui rappelle la problématique et donne une vue d'ensemble de la démarche suivie dans les chapitres suivants. Cependant, j'ai naturellement donné de nombreuses références croisées dans le texte, de manière à mettre en valeur les relations entre les différents résultats obtenus.

La première partie est presque entièrement informelle. Elle nécessite une expérience minimale de la programmation avec le langage Caml (ou une autre variante de ML), telle qu'on peut l'acquérir par la lecture du premier chapitre du manuel d'Objective Caml [LDG+a]. À l'inverse, aucune connaissance particulière relative à l'analyse de flots d'information n'est *a priori* requise : les questions relevant de ce domaine sont introduites progressivement au fil du texte.

Dans les deux parties suivantes, j'adopte un style plus mathématique ; le chapitre liminaire 1 présente le formalisme qui leur est commun. Une certaine connaissance formelle du langage ML — de sa sémantique et de son système de types — est probablement nécessaire : ces questions, aujourd'hui classiques, sont largement traitées dans la littérature, par exemple par Pottier et Rémy [PR03], auxquels j'ai d'ailleurs emprunté de nombreuses notations et définitions. Le lecteur qui désire comprendre l'essentiel du système de types développé dans cette thèse pourra lire les chapitres 5 et 6,

puis éventuellement le chapitre 8 (page 149). Le lecteur qui veut également avoir une idée de la manière dont l'inférence de types peut être réalisée y ajoutera les chapitres 7 et 10. Enfin, le lecteur intéressé par l'inférence de types en présence de sous-typage structurel, sans vouloir se pencher sur les questions relatives à l'analyse de flots d'information, peut directement lire les chapitres 10 et 11.

Conventions

Pour ne pas alourdir l'exposition, les définitions formelles sont généralement données au fil du texte ; seules celles qui nécessitent un énoncé complexe sont isolées et introduites par le mot *définition*. Les mots précisément définis sont cependant systématiquement indiqués en ***italique gras***, et indexés à la fin du mémoire.

Un énoncé intitulé *hypothèse* complète une ou plusieurs des définitions données précédemment, en introduisant de nouvelles conditions. Elles sont supposées vérifiées jusqu'à la fin de la partie dans laquelle elles apparaissent. Les résultats obtenus par une preuve formelle sont appelés *propriétés*, *lemmes* et *théorèmes*. La distinction entre ces trois dénominations n'est probablement pas indiscutable. J'ai cependant tenté d'appeler propriétés les résultats précisant des caractéristiques simples des objets manipulés, obtenues directement après leur définition. Les *lemmes* correspondent généralement à des résultats techniques intermédiaires, tandis que les *théorèmes* donnent les résultats essentiels.

CHAPITRE UN

1 Types et contraintes

Ce chapitre préliminaire présente le formalisme commun aux parties II et III de cette thèse. Il introduit une logique du premier ordre dont les variables, termes et formules sont respectivement appelés *variables de type*, *types* et *contraintes*. Sa définition est proche de celle donnée par Pottier et Rémy dans leur ouvrage sur l'inférence de type à la ML [PR03] ; cependant, je l'ai spécialisée au cas du sous-typage structurel qui m'intéresse dans cette thèse. Comme il est usuel, certains aspects de la définition sont laissés ouverts — par exemple le choix des constructeurs de types ou celui des prédicats primitifs — afin de conserver une certaine généralité et une meilleure abstraction dans chaque partie du développement. Ces points seront précisés dans les parties à venir, quand des hypothèses supplémentaires deviendront nécessaires.

1.1 Définitions préliminaires

1.1.1 Notations mathématiques

Dans cette thèse, je manipule les notions habituelles d'*ensembles*, *multi-ensembles*, de *relations* et de *fonctions*. Cette section décrit brièvement les notations employées.

▶ **Ensembles** L'ensemble vide est noté $\varnothing$, l'ensemble des entiers naturels $\mathbb{N}$, l'ensemble des entiers naturels strictement positifs $\mathbb{N}^*$ et l'ensemble des entiers relatifs $\mathbb{Z}$. La notation d'intervalle $[n_1, n_2]$ est utilisée pour désigner l'ensemble des entiers naturels compris (au sens large) entre n_1 et n_2. On note $\{x_1, \ldots, x_n\}$ l'ensemble fini dont les éléments sont $x_1, \ldots, x_n$.

On écrit $x \in E$ si et seulement si l'***élément*** x ***appartient*** à l'***ensemble*** E. Étant donnés deux ensembles E_1 et E_2, je note $E_1 \cup E_2$ leur union, $E_1 \cap E_2$ leur intersection, $E_1 \backslash E_2$ leur différence. On écrit $E_1 \subseteq E_2$ si E_1 est inclus (au sens large) dans E_2 et $E_1 \# E_2$ si E_1 et E_2 sont disjoints. J'écris $E_1 \uplus E_2$ pour $E_1 \cup E_2$ quand E_1 et E_2 sont disjoints.

Soit $n \in \mathbb{N}$. L'ensemble $E_1 \times \cdots \times E_n$ est le ***produit cartésien*** des ensembles $E_1, \ldots, E_n$, dont les éléments, appelés ***n-uplets*** sont notés $(x_1, \ldots, x_n)$ ou $x_1 \cdots x_n$, avec $x_1 \in E_1, \ldots, x_n \in E_n$. Dans le cas où $n = 0$, cet ensemble est réduit au 0-uplet, noté $\varnothing$ comme l'ensemble vide.

On note E^n le produit cartésien $E \times \cdots \times E$ (n fois), ses éléments étant appelés ***listes de longueur*** n ***sur*** E. Une ***liste d'association de*** E_1 ***dans*** E_2 est une liste l d'éléments de $E_1 \times E_2$ telle que pour chaque élément x de E_1 il existe au plus un élément y de E_2 tel que la

paire (x, y) soit un élément de l. Je dis dans ce cas que x est dans le domaine de l et j'écris $l(x)$ pour désigner l'élément y. Enfin, si x' n'est pas dans le domaine de l, je note $(l; x' : y')$ la liste d'association obtenue en ajoutant la paire (x', y') à la fin de l.

Toutes les notations relatives aux ensembles sont étendues aux multi-ensembles de la manière habituelle.

▶ **Relations** Soient $E_1, \ldots, E_n$ des ensembles. Une relation $\mathcal{R}$ sur $E_1, \ldots, E_n$ est une partie de $E_1 \times \cdots \times E_n$. L'appartenance à une relation est généralement notée de manière préfixe : on écrit $\mathcal{R}\, x_1 \cdots x_n$ pour $(x_1, \ldots, x_n) \in \mathcal{R}$; ou bien de manière infixe dans le cas $n = 2$: $x_1\ \mathcal{R}\ x_2$ est alors une abréviation pour $(x_1, x_2) \in \mathcal{R}$. On dit qu'une relation $\mathcal{R}'$ ***étend*** $\mathcal{R}$ si et seulement si $\mathcal{R}\, x_1 \cdots x_n$ implique $\mathcal{R}'\, x_1 \cdots x_n$, c'est-à-dire $\mathcal{R} \subseteq \mathcal{R}'$. Une ***relation binaire*** $\mathcal{R}$ sur un ensemble E est une relation sur E, E. Je note $\mathcal{R}^*$ la clôture réflexive transitive de $\mathcal{R}$. x est une forme normale pour $\mathcal{R}$ si et seulement si il n'existe pas d'élément y de E tel que $x\ \mathcal{R}\ y$. J'écris $x\ \mathcal{R}^{**}\ y$ si et seulement si $x\ \mathcal{R}^*\ y$ et y est une forme normale pour $\mathcal{R}$.

▶ **Fonctions** Soient E et F deux ensembles. Une ***fonction partielle*** φ de E vers F est une relation sur E et F telle que, pour tout élément x de E, il existe au plus un élément y tel que $(x, y) \in \varphi$. L'élément y est l'***image*** de x par φ et noté $\varphi(x)$. On dit que x est un ***antécédent*** de y pour φ. Le ***domaine*** de φ (noté : $\mathrm{dom}(\varphi)$) est le sous-ensemble de E pour les éléments duquel φ est définie. On dit que φ est ***totale*** si $\mathrm{dom}(\varphi) = E$. Inversement, son ***image*** (notée : $\mathrm{img}(\varphi)$) est le sous-ensemble des éléments de F qui ont un antécédent pour φ, et φ est dite ***surjective*** si $\mathrm{img}(\varphi) = F$. La fonction φ est ***injective*** si deux éléments de E distincts ont des images distinctes, et ***bijective*** si elle est à la fois injective et surjective. Enfin φ est ***quasi-constante*** (ou presque partout constante) si il existe un sous-ensemble E' de $\mathrm{dom}(\varphi)$ de complémentaire fini tel que φ est constante sur E', *i.e.* pour tous x et y de E', $\varphi(x) = \varphi(y)$.

Soient φ et φ' deux fonctions de E dans F. On dit que φ' ***étend*** φ si et seulement si $\mathrm{dom}(\varphi) \subseteq \mathrm{dom}(\varphi')$ et, pour tout $x \in \mathrm{dom}(\varphi)$, $\varphi(x) = \varphi'(x)$. Si $\mathrm{dom}(\varphi) \subseteq E' \subseteq E$, on dit que φ' ***étend*** φ ***sur*** E' si et seulement si φ' étend φ et $\mathrm{dom}(\varphi') = E'$. Soient $x \in E$ et $y \in F$. On désigne par $\varphi[x \mapsto y]$ la fonction de E dans F qui associe y à x et coïncide avec φ sur $E \backslash \{x\}$. De plus, on écrit $\varphi[\![x \mapsto y]\!]$ pour désigner $\varphi[x \mapsto y]$ dans le cas où $x \notin \mathrm{dom}(\varphi)$.

1.1.2 Noms, variables et meta-variables

Comme dans tout texte mathématique moderne, j'utilise dans cette thèse des noms ou *meta-variables* pour désigner les objets que je considère. Dans les domaines de la logique et des langages de programmation, une difficulté particulière apparaît : certains des objets mathématiques que l'on manipule contiennent eux-mêmes des noms, qui sont eux appelés simplement *variables*. Un traitement complet des difficultés afférentes — tel que celui proposé récemment par Gabbay et Pitts [GP02] — sort probablement du cadre de cette thèse. Il me semble toutefois utile de préciser, grâce à un exemple simple, l'approche que j'adopte dans la suite de ce mémoire. Considérons la définition inductive de la syntaxe abstraite du λ-calcul :

$$e ::= x \mid \lambda x.e \mid e\, e \qquad (\lambda\text{-terme})$$

Dans cette définition, les lettres e et x sont des *meta-variables* : la première est utilisée pour désigner un λ-terme arbitraire et la seconde une variable du λ-calcul. Dans la suite, j'utiliserai souvent les expressions « *le terme e* » ou « *la variable x* », là où il serait plus précis d'écrire « *le terme désigné par e* » ou « *la variable désignée par x* » (alors que « *les meta-variables e et x* » est tout-à-fait correct). La définition ci-dessus indique qu'un λ-terme est soit une variable, soit une paire d'une variable et d'une expression, soit une paire de deux expressions. Toutefois, elle distingue trop de λ-termes différents : on souhaiterait par exemple identifier les deux termes $\lambda x_1.x_1$ et $\lambda x_2.x_2$ quelles que soient les variables (désignées par) x_1 et x_2. Pour obtenir cela, suivant la théorie de Gabbay

et Pitts, il suffit de compléter la définition précédente en indiquant que « *la construction $\lambda x.e$ lie la variable x à l'intérieur de e* ». Alors, dans le modèle sous-jacent des *FM-sets*, le terme $\lambda x.e$ ne désigne plus une paire, mais l'*abstraction* de la variable x dans l'expression e, où le nom de la variable liée x n'a pas d'importance, on dit usuellement qu'il est α-convertible. (Une approche traditionnelle consisterait à effectuer un quotient de l'ensemble des arbres syntaxiques par une relation d'équivalence appropriée, mais cette vision des choses n'est ni très intuitive ni facile à utiliser rigoureusement en pratique.)

Une fois la position et la portée des liaisons données, deux notions classiques suivent : variables libres et substitution. L'ensemble $\mathrm{fv}(e)$ des *variables libres* d'un terme e et la *substitution sans capture* d'une variable x par une expression e' dans une expression e sont définies par :

$$\begin{array}{rcl@{\qquad}rcll} \mathrm{fv}(x) & = & \{x\} & x[x \leftarrow e'] & = & e' & \\ & & & x'[x \leftarrow e'] & = & x' & \text{si } x \neq x' \\ \mathrm{fv}(\lambda x.e) & = & \mathrm{fv}(e)\backslash\{x\} & (\lambda x'.e)[x \leftarrow e'] & = & \lambda x'.(e[x \leftarrow e']) & \text{si } x \neq x' \\ \mathrm{fv}(e_1\, e_2) & = & \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) & (e_1\, e_2)[x \leftarrow e'] & = & (e_1[x \leftarrow e'])\,(e_2[x \leftarrow e']) & \end{array}$$

On dit qu'un λ-terme e est *clos* s'il ne comporte pas de variable libre, *i.e.* $\mathrm{fv}(e) = \emptyset$. La définition de $(\lambda x'.e)[x \leftarrow e']$ donnée ci-dessus nécessite la condition $x = x'$. Cependant, celle-ci peut toujours être satisfaite en choisissant un nom x' approprié pour représenter l'abstraction $\lambda x'.e$. De plus, l'objet produit, $\lambda x'.(e[x \leftarrow e'])$, est indépendant du choix de ce nom, ce qui nous permet de voir la substitution comme une fonction.

Quelques difficultés supplémentaires apparaissent lorsque l'on considère des *contextes*, qui sont des λ-termes comportant un trou. Ils sont par exemple définis par la grammaire :

$$\mathbb{E} ::= [\,] \mid \lambda x.\mathbb{E} \mid \mathbb{E}\, e \mid e\, \mathbb{E} \qquad \text{(contexte)}$$

Étant donné un contexte $\mathbb{E}$ et un λ-terme e, $\mathbb{E}[e]$ désigne le λ-terme obtenu en remplaçant le trou de $\mathbb{E}$ par e. Les variables qui apparaissent dans un contexte à l'extérieur d'un λ-terme e sont importantes : par exemple, si x_1 et x_2 sont des variables différentes, $\lambda x_1.[\,]$ et $\lambda x_2.[\,]$ sont également des contextes différents. On dit que la variable x_1 est *définie* par le contexte $\lambda x_1.[\,]$. Ainsi, on distingue dans un contexte $\mathbb{E}$ les variables *libres*, dont l'ensemble est noté $\mathrm{fv}(\mathbb{E})$, des variables *définies*, dont l'ensemble est noté $\mathrm{dv}(\mathbb{E})$:

$$\begin{array}{rcl@{\qquad}rcl} \mathrm{fv}([\,]) & = & \emptyset & \mathrm{dv}([\,]) & = & \emptyset \\ \mathrm{fv}(\lambda x.\mathbb{E}) & = & \mathrm{fv}(\mathbb{E})\backslash\{x\} & \mathrm{dv}(\lambda x.\mathbb{E}) & = & \{x\} \cup \mathrm{dv}(\mathbb{E}) \\ \mathrm{fv}(\mathbb{E}\, e) = \mathrm{fv}(e\, \mathbb{E}) & = & \mathrm{fv}(\mathbb{E}) \cup \mathrm{fv}(e) & \mathrm{dv}(\mathbb{E}\, e) = \mathrm{dv}(e\, \mathbb{E}) & = & \mathrm{dv}(\mathbb{E}) \end{array}$$

On a alors $\mathrm{fv}(\mathbb{E}[e]) = \mathrm{fv}(\mathbb{E}) \cup \big(\mathrm{fv}(e)\backslash\,\mathrm{dv}(\mathbb{E})\big)$.

Terminons cette section par quelques définitions usuelles. Si y est une meta-variable parcourant un certain ensemble d'objets E, j'utilise la notation $\bar{y}$ pour désigner un ensemble fini (éventuellement vide) d'éléments de E, et $\vec{y}$ une liste (éventuellement vide) d'éléments de E. De plus, si $\bar{y}$ et $\vec{y}$ apparaissent dans le même énoncé, par convention l'ensemble $\bar{y}$ est supposé contenir exactement les éléments de $\vec{y}$, *i.e.* si $\vec{y} = (y_1, \ldots, y_n)$ alors $\bar{y} = \{y_1, \ldots, y_n\}$. Enfin, je note $\vec{y}_{|i}$ le i-eme élément de la liste $\vec{y}$.

1.2 Types bruts

Les *types bruts* sont la forme la plus élémentaire de types que nous rencontrerons dans cette thèse : il s'agit d'arbres construits à partir d'*atomes* et de *constructeurs de type*. Les types bruts ne comportent pas de variables. Munis d'un ordre partiel, le *sous-typage* — et éventuellement d'autres relations ou *prédicats* — ils forment un modèle dans lequel les types et les contraintes pourront être interprétés.

1.2.1 Définition

Soit $(\mathcal{L}, \leq_{\mathcal{L}})$ un treillis dont les éléments, notés ℓ, sont appelés ***atomes***. On note $\bot$ (resp. $\top$) son plus petit (resp. grand) élément et $\sqcup$ (resp. $\sqcap$) l'opérateur de plus petite borne supérieure (resp. grande borne inférieure). Soit $\mathcal{C}$ un ensemble dénombrable et non vide de ***constructeurs de types***, notés c. Soit $\mathcal{E}$ un ensemble dénombrable d'***étiquettes***, ses éléments et ses parties finies ou co-finies sont respectivement notés ξ et Ξ. Si ξ n'est pas dans Ξ, on note $\xi.\Xi$ l'ensemble $\{\xi\} \cup \Xi$. Les ***sortes*** sont définies comme suit :

$$\kappa ::= \mathsf{Atom} \mid \mathsf{Type} \mid \mathsf{Row}_{\Xi}\, \kappa \qquad \text{(sorte)}$$

Nous disposons de deux sortes de base, Atom pour les atomes et Type pour les types « normaux », c'est-à-dire ceux attribués aux programmes, ainsi que d'une sorte composite, $\mathsf{Row}_{\Xi}\, \kappa$, pour les *rangées* [Rém90]. Je note $\mathsf{Row}_{\Xi_1 \cdots \Xi_n}\, \kappa$ pour la sorte $\mathsf{Row}_{\Xi_1} \cdots \mathsf{Row}_{\Xi_n}\, \kappa$. Une ***signature*** d'arité n est une liste de n sortes. Les sortes permettent de restreindre la forme des types bruts par des règles de bonne formation. Pour cela, je suppose que chaque constructeur de type c est équipé d'une signature qui donne son arité et la sorte attendue pour chacun de ses paramètres. Le jugement $c :: \kappa_1 \cdots \kappa_n$ signifie précisément que le constructeur c a la signature $\kappa_1 \cdots \kappa_n$. L'énoncé suivant définit simultanément l'ensemble des types bruts et la sorte associée à chacun d'entre eux.

Définition 1.1 (Types bruts) *Un type brut de sorte* Atom *est un élément de $\mathcal{L}$. Un type brut de sorte* Type *est un terme de la forme $c\, t_1 \cdots t_n$ où $c :: \kappa_1 \cdots \kappa_n$ et, pour $i \in [1, n]$, t_i est un type brut de sorte κ_i. Un type brut de sorte $\mathsf{Row}_{\Xi}\, \kappa$ est une fonction quasi-constante associant à chaque étiquette de Ξ un type brut de sorte κ.* □

On note $\mathcal{T}$ l'ensemble des types bruts et $\mathcal{T}_\kappa$ l'ensemble des types bruts de sorte κ. De manière à ce que l'ensemble $\mathcal{T}_{\mathsf{Type}}$ ne soit pas vide, je suppose qu'il existe au moins un constructeur de type dont tous les paramètres ont une sorte de la forme $\mathsf{Row}_{\Xi_1 \cdots \Xi_n}\, \mathsf{Atom}$, où $n \in \mathbb{N}$. Un type brut de sorte $\mathsf{Row}_{\Xi}\, \kappa$ est appelé ***rangée brute*** et est généralement désigné dans la suite par la meta-variable r. Si $(t_\xi)_{\xi \in \Xi}$ est une famille de types bruts de sorte κ indexée par Ξ, on note $\{\xi \mapsto t_\xi\}_{\xi \in \Xi}$ la rangée brute de sorte $\mathsf{Row}_{\Xi}\, \kappa$ qui associe le type brut t_ξ à l'étiquette ξ. Si r est une rangée brute de sorte $\mathsf{Row}_{\Xi}\, \kappa$, t un type brut de sorte κ et ξ une étiquette qui n'est pas dans Ξ alors $(\xi : t; r)$ est la rangée brute de sorte $\mathsf{Row}_{\xi.\Xi}\, \kappa$ qui coïncide avec r sur Ξ et associe t à l'étiquette ξ. Si t est un type de sorte κ alors $\partial_{\Xi} t$ dénote la rangée brute qui associe à chaque étiquette de Ξ le type brut t (on écrit généralement ∂t pour désigner cette rangée lorsque le domaine Ξ peut être déduit du contexte). J'introduis également quelques notations particulières pour les rangées brutes d'atomes. Soit r une rangée brute de sorte $\mathsf{Row}_{\mathcal{E}}\, \mathsf{Atom}$. Je note $\Uparrow r$ (resp. $\Downarrow r$) la plus petite borne supérieure (resp. plus grande borne inférieure) de r, c'est-à-dire $\sqcup\{r(\xi) \mid \xi \in \mathcal{E}\}$ (resp. $\sqcap\{r(\xi) \mid \xi \in \mathcal{E}\}$). Pour toute partie Ξ finie ou co-finie de $\mathcal{E}$, $r_{|\Xi}$ désigne la rangée brute de sorte $\mathsf{Row}_{\mathcal{E}}\, \mathsf{Atom}$ définie par :

$$r_{|\Xi}(\xi) = \begin{cases} r(\xi) & \text{si } \xi \in \Xi \\ \bot & \text{sinon} \end{cases}$$

Enfin, si T est un ensemble de types bruts, je note $\uparrow T$ sa ***clôture supérieure***, c'est-à-dire l'ensemble $\{\, t \mid \exists t' \in T \;\; t' \leq t \,\}$.

La définition 1.1 doit être lue comme la construction d'un *plus petit* point fixe, je ne considère pas, dans le développement de cette thèse, de types récursifs. Ainsi, chaque type brut t a une ***hauteur*** $\mathrm{h}(t)$ qui est un entier positif défini par les égalités suivantes :

$$\begin{aligned} \mathrm{h}(\ell) &= 0 \\ \mathrm{h}(c\, t_1 \cdots t_n) &= 1 + \max\{\mathrm{h}(t_i) \mid i \in [1, n]\} \\ \mathrm{h}(\{\xi \mapsto t_\xi\}_{\xi \in \Xi}) &= 1 + \max\{\mathrm{h}(t_\xi) \mid \xi \in \Xi\} \end{aligned}$$

(Notons que l'ensemble $\{\mathrm{h}(t_\xi) \mid \xi \in \Xi\}$ est toujours fini, une rangée brute ayant par définition un image finie. Cela assure que la hauteur d'un type brut est toujours finie.)

$\otimes$	$+$	$-$	$\pm$
$+$	$+$	$-$	$\pm$
$-$	$-$	$+$	$\pm$
$\pm$	$\pm$	$\pm$	$\pm$

Figure 1.1 – Définition de la composition de variances

$$\text{LEQ-ATOM}\ \frac{\ell \leq_{\mathcal{L}} \ell'}{\ell \leq \ell'} \qquad \text{LEQ-TYPE}\ \frac{\forall i \in [1,n]\ \ + \in c.i \Rightarrow t_i \leq t'_i \quad \forall i \in [1,n]\ \ - \in c.i \Rightarrow t'_i \leq t_i}{c\,t_1 \cdots t_n \leq c\,t'_1 \cdots t'_n} \qquad \text{LEQ-ROW}\ \frac{\forall \xi \in \Xi\ \ t_\xi \leq t'_\xi}{\{\xi \mapsto t_\xi\}_{\xi\in\Xi} \leq \{\xi \mapsto t'_\xi\}_{\xi\in\Xi}}$$

Figure 1.2 – Définition du sous-typage entre types bruts

1.2.2 Sous-typage

Une ***variance*** ν est un sous-ensemble non vide de $\{-,+\}$ noté $-$ (*contravariant*), $+$ (*covariant*) ou $\pm$ (*invariant*). L'opérateur de composition de variances, noté $\otimes$, est défini par la table de la figure 1.1. Je suppose donnée, pour chaque constructeur c d'arité n, une liste de n variances, qui sont notées $c.1, \ldots, c.n$. Celles-ci sont généralement données avec la signature du constructeur, en écrivant $c :: \kappa_1^{\nu_1} \cdots \kappa_n^{\nu_n}$, où $\nu_i = c.i$ pour tout $i \in [1,n]$.

L'ensemble des types bruts est équipé d'une relation de ***sous-typage*** $\leq$ définie par les règles de la figure 1.2. Il s'agit d'une forme de sous-typage dite *structurelle* : deux types bruts comparables sont des arbres de même structure ne différant que par leur feuilles atomiques. Sur les atomes, le sous-typage coïncide avec l'ordre du treillis $(\mathcal{L}, \leq_{\mathcal{L}})$. Deux types bruts de sorte Type doivent avoir le même constructeur de tête c pour être comparables, et leurs paramètres doivent être reliés deux à deux suivant les variances de c. Enfin, le sous-typage est étendu point à point et de manière covariante aux rangées brutes.

Propriété 1.2 *La relation $\leq$ est un ordre partiel sur l'ensemble des types bruts. Deux types bruts comparables ont la même sorte et la même hauteur.* □

Muni de l'ordre partiel $\leq$, l'ensemble des types bruts ne forme pas un treillis — en particulier, il n'y a pas de type brut minimal ou maximal — mais une union disjointe de treillis. Pour expliciter cette structure, j'introduis la relation d'équivalence $\approx$ qui est définie comme la clôture symétrique transitive de $\leq$, ou, de manière équivalente, par les règles de la figure 1.3 (on écrit $\approx^+$ et $\approx^-$ pour $\approx$ et $\approx^\pm$ pour $=$). Cette relation étend la notion de *forme* introduite par Tiuryn [Tiu92]. Intuitivement, deux types bruts sont reliés par $\approx$ si et seulement si ce sont des arbres de même forme et ne diffèrent que par des feuilles atomiques reliées à la racine des types par un chemin ne comportant aucune position invariante. Sur chaque classe d'équivalence de la relation $\approx$, l'ordre partiel $\leq$ définit une structure de treillis.

Propriété et définition 1.3 *La relation $\approx$ est une relation d'équivalence. Sur chacune de ses classes d'équivalence, appelée* **squelette brut** *et notée T, l'ordre partiel $\leq$ définit une structure de treillis $(T, \bot_T, \top_T, \sqcup_T, \sqcap_T)$.* □

Propriété 1.4 *La relation $\approx$ est totalement vraie sur les types bruts de sorte $\mathsf{Row}_{\Xi_1 \cdots \Xi_n}\,\mathsf{Atom}$, i.e. l'ensemble $\mathcal{T}_{\mathsf{Row}_{\Xi_1\cdots\Xi_n}\,\mathsf{Atom}}\mathsf{Atom}$ est un squelette brut.* □

Propriété 1.5 *Si $t_1 \approx t_2$ alors t_1 et t_2 ont même hauteur.* □

On peut donc parler de la hauteur d'un squelette brut, qui est la hauteur commune de tous ses types bruts.

$$\text{SKK-ATOM}\quad \ell \approx \ell' \qquad \text{SSK-TYPE}\quad \frac{\forall i \in [1,n]\ \ t_i \approx^{c.i} t'_i}{c\,t_1 \cdots t_n \approx c\,t'_1 \cdots t'_n} \qquad \text{SSK-ROW}\quad \frac{\forall \xi \in \Xi\ \ t_\xi \approx t'_\xi}{\{\xi \mapsto t_\xi\}_{\xi\in\Xi} \approx \{\xi \mapsto t'_\xi\}_{\xi\in\Xi}}$$

Figure 1.3 – Définition de la relation $\approx$

$$\frac{\alpha \in \mathcal{V}_\kappa}{\vdash \alpha : \kappa} \qquad \vdash \ell : \mathsf{Atom} \qquad \frac{c :: [\kappa_1 \cdots \kappa_n] \qquad \forall i \in [1,n]\ \ \vdash \tau_i : \kappa_i}{\vdash c\,\tau_1 \cdots \tau_n : \mathsf{Type}}$$

$$\frac{\vdash \tau : \kappa \qquad \vdash \tau' : \mathsf{Row}_\Xi\,\kappa}{\vdash (\xi : \tau; \tau') : \mathsf{Row}_{\xi.\Xi}\,\kappa} \qquad \frac{\vdash \tau : \kappa}{\vdash \partial_\Xi \tau : \mathsf{Row}_\Xi\,\kappa}$$

Figure 1.4 – Règles de sortage des types

1.3 Types

Ayant introduit les types bruts, qui forment un modèle de la logique, je définis à présent les types, qui sont les composants élémentaires des contraintes. Leur définition est proche de celle des types bruts, avec quelques différences. Tout d'abord, les types sont des termes qui peuvent mentionner des variables : pour chaque sorte κ, je suppose donné un ensemble dénombrable $\mathcal{V}_\kappa$ de ***variables de type***. Les variables de type sont désignées, indépendamment de leur sorte, par les meta-variables α, β et γ. Je note ϕ les ***renommages*** des variables de types, c'est-à-dire les bijections de $\mathcal{V}$ sur lui-même qui respectent les sortes. D'autre part, la vision fonctionnelle des rangées que nous avons adoptée pour la construction du modèle n'est pas satisfaisante pour la conception d'algorithmes efficaces d'inférence de types ou, plus généralement, de résolution de contraintes. C'est pourquoi nous utilisons naturellement, dans la syntaxe des types, les termes de rangées de Rémy [Rém90]. Les ***types*** sont ainsi définis par la grammaire :

$$\tau ::= \alpha \mid \ell \mid c\,\vec{\tau} \mid (\xi : \tau; \tau) \mid \partial_\Xi \tau \qquad \text{(type)}$$

Cette définition syntaxique est restreinte par les règles de sortage données figure 1.4. Un type τ est bien formé si et seulement si il existe κ tel que le jugement $\vdash \tau : \kappa$ soit dérivable. Dans ce cas, la sorte κ est unique et appelée *sorte de* τ. Je ne considère par la suite que des types bien formés. J'introduis les notions habituelles de ***hauteur*** (noté $\mathrm{h}(\tau)$) et de ***taille*** (noté $\mathrm{s}(\tau)$) d'un type τ, qui sont définies par les égalités suivantes :

$$\begin{aligned}
\mathrm{h}(\alpha) &= 0 & \mathrm{s}(\alpha) &= 1\\
\mathrm{h}(\ell) &= 1 & \mathrm{s}(\ell) &= 1\\
\mathrm{h}(c\,\tau_1 \cdots \tau_n) &= 1 + \max\{\,\mathrm{h}(\tau_i) \mid i \in [1,n]\,\} & \mathrm{s}(c\,\tau_1 \cdots \tau_n) &= 1 + \textstyle\sum_{i\in[1,n]} \mathrm{s}(\tau_i)\\
\mathrm{h}(\xi : \tau_1; \tau_2) &= 1 + \max\{\,\mathrm{h}(\tau_1), \mathrm{h}(\tau_2)\,\} & \mathrm{s}(\xi : \tau_1; \tau_2) &= 1 + \mathrm{s}(\tau_1) + \mathrm{s}(\tau_2)\\
\mathrm{h}(\partial\tau) &= 1 + \mathrm{h}(\partial\tau) & \mathrm{s}(\partial\tau) &= 1 + \mathrm{s}(\partial\tau)
\end{aligned}$$

Un type est ***petit*** s'il est de hauteur inférieure ou égale à 1, c'est-à-dire d'une des formes suivantes : α, ℓ, $c\,\vec{\tau}$, $(\xi : \alpha_1; \alpha_2)$ et $\partial_\Xi \alpha$.

La signification des types est donnée par leur interprétation dans le modèle, qui est définie à partir de celles des variables de type. Une ***affectation*** φ est une fonction totale des variables de type vers les types bruts qui respecte les sortes, *i.e.* telle que, pour toute sorte κ, $\varphi(\mathcal{V}_\kappa) \subseteq \mathcal{T}_\kappa$. Les types sont interprétés dans le modèle en étendant les affectations grâce aux égalités suivantes :

$$\begin{aligned}
\varphi(\ell) &= \ell\\
\varphi(c\,\tau_1 \cdots \tau_n) &= c\,\varphi(\tau_1) \cdots \varphi(\tau_n)\\
\varphi(\xi : \tau; \tau') &= (\xi : \varphi(\tau); \varphi(\tau'))\\
\varphi(\partial\tau) &= \partial\varphi(\tau)
\end{aligned}$$

Cette extension est un homomorphisme sur les atomes et les types construits. La rangée $(\xi : \tau; \tau')$ est interprétée comme la rangée brute qui associe l'interprétation de τ à l'étiquette ξ et coïncide avec $\varphi(\tau')$ sur le reste de son domaine. Le lemme suivant montre que tout type brut peut être représenté par un type, de manière indépendante de l'interprétation des variables. Cette représentation n'est toutefois pas unique, en particulier puisque l'ordre des champs d'une rangée peut être choisi de manière arbitraire.

Lemme 1.6 *Soit t un type brut. Il existe un type τ tel que, pour toute affectation φ des variables de type, $\varphi(\tau) = t$.* □

⌜ *Preuve.* On procède par induction sur la structure de t.

∘ *Cas $t = \ell$.* On obtient le résultat voulu en posant $\tau = \ell$.

∘ *Cas $t = c\, t_1 \cdots t_n$.* Soit $i \in [1, n]$. Par hypothèse d'induction, il existe τ_i tel, que pour tout φ, $\varphi(\tau_i) = t_i$. On obtient le résultat recherché en posant $\tau = c\, \tau_1 \cdots \tau_n$.

∘ *Cas t est une rangée brute r de domaine Ξ.* Une rangée étant une fonction quasi-constante, il existe des étiquettes $\xi_1, \ldots, \xi_n$ et un type brut t_0 tels que, pour tout $\xi \in \Xi \backslash \{\xi_1, \ldots, \xi_n\}$, $r(\xi) = t_0$. Pour tout $i \in [1, n]$, on pose $t_i = r(\xi_i)$. Soit $i \in [0, n]$. Par hypothèse d'induction, il existe τ_i tel que, pour tout φ, $\varphi(\tau_i) = t_i$. On obtient le résultat recherché en posant $\tau = (\xi_n : \tau_n; \ldots \xi_1 : \tau_1; \partial\tau_0)$. ⌟

Étant donné un type τ, je note $\mathrm{ftv}(\tau)$ l'ensemble de ses variables libres. Je note également $\mathrm{ftv}^+(\tau)$ (respectivement $\mathrm{ftv}^-(\tau)$) le sous-ensemble de $\mathrm{ftv}(\tau)$ contenant les variables apparaissant en position covariante (respectivement contravariante) dans τ. Ces deux ensembles sont définis formellement comme suit :

$$
\begin{aligned}
\mathrm{ftv}^+(\ell) &= \varnothing & \mathrm{ftv}^-(\ell) &= \varnothing \\
\mathrm{ftv}^+(\alpha) &= \{\alpha\} & \mathrm{ftv}^-(\alpha) &= \varnothing \\
\mathrm{ftv}^+(c\,\tau_1 \cdots \tau_n) &= \Big(\bigcup_{c.i \in \{+,\pm\}} \mathrm{ftv}^+(\tau_i)\Big) \cup \Big(\bigcup_{c.i \in \{-,\pm\}} \mathrm{ftv}^-(\tau_i)\Big) & \mathrm{ftv}^-(c\,\tau_1 \cdots \tau_n) &= \Big(\bigcup_{c.i \in \{+,\pm\}} \mathrm{ftv}^-(\tau_i)\Big) \cup \Big(\bigcup_{c.i \in \{-,\pm\}} \mathrm{ftv}^+(\tau_i)\Big) \\
\mathrm{ftv}^+(\xi : \tau_1; \tau_2) &= \mathrm{ftv}^+(\tau_1) \cup \mathrm{ftv}^+(\tau_2) & \mathrm{ftv}^-(\xi : \tau_1; \tau_2) &= \mathrm{ftv}^-(\tau_1) \cup \mathrm{ftv}^-(\tau_2) \\
\mathrm{ftv}^+(\partial\tau) &= \mathrm{ftv}^+(\tau) & \mathrm{ftv}^-(\partial\tau) &= \mathrm{ftv}^-(\tau)
\end{aligned}
$$

Propriété 1.7 *Soient τ un type et φ_1, φ_2 deux affectations. Si, pour tout $\alpha \in \mathrm{ftv}^+(\tau)$, on a $\varphi_1(\alpha) \leq \varphi_2(\alpha)$ et, pour tout $\alpha \in \mathrm{ftv}^-(\tau)$, on a $\varphi_2(\alpha) \leq \varphi_1(\alpha)$ alors $\varphi_1(\tau) \leq \varphi_2(\tau)$.* □

1.4 Contraintes et schémas

1.4.1 Syntaxe

Les contraintes sont des formules logiques sur les types, construites à partir d'un ensemble arbitraire de ***prédicats***. On suppose que chaque prédicat p est donné avec un ensemble non vide de signatures, toutes ces signatures devant avoir la même arité n. On écrit $p :: \kappa_1 \cdots \kappa_n$ si $\kappa_1 \cdots \kappa_n$ est une signature du prédicat p. Chaque prédicat p d'arité n est muni d'une interprétation dans le modèle des types bruts, également notée p, qui est une relation sur $\mathcal{T}^n$. La définition de l'ensemble des prédicats est laissée ouverte. Je suppose toutefois qu'il inclut un prédicat binaire $\leq$ qui a la signature $\kappa \cdot \kappa$ pour toute sorte κ et qui est interprété dans le modèle par la relation de sous-typage $\leq$. Je présume également l'existence de deux prédicats d'arité 0, true et false, interprétés respectivement comme les constantes booléennes vraie et fausse.

Je suppose donné un ensemble dénombrable de *variables de programme*, notées x. Les schémas et contraintes sont mutuellement définis comme suit :

$$
\begin{aligned}
\sigma &::= \forall \bar{\alpha}[C].\tau && \text{(schéma)} \\
C &::= p\ \bar{\tau} \mid C \wedge C \mid \exists \bar{\alpha}.C \mid \mathsf{let}\, x : \sigma\, \mathsf{in}\, C \mid x \preceq \tau && \text{(contrainte)}
\end{aligned}
$$

Cette syntaxe est restreinte par les règles de sortage suivantes : ***(i)*** dans un schéma $\forall\bar{\alpha}[C].\tau$, le type τ doit avoir la sorte Type ; ***(ii)*** si les types $\tau_1, \ldots, \tau_n$ ont respectivement les sortes $\kappa_1, \cdots, \kappa_n$ alors, la contrainte $p\ \tau_1 \cdots \tau_n$ n'est bien formée que si $\kappa_1 \cdots \kappa_n$ est une signature de p. Dans un schéma $\forall\bar{\alpha}[C].\tau$ (resp. une contrainte $\exists\bar{\alpha}.C$), les variables de types $\bar{\alpha}$ sont liées dans C et τ (resp. C). Les ensembles des variables de type libres dans un schéma σ (noté $\mathrm{ftv}(\sigma)$) ou dans une contrainte C (noté $\mathrm{ftv}(C)$) sont définis en conséquence. Dans la contrainte $\mathsf{let}\, x : \sigma\ \mathsf{in}\ C$, la variable x est liée dans C. Les ensembles des variables de programme libres dans la contrainte C (noté $\mathrm{fpv}(C)$) ou un schéma σ (noté $\mathrm{fpv}(\sigma)$) sont définis en conséquence. Par abus de notation, j'écris parfois τ pour le schéma $\forall\varnothing[\mathsf{true}].\tau$. J'introduis enfin des formes de contraintes dérivées à partir des constructions précédentes. Soit σ le schéma $\forall\bar{\alpha}[C].\tau$; j'écris $\exists\sigma$ (lire : σ *a une instance*) pour la contrainte $\exists\bar{\alpha}.(C \wedge \tau \leq \tau')$, et, supposant $\bar{\alpha} \# \mathrm{ftv}(\tau')$, $\sigma \preceq \tau'$ (lire : τ' *est une instance de* σ) pour la contrainte $\exists\bar{\alpha}.(C \wedge \tau \leq \tau')$. J'utilise $\tau_1 = \tau_2$ comme abbréviation pour la conjonction $\tau_1 \leq \tau_2 \wedge \tau_2 \leq \tau_1$.

Avant de définir mathématiquement la sémantique des contraintes et des schémas *via* leur interprétation dans le modèle des types bruts, donnons quelques explications informelles sur le rôle de chacune des constructions. Tout d'abord, comme en ML, un schéma de type permet de représenter un ensemble potentiellement infini de types décrivant un (fragment de) programme. Leur forme est identique à celle de $\mathrm{HM}(\mathcal{X})$ [OSW99]. Elle étend celle de ML en autorisant un schéma à mentionner une contrainte arbitraire pouvant porter indifféremment sur ses variables libres ou quantifiées. Intuitivement, le schéma $\forall\bar{\alpha}[C].\tau$ représente l'ensemble des types τ pour des types $\bar{\alpha}$ *tels que la contrainte* C *soit satisfaite*. La signification des schémas de type est donnée de manière équivalente par les contraintes d'instanciation : en effet la contrainte $\sigma \preceq \tau'$ représente une condition minimale sur les variables de type libres dans σ et τ' pour que le type τ' soit une instance de σ.

Le langage de contraintes inclut la conjonction et la quantification existentielle qui ont leur sémantique habituelle. Il comprend également des constructions $\mathsf{let}\, x : \sigma\ \mathsf{in}\ C$ et $x \preceq \tau$, proposées par Pottier et Rémy [PR03]. Ces formes de contraintes permettent un traitement rigoureux de la généralisation et de l'instanciation des types, dans la description d'algorithmes d'inférence pour les systèmes à la ML, c'est-à-dire munis de polymorphisme let. En effet, ces opérations nécessitent la duplication de schémas de type. Cependant, dans une implémentation efficace, un schéma doit généralement être *simplifié* avant d'être copié : ainsi, avec les langages de contraintes habituels, il est *a priori* nécessaire de donner une définition mutuellement récursive des algorithmes de génération et de résolution des contraintes. L'introduction de la forme $\mathsf{let}\, x : \sigma\ \mathsf{in}\ C$ permet de séparer totalement la description de ces deux phases. En quelque sorte, cette construction lie, à l'intérieur de la contrainte C, la variable de programme x au schéma de type σ. Ce dernier peut alors être référencé par *nom* dans C, sans être dupliqué : pour chaque occurrence libre de x dans C la sous-contrainte $x \preceq \tau$ est interprétée comme $\sigma \preceq \tau$. Ainsi, lorsque les algorithmes habituels ajoutent le schéma σ à l'environnement (après l'avoir éventuellement simplifié), je peux générer le contexte $\mathsf{let}\, x : \sigma\ \mathsf{in}\ [\,]$. Quand ils prennent ensuite une copie du schéma σ pour l'instancier en un type τ, il me suffit de générer la contrainte $x \preceq \tau$. Le choix de la stratégie de résolution est ainsi totalement délégué au solveur de contraintes.

1.4.2 Sémantique

Un ***schéma brut*** s est un ensemble de types bruts de sorte Type clos supérieurement. Par abus de notation, un type brut t de sorte Type peut être vu comme le schéma brut $\uparrow\{t\}$. Un ***environnement brut*** X est une fonction partielle des identificateurs de programme vers les schémas bruts.

Les sémantiques des contraintes et des schémas de type sont définies de manière mutuellement récursive par les règles de la figure 1.5. Un jugement relatif à une contrainte C porte une affectation φ et un environnement brut X qui donnent respectivement l'interprétation des variables de type

$$\textsc{c-predicate}\ \frac{p\ \varphi(\tau_1)\cdots\varphi(\tau_n)}{\varphi, X \vdash p\ \tau_1 \cdots \tau_n} \qquad \textsc{c-and}\ \frac{\varphi, X \vdash C_1 \qquad \varphi, X \vdash C_2}{\varphi, X \vdash C_1 \wedge C_2} \qquad \textsc{c-exists}\ \frac{\varphi[\vec{\alpha} \mapsto \vec{t}], X \vdash C}{\varphi, X \vdash \exists \bar{\alpha}.C} \qquad \textsc{c-instance}\ \frac{\varphi(\tau) \in X(x)}{\varphi, X \vdash x \preceq \tau}$$

$$\textsc{c-let}\ \frac{\varphi, X \vdash \sigma \preceq s \qquad s \neq \varnothing \qquad \varphi, X[x \mapsto s] \vdash C}{\varphi, X \vdash \mathsf{let}\, x\ :\ \sigma\ \mathsf{in}\ C} \qquad \textsc{c-scheme}\ \frac{\varphi[\vec{\alpha} \mapsto \vec{t}], X \vdash C \qquad \varphi[\vec{\alpha} \mapsto \vec{t}](\tau) \leq t}{\varphi, X \vdash \forall \bar{\alpha}[C].\tau \preceq t}$$

Figure 1.5 – Interprétation des contraintes et des schémas

$$\begin{aligned}
(p\ \bar{\tau})[x \leftarrow \sigma] &= p\ \bar{\tau} \\
(C_1 \wedge C_2)[x \leftarrow \sigma] &= C_1[x \leftarrow \sigma] \wedge C_2[x \leftarrow \sigma] \\
(\exists \bar{\alpha}.C)[x \leftarrow \sigma] &= \exists \bar{\alpha}.(C[x \leftarrow \sigma]) && \text{si } \bar{\alpha} \# \mathrm{ftv}(\sigma) \\
(\mathsf{let}\, x\ :\ \sigma'\ \mathsf{in}\ C)[x \leftarrow \sigma] &= \mathsf{let}\, x\ :\ \sigma'[x \leftarrow \sigma]\ \mathsf{in}\ C[x \leftarrow \sigma] && \text{si } x \# \mathrm{fpv}(\sigma) \\
(x \preceq \tau)[x \leftarrow \sigma] &= \sigma \preceq \tau \\
(x' \preceq \tau)[x \leftarrow \sigma] &= x' \preceq \tau && \text{si } x \neq x' \\
(\forall \bar{\alpha}[C].\tau)[x \leftarrow \sigma] &= \forall \bar{\alpha}[C[x \leftarrow \sigma]].\tau && \text{si } \bar{\alpha} \# \mathrm{ftv}(\sigma)
\end{aligned}$$

Figure 1.6 – Substitution d'un identificateur dans une contrainte et un schéma

et des variables de programme libres dans C. Si le jugement $\varphi, X \vdash C$ est valide, on dit que φ et X *satisfont* ou *sont une solution de* C. Les jugements relatifs aux schémas sont de la forme $\varphi, X \vdash \sigma \preceq t$ et se lisent : *sous les hypothèses* φ *et* X*, le type brut* t *est une instance de* σ. Ces jugements permettent d'interpréter les schémas de type comme des schémas bruts dans le modèle : on écrit $\varphi, X \vdash \sigma \preceq s$ si et seulement si, pour tout $t \in s$, $\varphi, X \vdash \sigma \preceq t$ est dérivable.

La définition de la satisfaction des contraintes est dirigée par la syntaxe. Les règles trois premières règles, C-PREDICATE, C-AND et C-EXISTS sont habituelles. C-PREDICATE donne la sémantique des applications de prédicats en utilisant leur interprétation dans le modèle : la contrainte $p\ \tau_1 \cdots \tau_n$ est satisfaite par φ si et seulement si $\varphi(\tau_1), \cdots, \varphi(\tau_n)$ sont reliés par la relation p sur $\mathcal{T}^n$. C-AND exprime qu'une affectation et un environnement brut satisfont une conjonction si et seulement si ils satisfont chacun de ses membres. C-EXISTS permet aux variables $\bar{\alpha}$ quantifiées existentiellement de dénoter des types bruts arbitraires $\bar{t}$ dans C. L'environnement brut porté par les jugements de satisfaction est manipulé par les règles C-INSTANCE et C-LET. La contrainte $x \preceq \tau$ est satisfaite sous les hypothèses φ et X si $\varphi(\tau)$ appartient à l'interprétation du schéma lié à x dans le contexte, c'est-à-dire $X(x)$. Dans la règle C-LET, le schéma brut s représente un ensemble d'instances du schéma σ. L'environnement brut est alors étendu avec la liaison $x \mapsto s$, de telle sorte que les contraintes d'instanciation $x \preceq \tau$ qui apparaissent dans C interprètent la variable de programme x comme représentant le schéma σ. La deuxième prémisse, $s \neq \varnothing$, est utile dans le cas où x n'a pas d'occurrence libre dans C : elle assure que le schéma σ est tout de même instanciable.

Les jugements de la forme $\varphi, X \vdash \sigma \preceq t$ permettent une définition élégante de l'interprétation des schémas, de manière mutuellement récursive avec la satisfaction des contraintes. Cependant, il est parfois utile d'interpréter directement un schéma comme l'ensemble de ses instances, c'est-à-dire comme le schéma brut défini par :

$$[\![\sigma]\!]_{\varphi,X} = \{\, t \mid \varphi, X \vdash \sigma \preceq t \,\}$$

Par ailleurs, il est facile de vérifier que la validité des jugements $\varphi, X \vdash C$ et $\varphi, X \vdash \sigma \preceq t$, ainsi que le schéma brut $[\![\sigma]\!]_{\varphi,X}$, ne dépendent pas des valeurs prises par φ et X en dehors de $\mathrm{ftv}(C)$ et $\mathrm{fpv}(C)$ ou $\mathrm{ftv}(\sigma)$ et $\mathrm{fpv}(\sigma)$. En particulier, si la contrainte C n'a pas de variable de programme

libre, j'écris $\varphi \vdash C$ pour $\varphi, X \vdash C$, ce jugement ne dépendant alors pas de l'environnement X. De même, si $\mathrm{fpv}(\sigma) = \varnothing$, j'écris respectivement $\varphi \vdash \sigma \preceq t$ et $[\![\sigma]\!]_\varphi$ pour $\varphi, X \vdash \sigma \preceq t$ et $[\![\sigma]\!]_{\varphi,X}$.

Soit C_1 et C_2 deux contraintes. On dit que C_1 ***implique*** C_2 et on écrit $C_1 \Vdash C_2$ si et seulement si tout solution de C_1 est également une solution de C_2, *i.e.* $\varphi, X \vdash C_1$ implique $\varphi, X \vdash C_2$. On dit que C_1 et C_2 sont ***équivalentes*** et on note $C_1 \equiv C_2$ si et seulement si $C_1 \Vdash C_2$ et $C_2 \Vdash C_1$. On définit des notions similaires pour les schémas de type. Soient σ_1 et σ_2 deux schémas de type ; σ_1 est ***plus général*** que σ_2 (noté : $\sigma_1 \preccurlyeq \sigma_2$) si et seulement, pour tous φ, X et t, si $\varphi, X \vdash \sigma_2 \preceq t$ alors $\varphi, X \vdash \sigma_1 \preceq t$. On dit que σ_1 et σ_2 sont ***équivalents*** si σ_1 est plus général que σ_2 et σ_2 est plus général que σ_1.

1.4.3 Propriétés logiques des contraintes

Pour terminer cette section, je donne un ensemble de règles d'équivalence logiques sur les contraintes (figure 1.7). Ces lois constituent, pour la suite de cette thèse, une interface de haut niveau permettant de manipuler syntaxiquement les contraintes sans recourir à leur interprétation dans le modèle. Elles ne forment toutefois pas une axiomatisation de la logique — ce qui ne serait de toutes façons pas possible puisque l'ensemble des prédicats n'est pas fixé. Nous ne donnons pas la preuve de ces équivalences : à quelques détails cosmétiques près, elles sont identiques à celles énoncées et prouvées par Pottier et Rémy [PR03].

La règle LOG-CONTEXT montre que l'équivalence entre contraintes est une congruence pour les ***contextes de contraintes*** définis par la grammaire suivante :

$$\mathbb{C} ::= [\,] \mid \mathbb{C} \wedge C \mid C \wedge \mathbb{C} \mid \exists \bar{\alpha}.\mathbb{C} \mid \mathsf{let}\, x : \forall \bar{\alpha}[\mathbb{C}].\tau \;\mathsf{in}\; C \mid \mathsf{let}\, x : \sigma \;\mathsf{in}\; \mathbb{C} \qquad \text{(contexte)}$$

Je note $\mathrm{dtv}(\mathbb{C})$ et $\mathrm{dpv}(\mathbb{C})$ les ensembles respectifs de variables de type et de programme définis par un contexte $\mathbb{C}$. Les règles LOG-AND et LOG-ANDAND expriment respectivement la commutativité et l'associativité de la conjonction. LOG-DUP permet de supprimer l'un des membres d'une conjonction s'il est impliqué par l'autre membre. Par LOG-EX-EX, deux quantifications existentielles consécutives peuvent être regroupées et, par LOG-EX*, les quantifications inutiles retirées. LOG-EX-AND permet la commutation entre la quantification existentielle et la conjonction. LOG-EX-TRANS est une extension de la transitivité du sous-typage aux contraintes d'instanciation. Orientée de gauche à droite, elle peut être vue comme une règle de simplification permettant d'éliminer la variable locale α. LOG-IN-ID et LOG-IN* sont la traduction, en terme d'équivalences logiques, de la définition de l'interprétation des constructions let : grâce à LOG-IN-ID, chaque occurrence libre de l'identificateur x dans le contexte $\mathsf{let}\, x : \sigma \;\mathsf{in}\; [\,]$ peut être remplacée par le schéma σ. La première condition assure que l'occurrence de x considérée est bien libre, les deux autres conditions préviennent une capture des variables de type et identificateurs libres dans σ lors de la substitution. La règle LOG-IN* permet d'éliminer une construction let quand l'identificateur lié n'a plus d'occurrence dans la contrainte C. La règle LOG-IN-AND est comparable à LOG-EX-AND : elle permet la commutation entre la construction let et la conjonction. LOG-IN-AND* est une combinaison de LOG-IN-AND, LOG-IN* et LOG-DUP. Similairement à LOG-IN-AND, les règles LOG-IN-EX et LOG-LET-LET permettent la commutation d'une construction let avec un quantificateur existentiel ou une autre quantification let. Les conditions d'application assurent que les liaisons de variables de type ou d'identificateurs sont préservées lors des commutations. LOG-LET-AND permet à la contrainte C_1 d'être déplacée à l'extérieur du schéma $\forall \bar{\alpha}[C_1 \wedge C_2].\tau$ à condition qu'elle ne mentionne aucune des variables quantifiées universellement $\bar{\alpha}$. LOG-LET-DUP permet d'introduire un contexte let à l'intérieur d'un schéma, à condition qu'aucune capture n'intervienne. LOG-LET-EX exprime que les variables de type quantifiées existentiellement dans la contrainte d'un schéma de type peuvent de manière équivalente être quantifiées universellement au niveau du schéma lui-même. LOG-LET-EQ permet de modifier la racine d'un schéma en utilisant une égalité portée par la contrainte de ce dernier. LOG-EQ exprime qu'il est valide de remplacer des variables de types par des types auxquels elles sont égales. Orientée de gauche à droite, LOG-NAME autorise l'introduction de noms frais

$$\mathbb{C}[C_1] \equiv \mathbb{C}[C_2] \quad \textit{si } C_1 \equiv C_2 \qquad \text{LOG-CONTEXT}$$

$$\mathbb{C}[\mathsf{false}] \equiv \mathsf{false} \qquad \text{LOG-FALSE}$$

$$C_1 \wedge C_2 \equiv C_2 \wedge C_1 \qquad \text{LOG-AND}$$

$$(C_1 \wedge C_2) \wedge C_3 \equiv C_1 \wedge (C_2 \wedge C_3) \qquad \text{LOG-ANDAND}$$

$$C_1 \wedge C_2 \equiv C_1 \quad \textit{si } C_1 \Vdash C_2 \qquad \text{LOG-DUP}$$

$$\exists \bar{\alpha}.\exists \bar{\beta}.C \equiv \exists \bar{\alpha}\bar{\beta}.C \qquad \text{LOG-EX-EX}$$

$$\exists \bar{\alpha}.C \equiv C \quad \textit{si } \bar{\alpha} \# \mathrm{ftv}(C) \qquad \text{LOG-EX}^*$$

$$(\exists \bar{\alpha}.C_1) \wedge C_2 \equiv \exists \bar{\alpha}.(C_1 \wedge C_2) \quad \textit{si } \bar{\alpha} \# \mathrm{ftv}(C_2) \qquad \text{LOG-EX-AND}$$

$$\exists \gamma.(\sigma \preceq \gamma \wedge \gamma \leq \tau) \equiv \sigma \preceq \tau \quad \textit{si } \gamma \notin \mathrm{ftv}(\sigma, \tau) \qquad \text{LOG-EX-TRANS}$$

$$\mathsf{let}\, x : \sigma \,\mathsf{in}\, \mathbb{C}[x \preceq \tau] \equiv \mathsf{let}\, x : \sigma \,\mathsf{in}\, \mathbb{C}[\sigma \preceq \tau] \quad \textit{si } x \notin \mathrm{dpv}(\mathbb{C}) \textit{ et } \mathrm{dtv}(\mathbb{C}) \# \mathrm{ftv}(\sigma) \textit{ et } \{x\} \cup \mathrm{dpv}(\mathbb{C}) \# \mathrm{fpv}(\sigma) \qquad \text{LOG-IN-ID}$$

$$\mathsf{let}\, x : \sigma \,\mathsf{in}\, C \equiv \exists \sigma \wedge C \quad \textit{si } x \notin \mathrm{fpv}(C) \qquad \text{LOG-IN}^*$$

$$\mathsf{let}\, x : \sigma \,\mathsf{in}\, (C_1 \wedge C_2) \equiv (\mathsf{let}\, x : \sigma \,\mathsf{in}\, C_1) \wedge (\mathsf{let}\, x : \sigma \,\mathsf{in}\, C_2) \qquad \text{LOG-IN-AND}$$

$$\mathsf{let}\, x : \sigma \,\mathsf{in}\, (C_1 \wedge C_2) \equiv (\mathsf{let}\, x : \sigma \,\mathsf{in}\, C_1) \wedge C_2 \quad \textit{si } x \notin \mathrm{fpv}(C_2) \qquad \text{LOG-IN-AND}^*$$

$$\mathsf{let}\, x : \sigma \,\mathsf{in}\, \exists \bar{\alpha}.C \equiv \exists \bar{\alpha}.\, \mathsf{let}\, x : \sigma \,\mathsf{in}\, C \quad \textit{si } \bar{\alpha} \# \mathrm{ftv}(\sigma) \qquad \text{LOG-IN-EX}$$

$$\mathsf{let}\, x_1 : \sigma_1 \,\mathsf{in}\, \mathsf{let}\, x_2 : \sigma_2 \,\mathsf{in}\, C \equiv \mathsf{let}\, x_2 : \sigma_2 \,\mathsf{in}\, \mathsf{let}\, x_1 : \sigma_1 \,\mathsf{in}\, C \quad \textit{si } x_1 \neq x_2 \textit{ et } x_2 \# \mathrm{fpv}(\sigma_1) \textit{ et } x_1 \# \mathrm{fpv}(\sigma_2) \qquad \text{LOG-LET-LET}$$

$$\mathsf{let}\, x : \forall \bar{\alpha}[C_1 \wedge C_2].\tau \,\mathsf{in}\, C_3 \equiv C_1 \wedge \mathsf{let}\, x : \forall \bar{\alpha}[C_2].\tau \,\mathsf{in}\, C_3 \quad \textit{si } \bar{\alpha} \# \mathrm{ftv}(C_1) \qquad \text{LOG-LET-AND}$$

$$\mathsf{let}\, x_1 : \sigma_1 \,\mathsf{in}\, \mathsf{let}\, x_2 : \forall \bar{\alpha}[C_2].\tau_2 \,\mathsf{in}\, C \equiv \mathsf{let}\, x_1 : \sigma_1 \,\mathsf{in}\, \mathsf{let}\, x_2 : \forall \bar{\alpha}[\mathsf{let}\, x_1 : \sigma_1 \,\mathsf{in}\, C_2].\tau_2 \,\mathsf{in}\, C \quad \textit{si } \bar{\alpha} \# \mathrm{ftv}(\sigma_1) \textit{ et } x_1 \notin \mathrm{fpv}(\sigma_1) \qquad \text{LOG-LET-DUP}$$

$$\mathsf{let}\, x : \forall \bar{\alpha}[\exists \bar{\beta}.C_1].\tau \,\mathsf{in}\, C_2 \equiv \mathsf{let}\, x : \forall \bar{\alpha}\bar{\beta}[C_1].\tau \,\mathsf{in}\, C_2 \quad \textit{si } \bar{\beta} \# \mathrm{ftv}(\tau) \qquad \text{LOG-LET-EX}$$

$$\mathsf{let}\, x : \forall \bar{\alpha}[C_1 \wedge \tau_1 = \tau_2].\tau_1 \,\mathsf{in}\, C_2 \equiv \mathsf{let}\, x : \forall \bar{\alpha}[C_1 \wedge \tau_1 = \tau_2].\tau_2 \,\mathsf{in}\, C_2 \qquad \text{LOG-LET-EQ}$$

$$\vec{\alpha} = \vec{\tau} \wedge C[\vec{\alpha} \leftarrow \vec{\tau}] \equiv \vec{\alpha} = \vec{\tau} \wedge C \qquad \text{LOG-EQ}$$

$$\mathsf{true} \equiv \exists \bar{\alpha}.(\vec{\alpha} = \vec{\tau}) \quad \textit{si } \bar{\alpha} \# \mathrm{ftv}(\vec{\tau}) \textit{ et } \vec{\alpha} \textit{ distinctes} \qquad \text{LOG-NAME}$$

$$C[\vec{\alpha} \leftarrow \vec{\tau}] \equiv \exists \bar{\alpha}.(\vec{\alpha} = \vec{\tau} \wedge C) \quad \textit{si } \bar{\alpha} \# \mathrm{ftv}(\vec{\tau}) \qquad \text{LOG-NAME-EQ}$$

Figure 1.7 – Équivalences logiques

pour des types arbitraires. Lue de droite à gauche, elle permet à l'inverse l'élimination de variables locales dont la valeur a été déterminée. Enfin, LOG-NAME-EQ est une combinaison de LOG-EQ et LOG-NAME.

PREMIÈRE PARTIE

Le système Flow Caml

Flow Caml est une extension du langage Objective Caml dotée d'un système de types qui permet de décrire et de vérifier les flots d'information dans les programmes. En Flow Caml, les types sont décorés par des niveaux d'information choisis dans un treillis définissable par l'utilisateur (sections 2.1.1 et 4.2). Ces annotations décrivent la quotité d'information portée par chaque expression ou valeur du langage. Les types de Flow Caml offrent ainsi un formalisme permettant de décrire des propriétés de confidentialité ou d'intégrité des données qui doivent être respectées par les (fragments de) programmes auxquels ils sont associés. Grâce à son algorithme d'inférence complet, Flow Caml permet en outre de vérifier, de manière automatisée, ces propriétés dans des programmes de taille réaliste.

Outre les questions liées à l'analyse de flots d'information, un autre intérêt de Flow Caml réside dans les caractéristiques de son système de types : il s'agit en effet d'une des premières implémentations de taille réaliste d'un langage de programmation doté simultanément de sous-typage, de polymorphisme et d'un synthétiseur de types complet (section 2.2, page 40).

Le langage Flow Caml est un substantiel sous-ensemble d'Objective Caml, qui correspond presque exactement à son ancêtre Caml Special Light. Il comporte toutes les constructions de base du langage Caml, types de bases (section 2.1, page 35), structures de données et filtrage (section 2.1.3, page 38), fonctions de première classe (section 2.2.5, page 49) ; ainsi que ses constructions impératives : valeurs mutables (section 3.1, page 53) et exceptions (section 3.2, page 56). Flow Caml dispose également des types de données algébriques (section 4.1, page 65) ainsi que de la couche modulaire du langage Objective Caml (section 4.3, page 73). Le langage Flow Caml comprend également des déclarations qui permettent au programmeur de préciser la politique de sécurité vis-à-vis de laquelle il souhaite vérifier chaque partie d'un programme, c'est-à-dire spécifier les règles de confidentialité et d'intégrité qui doivent être respectées entre les différents principaux avec lequel il interagit (sections 4.2 et 4.4.2).

Cette partie est une présentation informelle de Flow Caml, à la manière d'un tutoriel. Dans les chapitres qui suivent, j'essaye à la fois de présenter les différentes fonctionnalités du système, et d'expliquer les choix de conception que j'ai été amené à faire lors de sa réalisation. Si une certaine connaissance d'un langage fonctionnel — tel que SML [MTHM97, sml], Caml [Ler, LDG+b] ou Haskell [Pey03] — est parfois utile, aucun pré-requis concernant l'analyse de flots d'information n'est en général supposé. La plus grande partie des chapitres qui suivent utilise la boucle interactive de Flow Caml (ou toplevel loop *en anglais), qui permet à l'utilisateur d'entrer successivement chacune des phrases de son programme — expression, définition, déclaration de type, etc. — qui est immédiatement typée par le système, et d'obtenir en réponse son type principal. Toutefois, pour terminer ce tutoriel, j'explique, à travers un exemple complet détaillé, comment il est possible d'écrire un programme autonome en Flow Caml, en utilisant le compilateur en ligne de commande, de manière à obtenir un exécutable.*

Cette partie est une version française du chapitre tutoriel du manuel de l'utilisateur du système Flow Caml, publié comme rapport technique par l'INRIA [Sim03b].

CHAPITRE DEUX

2

Types, contraintes et niveaux d'information

2.1 Types de base et niveaux d'information

Cette première section explique comment les types habituels de ML sont, dans Flow Caml, annotés par des niveaux de sécurité pour décrire les flots d'information.

2.1.1 Entiers

Commençons par la définition suivante :

```
let x = 1;;
x : 'a int
```

(Dans cette partie, les fragments de code Flow Caml apparaissent en caractères `courier romains`. Ils sont généralement suivis des réponses produites par la boucle d'interaction en *`courier oblique`*.) Mon premier exemple lie simplement l'identificateur `x` à la constante entière `1`. La boucle interactive répond que cette constante a le type `'a int`. En Flow Caml, le constructeur de type `int` a un paramètre, qui est un *niveau d'information*, lequel appartient à un treillis arbitraire. Ces annotations permettent au système de types de décrire les flots d'information. Dans l'exemple ci-dessus, le niveau de sécurité est une variable, `'a`. Comme toute variable apparaissant dans un (schéma de) type, elle est implicitement quantifiée universellement : ainsi, la réponse `x : 'a int` signifie que, en dehors de tout contexte, la constante `1` à laquelle est liée `x` peut avoir n'importe quel niveau.

Le niveau d'une telle constante peut être spécifié par une contrainte de type. Supposons que nous recevons trois entiers de trois sources nommées *Alice*, *Bob* et *Charlie* (de telles sources sont généralement appelées *principaux* dans la littérature) :

```
let x1 : !alice int = 42;;
val x1 : !alice int
let x2 : !bob int = 53;;
val x2 : !bob int
let x3 : !charlie int = 11;;
val x3 : !charlie int
```

En Flow Caml, chaque source peut être symbolisée par un niveau d'information comme `!alice`, `!bob` ou `!charlie` (n'importe quel identificateur alphanumérique précédé du symbole `!` convient). Initialement, ces niveaux sont des points incomparables dans le treillis : cela signifie que les principaux qu'ils représentent ne peuvent pas échanger d'information. J'expliquerai à la section 4.2 (page 70) comment autoriser certains flots d'information

Les définitions précédentes sont globales et persistent jusqu'à la fin de la session ; ainsi, nous pouvons les utiliser dans les prochaines phrases soumises au système :

```
x1 + x1;;
- : !alice int
x1 + x2;;
- : [> !alice, !bob] int
x1 * x2 * x3;;
- : [> !alice, !bob, !charlie]  int
```

La première expression ne contient de l'information que sur la constante `x1`, de telle sorte que son niveau d'information est `!alice`. La somme `x1 + x2` porte de l'information sur `x1` et `x2`. C'est pourquoi son niveau d'information doit être supérieur ou égal à ceux de `x1` et `x2` : en effet `[> !alice, !bob]` représente tout niveau qui est supérieur ou égal aux constantes `!alice` et `!bob`. Ceci peut être lu comme une *union symbolique* des deux niveaux. De même, le niveau d'information de la troisième expression doit être supérieur ou égal à `!alice`, `!bob` et `!charlie`.

Cependant, un peu d'expérience de la programmation avec le système Flow Caml montre qu'utiliser des niveaux d'information de manière explicite dans les programmes est, la plupart du temps, inutile : grâce au polymorphisme à la ML, des contraintes de sous-typage entre variables universellement quantifiées sont généralement suffisantes pour décrire les flots d'information générés par un fragment de code. En fait, la nécessité des niveaux d'information constants — les principaux — apparaît lors de l'interaction avec des entités extérieures (comme par exemple le système de fichiers ou le réseau). J'aborderai ce point à la section 4.2 (page 70). Pour l'instant, j'utilise les principaux `!alice`, `!bob` ou `!charlie` d'une manière relativement artificielle, dans le but de guider l'intuition du lecteur.

Pour continuer, je définis une fonction qui calcule le successeur d'un entier :

```
let succ = function x -> x + 1;;
val succ : 'a int -> 'a int
```

À nouveau, le système interactif détermine automatiquement le type principal (c'est-à-dire le plus général) qui correspond à cette définition : `'a int -> 'a int`. Il signifie que la fonction `succ` prend comme argument un entier d'un certain niveau `'a` et produit un autre entier dont le niveau d'information est identique : en effet, le résultat de cette fonction donne de l'information sur son argument. Puisque le type de `succ` est polymorphe, on peut appliquer cette fonction à des entiers de différents niveaux :

```
succ x1;;
- : !alice int
succ x2;;
- : !bob int
```

Cet exemple trivial est suffisant pour illustrer le caractère essentiel du polymorphisme sur les niveaux d'information dans un système de types pour l'analyse de flots d'information. Ici, avec un typage monomorphe, nous aurions dû écrire une version spécialisée de la fonction `succ` pour chaque niveau auquel elle est utilisée.

```
return_zero x1;;
- : !alice int
return_zero' x1;;
- : 'a int
```

En plus des entiers, le langage Flow Caml offre tous les types de données usuels : Booléens, flottants, caractères, etc. Comme `int`, les constructeurs de types correspondants portent un niveau

d'information :

```
let y0 = true;;
- : 'a bool
let z0 = 'v';;
val z0 : 'a char
```

Ces types ne suscitent pas de commentaire particulier, puisqu'ils se comportent exactement comme ceux des entiers pour ce qui concerne l'analyse de flots d'information. Par exemple, les opérations usuelles sur les nombres flottants sont disponibles en Flow Caml :

```
let pi = 3.14159265359;;
val pi : 'a float
let pi' = 4.0 *. atan 1.0;;
val pi' : 'a float
```

Les niveaux sont utilisés par Flow Caml pour décrire les flots d'information. Cela permet d'établir des instances d'une propriété appelée « *non-interférence* » qui exprime des absences de dépendances. Par exemple, si une entrée `x` d'un programme a le niveau `!alice`, alors le système de type garantit qu'un résultat qui ne porte pas ce niveau ne dépend pas de la valeur de `x` : si on exécute le programme avec différentes valeurs de `x`, le ou les résultats du programme ne sont pas affectés. Cette propriété sera formalisée dans la deuxième partie du mémoire. Nous pouvons cependant d'ores et déjà deux remarques. Tout d'abord, le système de type ne concerne que les résultats « normaux » du programme. Certains de ses effets observables en pratique ne sont pas pris en considération, comme par exemple le temps d'exécution ou la consommation de ressources.

D'autre part, il faut noter que les flots d'information sont analysés de manière « conservative » : dans le modèle sous-jacent, il y a un flot d'information d'une entrée vers une sortie dès lors que la connaissance de la dernière donne quelque information, même incomplète, sur la première. Considérons par exemple la fonction qui calcule la division euclidienne d'un entier par deux, grâce à un décalage logique :

```
let half = function x -> x lsr 1;;
val half : 'a int -> 'a int
```

Le schéma de type inféré pour `half` est exactement le même que celui obtenu pour `succ` : il reflète que le résultat produit par `half` révèle quelque connaissance de son argument. Cependant, cette fuite est seulement partielle, puisque, par exemple, il n'est pas possible de retrouver la valeur de `x1` à partir de `half x1`. Dans certaines situations, cela peut donner naissance à des types relativement surprenants au premier abord :

```
let return_zero = function x -> x * 0;;
val return_zero : 'a int -> 'a int
```

Dans cet exemple, le système détecte une dépendance entre l'entrée et la sortie de la fonction, bien qu'il n'y en ait clairement pas : `return_zero` retourne toujours zéro ! Cela provient du fait que, pour le système, le résultat d'un produit est toujours susceptible de donner de l'information sur ses deux facteurs, quels qu'ils soient. Bien entendu, si nous ré-écrivons la fonction comme suit :

```
let return_zero' = function x -> 0;;
val return_zero' : 'a int -> 'b int
```

nous obtenons une description plus fidèle de son comportement : dans le schéma de type inféré, le niveau du résultat (noté `'b`) n'est pas relié à celui de l'argument (`'a`), ce qui reflète l'absence de dépendance.

2.1.2 Chaînes

Il y a deux sortes de chaînes de caractères en Flow Caml : les immutables, de type `string`, et les mutables, de type `charray`. Cela diffère du langage Caml où toutes les chaînes sont mutables, même si, pour certains de leur usages, elles ne sont jamais modifiées en place. Cette distinction

supplémentaire est motivée par le fait que, dans un système de type analysant les flots d'information comme celui de Flow Caml, les valeurs mutables nécessitent une attention particulière. De ce fait, les types des opérations qui les concernent sont plus complexes (section 3.1, page 53).

Les chaînes de caractères immutables littérales apparaissent dans le code source des programmes entre guillemets :

```
let s = "Flow Caml";;
val s : 'a string
```

Comme illustré par cet exemple, le constructeur de type pour les chaînes immutables est `string`. Il a un paramètre, qui est un niveau décrivant la quotité d'information attachée à la chaîne. Le module `String` fournit des fonctions pour manipuler ces chaînes.

2.1.3 Listes

Les variables `'a`, `'b`, etc. que nous avons rencontrés jusqu'à présent représentent des *niveaux* du treillis. Dans le langage Flow Caml, le polymorphisme s'applique également aux *types* complets, comme en Objective Caml. Par exemple, définissons la fonction identité :

```
let id x = x;;
val id : 'a -> 'a
```

Dans ce schéma, la variable `'a` est un type. En effet, en Flow Caml, une variable qui apparaît dans un schéma peut être de différentes sortes : elle peut en particulier représenter un niveau ou un type (à la section 3.2 (page 56), nous verrons qu'elle peut également dénoter une *rangée*). Les sortes ne sont pas indiquées de manière explicite par le système (et le programmeur n'a pas à les préciser lorsqu'il entre une expression de type, par exemple dans une interface) car elles peuvent être déduites à partir du contexte. Le schéma de type pour la fonction identité ne fait apparaître aucune annotation explicitement : il est identique à celui donné en Objective Caml. Cependant, le type dénoté par `'a` peut contenir des niveaux d'information : par exemple, si on spécialise la fonction `id` aux entiers, on obtient `'b int -> 'b int`.

En Flow Caml, le constructeur de type `list` a deux paramètres (tandis qu'il n'en a qu'un seul en Objective Caml). Ainsi, dans le type `('a, 'b) list`, `'a` est une variable de *type* qui donne le type des éléments de la liste ; et `'b` est une variable de *niveau*, décrivant la quotité d'information attachée à la *structure* de la liste. Cela correspond par exemple à l'information obtenue en testant si la liste est vide.

```
let l1 = [1; 2; 3; 4];;
val l1 : ('a int, 'b) list
let l2 = [x1; x2];;
val l2 : ([> !alice; !bob] int, 'b) list
```

Comme il est usuel en ML, les fonctions qui manipulent les listes effectuent généralement un *filtrage* sur leur structure. Voici par exemple une fonction qui teste si une liste est vide :

```
let is_empty = function
    [] -> true
  | _ :: _ -> false
;;
val is_empty: ('a, 'b) list -> 'b bool
```

Dans ce type, le niveau du résultat Booléen, `'b`, est le même que celui de la liste donnée en argument, mais ne dépend pas du type des éléments de la liste, `'a` : en effet, la fonction ne peut révéler de l'information que sur la structure de la liste et pas sur son contenu. Les fonctions qui manipulent les listes sont souvent définies de manière récursive, mais cela ne soulève pas de difficulté particulière au niveau du typage :

```
let rec length = function
    [] -> 0
  | _ :: tl -> 1 + length tl
```

```
;;
val length: ('a, 'b) list -> 'b int
```

Le schéma de type obtenu pour `length` est similaire à celui de `is_empty` : la longueur de la liste ne donne aucune connaissance relative aux éléments de la liste, mais seulement sur sa structure. À l'inverse, une fonction qui teste si l'entier `0` apparaît dans une liste révèle de l'information à la fois sur sa longueur et son contenu, d'où le type inféré :

```
let rec mem0 = function
    [] -> false
  | hd :: tl -> hd = 0 || mem0 tl
;;
val mem0: ('a int, 'a) list -> 'a bool
```

Le module `List` de la bibliothèque standard offre toutes les fonctions usuelles pour manipuler les listes, dont les deux exemples suivants :

```
let rec rev_append l1 l2 =
  match l1 with
      [] -> l2
    | hd :: tl -> rev_append tl (hd :: l2)
;;
val rev_append: ('a, 'b) list -> ('a, 'b) list -> ('a, 'b) list
let rev l = rev_append l [];;
val rev: ('a, 'b) list -> ('a, 'b) list
```

2.1.4 Options

En ML, une option est une valeur qui peut avoir deux formes différentes : soit `None` (l'option vide), soit `Some v`, où `v` est une autre valeur, le *contenu* de l'option. Le type `option` se comporte, en Flow Caml, de la même manière que celui des listes. Il a deux arguments : dans `('a, 'b) option`, `'a` est le type du contenu de l'option, tandis que `'b` est un niveau de sécurité attaché à l'option elle-même, donnant la quotité d'information relative à sa forme. Cette lecture est illustrée par les fonctions suivantes :

```
let is_none = function
    None -> true
  | Some _ -> false
;;
val is_none: ('a, 'b) option -> 'b bool
```

La fonction `is_none` teste si une option est vide. Ainsi, le niveau du Booléen obtenu est exactement celui de l'option : le test est susceptible de révéler de l'information seulement sur la forme de son argument.

```
let default = function
    None -> 0
  | Some x -> x
;;
val default: ('a int, 'a) option -> 'a int
```

De manière similaire, la fonction `default` examine une option entière. S'il s'agit de `None`, elle retourne la valeur par défaut `0`, sinon le contenu de l'option lui-même. Ainsi, le résultat produit par une application de `default` donne de l'information à la fois sur la forme de l'option et son contenu.

2.1.5 n-uplets

Des n-uplets de longueur arbitraire sont également disponibles dans le langage Flow Caml : si $\mathtt{x}_1, \ldots, \mathtt{x}_n$ sont des valeurs dont les types respectifs sont $\mathtt{t}_1, \ldots, \mathtt{t}_n$, alors $(\mathtt{x}_1, \ldots, \mathtt{x}_n)$ est un n-uplet de type $\mathtt{t}_1$ `*` $\ldots$ `*` $\mathtt{t}_n$. Par exemple :

```
let pair0 = (0, true);;
val pair0 : 'a int * 'a bool
let triple0 = (0, 1, 'a');;
val triple0 : 'a int * 'a int * 'a char
```

Les types produits ne portent pas d'annotation particulière puisque — à l'exception près de l'égalité physique (section 2.2.3, page 46) — toute l'information que l'on peut extraire d'un n-uplet est en fait portée par ses éléments. Cependant, le type de chaque composant d'un n-uplet a son ou ses propres niveaux habituels, qui peuvent différer de ceux des autres composants. Par exemple, on peut définir une paire d'entiers dont le premier élément a le niveau `!alice` et le second élément le niveau `!bob` :

```
let pair1 = x1, x2;;
val pair0 : !alice int * !bob int
```

2.2 Schémas de types polymorphes contraints

Le système de type de ML, qui est la base des systèmes de type de SML [MTHM97, sml], Caml [Ler, LDG+b] ou Haskell [Pey03], repose sur l'unification : cela signifie que la seule relation exprimable entre deux types est l'égalité. Malheureusement, comme le montreront les exemples suivants, cela n'est pas suffisamment expressif pour décrire les flots d'information de manière satisfaisante dans beaucoup de situations. C'est pourquoi les schémas de type de Flow Caml doivent inclure des *contraintes* d'inégalités entre niveaux et/ou types, afin d'offrir une vue *orientée* des programmes.

2.2.1 Sous-typage

▶ **Inégalités entre niveaux** Considérons un premier exemple de code pour lequel Flow Caml infère un schéma mentionnant une inégalité : la fonction `f1` prend un entier `x` comme argument, et retourne la paire formée de son successeur et de sa somme avec la constante `x1` définie dans la section précédente.

```
let f1 x = (x + 1, x + x1);;
val f1 : 'a int -> 'a int * 'b int
         with 'a < 'b
         and  !alice < 'b
```

Le schéma de type produit par le système mentionne deux variables, `'a` et `'b`. La première, `'a`, est le niveau d'information associé à l'argument. Naturellement, il s'agit également de celui de la première composante de la paire retournée par la fonction. Le deuxième entier retourné par la fonction est annoté par la variable de niveau `'b`. Celle-ci est relié à `'a` *via* la première contrainte apparaissant après le mot-clef **`with`** : l'inégalité `'a < 'b` signifie que le niveau `'b` doit être supérieur ou égal à `'a`. (Notons que le symbole `<` imprimé sur le terminal par la boucle d'interaction doit être lu comme le symbole mathématique $\leq$.) En ce qui concerne l'analyse de dépendance, cette inégalité signale un flot d'information potentiel de l'argument de la fonction (étiqueté `'a`) vers le deuxième résultat (étiqueté `'b`). La deuxième inégalité, `!alice < 'b`, contraint le niveau d'information `'b` à être supérieur ou égal à la constante `!alice`. Elle montre que le deuxième résultat de la fonction porte également de l'information issue du principal *Alice* : l'entier `x1`.

Nous pouvons maintenant appliquer cette fonction à différents entiers :

```
f1 0;;
- : 'a int * !alice int
f1 x1;;
- : !alice int * !alice int
f1 x2;;
- : !bob int * [> !alice, !bob] int
```

Le schéma de type inféré pour `f1` signifie que toute instance de `'a int -> 'a int * 'b int` pour des niveaux `'a` et `'b` qui satisfont les inégalités `'a < 'b` et `!alice < 'b` est un type valide pour la fonction. Cet ensemble de types ne peut pas être exprimé d'une manière aussi précise dans un système de type à base d'unification. Dans un tel cadre, chaque occurrence de `<` devrait être lue comme une égalité, de telle sorte que, dans le schéma de `f1`, les variables `'a` et `'b` devraient toutes deux être unifiées avec la constante `!alice`. On obtiendrait ainsi le jugement suivant :

```
val f1 : !alice int -> !alice int * !alice int
```

qui est beaucoup plus restrictif que le précédent. Avec ce type pour `f1`, l'application `f1 0` produirait un entier de niveau `!alice int` (au lieu de `'a int`), tandis que l'expression `f1 x2` serait mal typée, à moins d'avoir déclaré l'inégalité `!alice < !bob` (section 4.2, page 70). La même observation peut être faite avec la fonction `f2` définie ci-dessous, qui prend trois entiers comme arguments, et calcule leurs sommes deux à deux :

```
let f2 x y z =
  (x + y, y + z, x + z)
;;
val f2 : 'a int -> 'b int -> 'c int -> 'd int * 'e int * 'f int
         with 'a < 'd, 'f
         and  'b < 'd, 'e
         and  'c < 'e, 'f
```

Le schéma de type obtenu comprend trois contraintes. Chacune d'entre elles relie le niveau d'un argument à ceux des deux résultats dans lequel il s'injecte. Par exemple, la contrainte `'a < 'd, 'f` (qui est une abréviation pour `'a < 'd and 'a < 'f`) décrit le flot d'information entre le premier argument, `x`, et les première et troisième composantes du résultat, `x + y` et `x + z`. Les deux contraintes suivantes procèdent de manière similaire avec les autres arguments. Naturellement, le système effectue quelque choix arbitraire lorsqu'il affiche la liste des contraintes d'un schéma. Par exemple, le schéma de type pour `f2` peut être écrit de manière équivalente comme suit :

```
val f2 : 'a int -> 'b int -> 'c int -> 'd int * 'e int * 'f int
         with 'a, 'b < 'd
         and  'b, 'c < 'e
         and  'a, 'c < 'f
```

Lorsque la fonction est appliquée aux trois constantes `x1`, `x2` et `x3`, les contraintes permettent de calculer les niveaux d'information respectifs des trois entiers produits :

```
f2 x1 x2 x3;;
- : [> !alice, !bob] int * [> !bob, !charlie] int
    * [> !alice, !charlie] int
```

À nouveau, si le système n'était pas muni de sous-typage mais seulement de contraintes d'unification, `f2` aurait un type beaucoup plus restrictif :

```
val f2 : 'a int -> 'a int -> 'a int -> 'a int * 'a int * 'a int
```

indiquant seulement que chaque composante du triplet résultat est susceptible de dépendre de chacun des trois arguments.

▶ Sous-typage entre types Les inégalités rencontrées dans les exemples précédents ne font intervenir que des niveaux d'information. Cependant, dans l'algèbre de types manipulée par Flow Caml, les types sont également ordonnés par l'ordre partiel `<`, lequel est alors appelé ordre de *sous-typage*. Le sous-typage de Flow Caml est *structurel* : il est défini en propageant l'ordre du treillis des niveaux à travers la structure des types. Ainsi, deux types comparables doivent avoir la même structure et ne peuvent différer que par leurs annotations. Pour cela, chaque constructeur de type (tel que `int`, `list` ou `->`) est équipé d'une *signature* qui donne la variance (et la sorte) de chacun de ses arguments. Une variance est soit `+` (*covariant*), `-` (*contravariant*) ou `=` (*invariant*). La signature d'un constructeur de type peut être affichée dans la boucle interactive grâce à la directive `#lookup_type` :

```
#lookup_type "int";;
type (#'a:level) int
```

Cette signature signifie que l'unique paramètre (`'a`) de `int` est un niveau et est covariant. (Le symbole `#` est une forme distinguée de `+` dont le rôle sera expliqué à la section 2.2.2 (page 44). Pour l'instant, il peut être lu comme `+`.) Cela définit l'ordre de sous-typage sur les types d'entiers : étant donnés deux niveaux d'information `'a` et `'b`, l'inégalité `'a int < 'b int` est vérifiée si et seulement si `'a < 'b`. Les deux paramètres du constructeur de type `list` sont également covariants :

```
#lookup_type "list";;
type (+'a:type, #'b:level) list = ...
```

Ainsi, `('a1, 'b1) list < ('a2, 'b2) list` est une contrainte équivalente à `'a1 < 'a2 and 'b1 < 'b2`. Les inégalités entre deux types de même forme se décomposent récursivement : par exemple `('a1 int, 'b1) list < ('a2 int, 'b2) list` se décompose successivement en

```
'a1 int < 'a2 int and 'b1 < 'b2
```

puis en

```
'a1 < 'a2 and 'b1 < 'b2
```

La fonction `f3` suivante accepte trois arguments, et construit trois listes de deux éléments chacune :

```
let f3 x y z =
  ([x; y], [y; z], [x; z])
;;
val f3 : 'a ->  'b -> 'c -> ('d, 'e) list * ('f, 'g) list * ('h, 'i) list
         with 'a < 'd, 'h
         and  'b < 'd, 'f
         and  'c < 'f, 'h
```

Le type inféré par le système est similaire à celui de `f2`. Chaque contrainte relie le type d'une des entrées à ceux des deux résultats qui en dépendent : ainsi, le type du premier argument, `'a` est « injecté » dans ceux des éléments de la première et de la troisième liste, reflétant la dépendance. Cependant, il faut bien noter que les variables `'a`, `'b`, `'c`, `'d`, `'e` et `'f` sont ici des variables de *types* et non de niveaux.

Le constructeur de type `->` que nous avons rencontré dans les exemples précédents a la signature suivante :

```
type (-'a:type) -> (+'b:type)
```

Comme il est usuel en présence de sous-typage, le type du résultat d'une fonction est covariant, tandis que celui de l'argument est contravariant. Cela signifie que l'inégalité `'a1 -> 'b1 < 'a2 -> 'b2` est vérifiée si et seulement si `'b1 < 'b2` et `'a2 < 'a1`.

▶ Simplification des schémas de types Flow Caml simplifie, de manière automatique, les schémas de type avant de les présenter à l'utilisateur. En effet, de par la présence de sous-typage, le même schéma de type peut être écrit de manières différentes mais équivalentes. Pour illustrer ce phénomène, considérons l'opérateur de somme entière `+`. Dans la bibliothèque standard de Flow Caml, il est déclaré avec le schéma de type suivant :

```
val ( + ) : 'a int -> 'a int -> 'a int
```

Ce schéma de type contraint apparemment les deux membres d'une somme à avoir le même niveau d'information, lequel est également celui du résultat. Cependant, il est quand même possible d'additionner deux entiers de niveaux différents, comme le montre l'expression suivante :

```
x1 + x2;;
- : [> !alice, !bob] int
```

L'entier `x1` a le type `!alice int`. Par sous-typage, il peut être librement utilisé avec n'importe quel super-type, par exemple `[> !alice, !bob] int`. (Le synthétiseur de types est capable d'effectuer

la coercion implicitement, aucune annotation du code source n'est nécessaire pour cela.) De même, la constante `x2` a le type `!bob int`, mais il peut également être utilisé comme une valeur de type `[> !alice, !bob] int`. Il s'ensuit que l'expression `x1 + x2` est bien typée et produit un résultat de type `[> !alice, !bob] int`. En généralisant ce procédé, on peut naturellement proposer un autre schéma de type pour `( + )` — équivalent au précédent — qui explicite le mécanisme de sous-typage :

```
val ( + ) : 'a1 int -> 'a2 int -> 'a3 int
            with 'a1, 'a2 < 'a3
```

Cependant, Flow Caml tente d'afficher chaque schéma de type sous une forme aussi concise que possible. Pour des raisons d'efficacité de l'inférence, son algorithme de simplification est toutefois incomplet, dans le sens où les résultats qu'il produit ne satisfont pas un critère de minimalité précis. De plus, dans le but de faciliter la lecture des types, chaque occurrence d'une variable est affichée sur le terminal avec une couleur qui indique sa polarité : les occurrences négatives apparaissent en vert, tandis que les occurrences positives sont en rouge. L'intérêt de ce code est que, lors de la lecture d'un schéma de type, chaque occurrence négative (resp. positive) d'une variable `'a` peut être remplacée par une variable fraîche `'b` et la contrainte `'b < 'a` (resp. `'a < 'b`). En appliquant ce principe au schéma

```
val ( + ) : 'a int -> 'a int -> 'a int
```

on obtient

```
val ( + ) : 'a1 int -> 'a2 int -> 'a3 int
            with 'a1 < 'a
            and  'a2 < 'a
            and  'a < 'a3
```

lequel devient, par la transitivité de `<`,

```
val ( + ) : 'a1 int -> 'a2 int -> 'a3 int
            with 'a1 < 'a3
            and  'a2 < 'a3
```

Pour interpréter un schéma de type produit par Flow Caml, il faut donc systématiquement prendre en compte la présence de sous-typage de manière « ambiante » dans le système. En particulier, l'identité de deux annotations ne doit pas être systématiquement interprétée comme la présence d'un flot d'information entre les deux entités correspondantes. Par exemple, dans le type inféré pour la fonction suivante

```
(fun x y -> let _ = x + y in ());;
- : : 'a int -> 'a int -> unit
```

les deux arguments peuvent recevoir le même niveau — puisque celui-ci est arbitraire et peut être rendu arbitrairement grand par sous-typage — bien qu'il n'y ait aucune dépendance entre les deux entiers. Ce type est équivalent à

```
- : : 'a int -> 'b int -> unit
```

(pour la notion d'équivalence définie à la section 1.4.2 (page 26). De même, le schéma suivant

```
(fun x y -> (x, x + y));;
- : 'a int -> 'b int -> 'a int * 'b int
    with 'a < 'b
```

pourrait laisser supposer un flot d'information entre le premier argument de `f` et le second. Cependant, l'inégalité `'a < 'b` ne contraint réellement que les occurences *contravariantes* de son membre gauche dans le corps du type, `'a` — c'est-à-dire la première — et celles *covariantes* de son membre droit, `'b` — c'est-à-dire la dernière. En effet, en appliquant le même procédé que celui détaillé ci-dessus pour `( + )`, on peut vérifier que le schéma donné par Flow Caml est équivalent au suivant :

```
- : 'a1 int -> 'b1 int -> 'a2 int * 'b2 int
    with 'a1 < 'a2
    and  'a1, 'b1 < 'b2
```

Les formes simplifiées des schémas de type peuvent parfois sembler moins lisibles, car elles donnent éventuellement une description moins explicite des flots d'information. La simplification des schémas est cependant essentielle lorsque l'on considère des fragments de code dont les types font intervenir un plus grand nombre de variables. Le système Flow Caml peut également produire une représentation graphique des schémas, qui donne une vision aussi explicite des flots d'information que les schémas alternatifs donnés dans les exemples précédents, tout en gardant une concision proche de celle des formes simplifiées. Elle est l'objet de la section 2.2.6 (page 50).

2.2.2 Contraintes `level`

Le langage Flow Caml dispose de la construction conditionnelle **`if`** ...**`then`** ...**`else`**, qui a la même sémantique qu'en Caml. Comme expliqué à la fin de la section 2.1.1 (page 35), le type des valeurs booléennes est annoté par un niveau d'information :

```
let y1 : !alice bool = false;;
val y1 : !alice bool
let y2 : !bob bool = false;;
val y2 : !bob bool
```

Lorsqu'elle atteint une construction conditionnelle, l'exécution d'un programme consiste à évaluer la condition puis, suivant le résultat, continuer avec l'une ou l'autre des branches données après les mots-clefs **`then`** et **`else`**. Ainsi, le type de l'expression entière doit être un super-type de ces sous-expressions. Par exemple, dans le cas simple de résultats entiers, cela signifie que le niveau d'information du résultat doit être (au moins) l'union de ceux portés par les branches :

```
if y0 then x1 else x2;;
- : [> !alice, !bob] int
```

Dans cet exemple, `x1` a le type `!alice int` et `x2` a le type `!bob int`, de telle sorte que l'expression entière a le type `[> !alice, !bob] int`.

La valeur produite par une expression conditionnelle donne également de l'information sur le résultat du test. C'est pourquoi, afin de prendre en compte ce possible flot d'information, le type du résultat doit être *gardé* par le niveau associé au test, c'est-à-dire porter un (des) niveau(x) supérieur(s) ou égal (égaux) à celui du test :

```
if y1 then 1 else 0;;
- : !alice int
```

Ici, la condition `y1` a le niveau `!alice`; de telle sorte que le résultat de l'expression toute entière doit également avoir ce niveau. De la même façon, si une expression conditionnelle produit un n-uplet, le type de chacune de ses composantes doit être gardé par le niveau du test :

```
if y1 then (x1, (true, 'a')) else (x2, (false, 'b'));;
- : [> !alice, !bob] int * (!alice bool * !alice char)
```

Je m'intéresse maintenant à des fonctions dont les résultats dépendent de quelques tests effectués sur leurs arguments. La fonction `int_of_bool` effectue simplement la conversion d'un Booléen en entier :

```
let int_of_bool y =
  if y then 1 else 0
;;
val int_of_bool : 'a bool -> 'a int
```

Son type n'appelle pas de commentaire particulier. Cependant, à cause du polymorphisme, il est possible que la forme du type décrivant le résultat d'une expression conditionnelle ne soit pas connue, par exemple car il est relié à celui d'un argument de fonction :

```
let choose x1 x0 y =
  if y then x1 else x0
;;
val choose : 'a -> 'a -> 'b bool -> 'a
             with 'b < level('a)
```

Le résultat produit par `choose` dépend clairement de la valeur du Booléen `y`. Son type doit donc être gardé par le niveau d'information `'b` de `y`. Cependant, la forme de ce type peut être arbitraire, puisqu'il n'est relié qu'à ceux des arguments de la fonction `x0` et `x1`. C'est la raison pour laquelle cet exemple introduit une nouvelle forme de contrainte, `'b < level('a)`. Remarquons tout d'abord que, dans cette contrainte, les variables `'a` et `'b` sont de sortes différentes : la première est un type alors que la seconde représente un niveau d'information. Cette contrainte peut être vue comme une inégalité suspendue jusqu'à ce que la forme du type `'a` soit connue : *grosso modo*, elle signifie que le ou les niveaux d'information extérieurs du type `'a` doivent être supérieurs ou égaux à `'b`. Par exemple, si on instancie `'a` par `'a1 int`, la contrainte se décompose en l'inégalité `'b < 'a1`. Mais, si `'a` est rendu égal à `'a1 int * 'a2 int`, elle produit deux inégalités : `'b < 'a1` et `'b < 'a2`. Pour observer ce mécanisme de décomposition, on peut considérer des applications partielles de la fonction `choose` :

```
let choose1 y = choose 1 0 y;;
val choose : 'a bool -> 'a int
let choose2 y = choose (1, 1) (0, 0) y;;
val choose : 'a bool -> 'a int * 'a int
```

Dans le premier exemple, `'a` est instancié en `'a int`, et la contrainte `'b < level('a1 int)` est décomposée en `'b < 'a1`. Cela donne le schéma :

```
'b bool -> 'a1 int with 'b < 'a1
```

qui est ensuite simplifié en `'a bool -> 'a int`. Dans le deuxième exemple, `'a` est instancié en `'a1 int * 'a2 int`. La contrainte `'b < level('a)` devient successivement :

```
'b < level('a1 int * 'a2 int)
'b < level('a1 int) and 'b < level('a2 int)
'b < 'a1 and 'b < 'a2
```

Le schéma de type obtenu peut alors être simplifié en `'a bool -> 'a int * 'a int`. De la même manière, on peut également considérer des listes :

```
let choose3 y = choose [] [1;2] y;;
val choose3 : 'a bool -> ('b, 'a) list
```

Ici, la variable de type `'a` est instanciée en `('a1, 'a2) list`. Cela donne la contrainte `'b < level(('a1, 'a2) list)` qui peut être décomposée en `'b < 'a2`.

Une question survient naturellement : comment le synthétiseur de type détermine-t-il sur quel(s) paramètre(s) de chaque constructeur de type le « destructeur » **`level`** doit se décomposer ? Cette information est donnée par les signatures : les paramètres sur lesquels **`level`** s'applique sont ceux qui sont marqués « gardés » par le symbole **`#`** :

```
#lookup_type "int";;
type (#'a:level) int
#lookup_type "list";;
type (+'a:type, #'b:level) list = ...
```

Cela signifie par exemple que **`level`** s'applique sur l'unique paramètre de `int` et le second de `list`. Rappelons que **`#`** est une forme distinguée de `+`, ce qui implique que les paramètres gardés sont toujours covariants. Cela assure que le destructeur **`level`** est croissant, *i.e.*

```
'a < level('b1) and 'b1 < 'b2
```

implique

```
'a < level('b2)
```

2.2.3 Contraintes content

Flow Caml supporte les primitives génériques de comparaison du langage Caml, telles que = ou <=. Ces opérations peuvent être utilisées pour comparer des structures de données de type arbitraire. Elles ont ainsi, en Objective Caml, le type suivant :

```
'a -> 'a -> bool
```

qui signifie qu'elles attendent deux arguments du même type et retournent un Booléen. En Flow Caml, le type du résultat doit porter une annotation, disons 'b. Ce Booléen est susceptible de contenir de l'information sur n'importe quel fragment des valeurs comparées. Pour décrire ce flot potentiel, le niveau 'b doit être relié à ceux qui décrivent les deux valeurs, c'est-à-dire qui apparaissent dans le type 'a. Par exemple, des versions de l'égalité spécialisées aux entiers, paires d'entiers et listes d'entiers devraient avoir les types suivants :

```
val eq_int : 'a int -> 'a int -> 'b bool
             with 'a < 'b
val eq_int_pair : ('a1 int * 'a2 int) -> ('a1 int * 'a2 int) -> 'b bool
                  with 'a1, 'a2 < 'b
val eq_int_list : ('a1 int, 'a2) list -> 'b bool
                  with 'a1, 'a2 < 'b
val eq_int_pair_list : ('a1 int * 'a2 int, 'a3) list -> 'b bool
                       with 'a1, 'a2, 'a3 < 'b
```

En effet, comparer deux paires d'entiers peut révéler de l'information à propos de chaque composante de chaque paire, tandis que la comparaison de deux listes est susceptible de faire fuir de l'information aussi bien sur leur structure (*e.g.* leur longueur) que leurs éléments.

On observe dans tous les cas que le niveau d'information 'b qui étiquette le résultat de la comparaison doit être supérieur ou égal à *tous* les niveaux qui apparaissent dans le type des arguments. Pour donner un type principal et polymorphe à ces comparaisons, nous avons besoin d'une forme supplémentaire de contraintes, **content**('a) < 'b, où 'a est un type et 'b un niveau. Elle requiert que chaque niveau d'information qui apparaît dans le type 'a soit inférieur ou égal à 'b. Comme **level**, le « destructeur » **content** se décompose récursivement en suivant la structure des types. Par exemple, l'inégalité **content**('a1 int * 'a2 int) < 'b devient successivement

```
content('a1 int), content('a2 int) < 'b
```

puis

```
'a1, 'a2 < 'b
```

Cette définition mime au niveau des types le comportement des opérateurs de comparaison tels que = et <= qui traversent les structures de données récursivement.

Les fonctions = et <= ont ainsi les types suivants en Flow Caml :

```
val ( = ) : 'a -> 'a -> 'b bool
            with content('a) < 'b
val ( <= ) : 'a -> 'a -> 'b bool
             with content('a) < 'b
```

Elles peuvent par exemple être utilisées pour écrire une fonction polymorphe mem qui cherche si un élément est membre d'une liste :

```
let rec mem x = function
    [] -> false
  | hd :: tl -> (x = hd) || mem x tl
;;
val mem : 'a -> ('a, 'b) list -> 'b bool
          with content('a) < 'b
```

ou bien un tri par insertion de listes :

```
let rec insert x = function
    [] -> [x]
```

```
  | hd :: tl -> (min hd x) :: insert (max hd x) tl
;;
 val insert : 'a -> ('a, 'b) list -> ('c, 'b) list
              with 'a < 'c
              and  content('a) < level('c)
let rec sort = function
    [] -> []
  | hd :: tl -> insert hd (sort tl)
;;
val sort : ('a, 'b) list -> ('c, 'b) list
           with 'a < 'c
           and  content('a) < level('c)
```

Avec Objective Caml, il est possible d'appliquer les primitives génériques de comparaison à des valeurs *fonctionnelles* : par exemple, si `f1` et `f2` sont deux fonctions, `f1 = f2` retourne `true` si `f1` et `f2` sont représentées par la même clôture en mémoire, ou bien, dans tous les autres cas, lève une exception. Cette sémantique a un intérêt limité, et n'est pas vraiment utilisable puisqu'elle dépend très largement de l'implémentation : par exemple, `let f = fun x -> x in f = f` retourne `true` tandis que `(fun x -> x) = (fun x -> x)` lève une exception. Cependant, le système de type d'Objective Caml n'a pas de moyen pour empêcher les fonctions d'être utilisées comme argument pour `=`. Le langage SML [MTHM97] traite ce problème d'une manière différente en introduisant des types *égalité* et refusant, par le typage, l'application des primitives de comparaisons à des valeurs dont le type n'est pas *égalité*, puisqu'elles sont susceptibles de contenir des clôtures. La même approche est suivie en Flow Caml, où les types non-*égalité* sont marqués par le mot-clef **noneq** dans leur définition, et où la contrainte `content('a) < 'b` ne peut pas être satisfaite si `'a` n'est pas un type *égalité*. Ainsi, le fragment de code suivant produit une erreur de type :

```
(fun x -> x) = (fun x -> x);;
Generic primitives cannot be applied on expressions
of type ~a -> ~a
```

L'opérateur `==`, appelé *égalité physique* est une primitive de comparaison générique particulière : il permet de tester si deux structures de données sont représentées par le même objet en mémoire. En Flow Caml, cette primitive reçoit le même type que l'égalité structurelle `=`

```
val ( == ) : 'a -> 'a -> 'b bool
             when content('a) < 'b
```

Ce schéma de type décrit le comportement de l'opérateur `==` de manière relativement peu précise, en laissant croire qu'il parcourt toute la structure de ses arguments. Par exemple, en spécialisant cet opérateur aux références entières obtient le type suivant

```
('b int, 'a) ref -> ('c int, 'a) ref -> 'a bool
with 'b, 'c < 'a
```

D'après ce type, le Booléen retourné par la comparaison physique de deux références entières peut donner de l'information à la fois sur leurs contenus et leurs adresses, alors que seules ces dernières sont examinées par `==`. En fait, le type suivant serait parfaitement valide :

```
('b int, 'a) ref -> ('c int, 'a) ref -> 'a bool
```

En généralisant cette observation, on pourrait donner un type plus précis pour `==` en introduisant un nouveau destructeur, **address**, dont le comportement serait exactement symétrique à celui **level** :

```
val ( == ) : 'a -> 'a -> 'b bool
             when address('a) < 'b
```

La contrainte `address(('a, 'b) ref) < 'c` serait par exemple équivalente à `'b < 'c` et `address ('a int) < 'b` équivalente à `'a < 'b`.

J'ai cependant choisi de ne pas introduire cette forme de contraintes pour deux raisons. J'ai tout d'abord souhaité limiter autant que possible la complexité du système de types de Flow

Caml de manière à ce qu'il soit pratiquement utilisable. Les situations où le type donné à l'égalité physique n'est pas assez précis me semblent suffisamment rares pour ne pas justifier l'introduction d'une nouvelle forme de contraintes. De plus, dans les cas où un type précis est nécessaire, il est généralement possible d'introduire une version monomorphe spécialisée de l'opérateur. D'autre part, la forme des type produits ne permet pas une définition satisfaisante de `address`. On peut certes décomposer la contrainte `address('a1 * 'a2) < 'b` en `address('a1) < 'b and address ('a2) < 'b`, mais, à nouveau, on laisse croire que `==` traverse les paires, ce qui n'est pas le cas. Pour contourner ce problème, il faudrait ajouter une annotation supplémentaire aux types produits, décrivant l'*adresse* physique des n-uplets.

2.2.4 Contraintes « même squelette »

En plus des inégalités, les schémas de type de Flow Caml peuvent faire intervenir une autre sorte de contraintes appelée contraintes de *squelette*. L'expérience montre toutefois que ces contraintes apparaissent rarement dans les types inférés et les interfaces de modules, car elles correspondent généralement à des cas pathologiques de fonctions effectuant des calculs sur leurs arguments, inutiles pour le résultat final. Considérons par exemple le fragment de code suivant :

```
let skel x y =
  let _ = [x; y] in
  x
;;
val skel : 'a -> 'b -> 'a
           with 'a ~ 'b
```

`skel x y` crée une liste d'éléments `x` et `y`, puis l'abandonne pour rendre le résultat `x`. Cette fonction est relativement artificielle, puisqu'elle pourrait être réécrite en

```
let skel x y =
  x
;;
```

mais suffit pour illustrer la nécessité des contraintes de squelette. En effet, pour que la liste `[x; y]` puisse être construite, il faut que les types des arguments `x` et `y` aient un super-type commun. En Objective Caml, ils seraient contraints d'avoir le *même* type, mais ici, certaines de leurs annotations de sécurité peuvent différer : par exemple, `x` peut avoir le type `!alice int` et `!bob int`, puisque `[> !alice, !bob] int` est un super-type de ces deux types. De manière générale, si `x` et `y` ont respectivement les types `'a` et `'b`, il est suffisant de requérir l'existence d'un type `'c` tel que `'a < 'c` et `'b < 'c` pour que la fonction `skel` soit bien typée. C'est précisément la sémantique de la contrainte `'a ~ 'b`, de telle sorte que le schéma donné ci-dessus est en fait équivalent à

```
val skel : 'a -> 'b -> 'a
           with 'a < 'c
           and  'b < 'c
```

Il est facile de vérifier qu'un tel type `'c` existe si et seulement si les types `'a` et `'b` ont la même forme et ne diffèrent que par des annotations qui apparaissent en position *non* invariante. La relation `~` est réflexive, transitive et associative (il s'agit en fait de la clôture symétrique et transitive de `<`), de telle sorte que les contraintes de squelette qui ont une variable commune peut être fusionnées, comme dans l'exemple suivant :

```
let skel3 x y z =
  let _ = [x; y; z] in
  x
;;
val skel3 : 'a -> 'b -> 'c -> 'a
            with 'a ~ 'b ~ 'c
```

Pour terminer cette description des différentes formes de contraintes que fait intervenir le système Flow Caml, la figure 2.1 précise la correspondance entre la représentation textuelle qui est

Syntaxe Flow Caml	Notations mathématiques
`'a < 'b`	$\alpha \leq \beta$
`'a < level('b)`	$\alpha \lhd \beta$
`content('a) < 'b`	$\alpha \blacktriangleleft \beta$
`'a ~ 'b`	$\alpha \approx \beta$

Figure 2.1 – Correspondance entre la syntaxe de Flow Caml et les notations mathématiques

donnée par le système et les notations mathématiques utilisées dans le reste du mémoire.

2.2.5 Valeurs fonctionnelles

Flow Caml est un langage fonctionnel : les fonctions sont des entités de première classe qui peuvent être manipulées comme toutes les autres valeurs. On peut ainsi définir une fonction dont le résultat lui-même est une fonction :

```
let pred x = x + 1;;
val pred : 'a int -> 'a int
let pred_or_succ y = if y then pred else succ;;
val pred_or_succ : 'a bool -> 'b int -{|| 'a}-> 'b int
```

Les schémas de type inférés peuvent être parenthésés comme suit :

```
val pred_or_succ : 'a bool -> ('b int -{|| 'a}-> 'b int)
```

Cependant, cela n'est bien entendu pas nécessaire puisque la flèche est associative à droite dans les types. Savoir quelle fonction parmi `pred` et `succ` une application de `pred_or_succ` retourne donne de l'information sur le Booléen passé en argument. Pour refléter ce flot d'information, le type assigné à la fonction retournée par `pred_or_succ` comporte un niveau d'information additionnel, `'a`, imprimé à l'intérieur du symbole flèche : son but est de décrire la quotité d'information attachée à la connaissance de la fonction elle-même. Par exemple, une application de `pred_or_succ` avec un Booléen de niveau `!alice` produit une fonction dont l'identité a le niveau `!alice` :

```
pred_or_succ y1;;
- : 'a int -{|| !alice}-> 'a int
```

Le langage Caml permet d'observer l'identité d'une fonction *via* les résultats produits par son application à différents arguments. Par exemple, quand `pred_or_succ y1` est appliquée à un entier, le résultat produit doit être gardé par le niveau `!alice`, car il permet de déterminer si la fonction `(pred_or_succ y1)` est `pred` ou `succ` puis d'en déduire le Booléen `y1`.

```
(pred_or_succ y1) 0;;
- : !alice int
(pred_or_succ y1) x2;;
- : [> !alice, !bob] int
```

En fait, les types flèches de Flow Caml font intervenir trois annotations ; de telle sorte que la forme générale d'un type de fonction est

```
'a -{'b | 'c | 'd}-> 'e
```

où `'a` et `'e` sont respectivement les types de l'argument attendu par la fonction et du résultat qu'elle produit. De plus, `'b` et `'d` sont des niveaux. Le premier est une borne inférieure sur les effets de bord produits par la fonction — il sera introduit à la section 3.1 (page 53) — et le second représente la quotité d'information attachée à l'identité de la fonction, comme expliqué ci-avant. Enfin, `'c` est une *rangée* décrivant les exceptions que la fonction est susceptible de lever — je détaillerai son usage à la section 3.2 (page 56).

Dans le but d'améliorer la lisibilité des types inférés, Flow Caml n'affiche pas les annotations sur les flèches qui ne véhiculent pas d'information, c'est-à-dire qui sont des variables universellement

quantifiées et non contraintes. Par exemple, `'a -> 'b` est une abréviation pour `'a -{'c | 'd | 'e}-> 'b` (où `'c`, `'d` et `'e` sont des variables fraîches), tandis que `'a -{|| 'b}-> 'c` doit être lu `'a -{'d | 'e | 'b}-> 'c` (où `'d` et `'e` sont des variables fraîches).

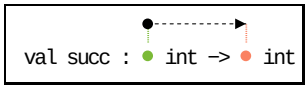
2.2.6 Interlude : affichage graphique des schémas de types

La boucle d'interaction de Flow Caml peut donner une représentation graphique des schémas de type, en plus de leur impression textuelle habituelle. La sortie graphique peut être activée en invoquant la commande `flowcaml` avec l'option `-graph` ou en entrant la directive

```
#open_graph;;
```

La représentation graphique d'un schéma de type peut être plus facile à interpréter que sa contrepartie textuelle, puisqu'elle donne une description visuelle des flots d'information sous forme de flèches. Dans les paragraphes suivants, j'explique succinctement au travers de quelques exemples comment lire ces tracés.

```
let succ x = x + 1;;
val succ : 'a int -> 'a int
```

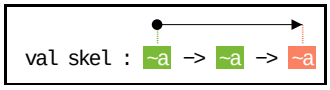


La représentation graphique d'un schéma de type est constituée de deux parties : en bas apparaît le *squelette* du schéma, qui consiste d'une expression de type où les annotations (niveaux et rangées) sont remplacées par des points •. En ignorant ces dernières, le squelette d'un schéma Flow Caml peut être lu à peu près comme un type d'Objective Caml. Un code de couleur est adopté pour afficher les points correspondant aux annotations : les variables en position contravariante apparaissent en vert, tandis que celles qui sont covariantes sont rouges et celles invariantes en orange. Les contraintes de sous-typage sont représentées dans la partie supérieure du tracé par des flèches. Dans la représentation du schéma de `succ`, la flèche pointillée du point vert vers le rouge indique une inéquation dont les membres gauche et droit sont respectivement les variables de niveaux symbolisées par ces deux points. Ainsi, la représentation graphique du schéma doit être lue comme le schéma suivant :

```
val succ : 'a int -> 'b int
           with 'a < 'b
```

qui est équivalent à `'a int -> 'a int`. Montrons maintenant comment les variables de type sont représentées graphiquement.

```
let skel x y =
  let _ = [x; y] in
  x
;;
val skel : 'a -> 'b -> 'a
           with 'a ~ 'b
```



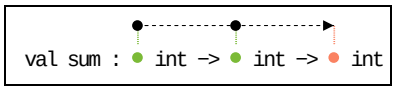
Dans le corps de la représentation du schéma de `skel`, les boîtes étiquetées `~a` représentent des variables de types : chaque occurrence de `~a` doit être lue comme une variable de type distincte $\mathtt{a}_1$, …, $\mathtt{a}_n$, avec la contrainte $\mathtt{a}_1 \sim \mathtt{a}_2 \sim \ldots \sim \mathtt{a}_n$. Par exemple, `~a -> ~a -> ~a` représente :

$$\mathtt{'a}_1 \texttt{ -> } \mathtt{'a}_2 \texttt{ -> } \mathtt{'a}_3 \quad \texttt{with } \mathtt{'a}_1 \sim \mathtt{'a}_2 \sim \mathtt{'a}_3$$

Le même code de couleur est employé pour les boîtes que pour les points. Les contraintes de sous-typage entre variables de type sont représentées par des flèches pleines : dans le schéma de `skel`, la flèche de la première boîte vers la troisième symbolise la contrainte $\mathtt{'a}_1 < \mathtt{'a}_3$.

Les origines et les têtes des flèches peuvent être partagées, comme dans les exemples suivants. Je donne pour chacun le schéma inféré par le système, sa représentation graphique, ainsi qu'un deuxième schéma équivalent au premier mais dont la lecture est plus proche de celle de la représentation graphique.

```
let sum x y = x + y;;
val sum : 'a int -> 'a int -> 'a int
val sum : 'a1 int -> 'a2 int -> a3 int
          with a1, a2 < a3
```



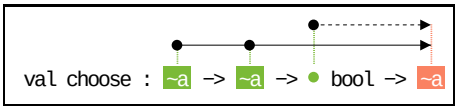

```
let make_pair x = (x, x);;
val make_pair : 'a -> 'a * 'a
val make_pair : 'a1 -> 'a2 * 'a3
                with 'a1 < 'a2, 'a3
```

Il reste à montrer comment les inégalités des formes **content**('a) < 'b, 'a < **level**('b) et **content**('a) < **level**('b) sont tracées. Toutes ces contraintes sont représentées par une flèche en pointillés de la boîte ou du point qui représente 'a à celle ou celui de 'b. Aucune confusion ne peut intervenir sur la nature de la contrainte, comme montré par la table suivante :

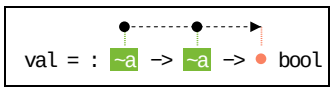
sorte de 'a	sorte de 'b	sémantique d'une flèche pointillée de 'a vers 'b
level	**level**	'a < 'b
level	**type**	'a < **level**('b)
type	**level**	**content**('a) < 'b
type	**type**	**content**('a) < **level**('b)

Cela est illustré dans les deux exemples suivants :

```
let choose y1 y0 x =
  if x then y1 else y0
;;
val choose : 'a -> 'a -> 'b bool -> 'a
             with 'b < level('a)
```




```
( = );;
val ( = ) : 'a -> 'a -> 'b bool
            with content('a) < 'b
```



Dans le schéma de type pour choose, la flèche en pointillés représente la contrainte 'b < **level**('a), tandis que dans celui pour (=), **content**('a) < 'b.

CHAPITRE 3 TROIS

Traits impératifs

Tous les exemples donnés dans le chapitre 2 (page 35) sont écrits dans un style « purement fonctionnel ». Le langage Flow Caml est cependant muni de toutes les traits impératifs habituels, comme les valeurs mutables dont le contenu peut être modifié en place et les exceptions. L'utilisation de ces constructions nécessite la prise en compte d'annotations supplémentaires dans le système de type. Je les présente dans ce chapitre.

3.1 Valeurs mutables

3.1.1 Flots d'information directs et indirects

Dans un langage de programmation autorisant des effets de bord, il est possible de transmettre de l'information de manière *indirecte* en observant la réalisation d'un effet. Pour illustrer ce phénomène, considérons les phrases suivantes :

```
r := not y
r := if y then false else true
if y then r := false else r := true
r := true; if y then r := false
```

Toutes ces expressions ont la même sémantique : elles modifient le contenu de la référence `r` en lui affectant la négation du Booléen `y`. Cela induit un flot d'information de `y` vers `r`. Cependant, la nature de ce flot est différente suivant les cas. Dans les deux premiers exemples, le flot est dit *direct* : une valeur dépendant de `y` est calculée puis stockée dans `r`. Ce phénomène est très similaire à ce que nous avons rencontré jusqu'à présent. Au contraire, dans les deux derniers exemples, aucune des valeurs apparaissant à droite d'un opérateur `:=` ne fait intervenir `y`, puisqu'elles sont données explicitement dans le code source. Cependant, le contenu de la référence peut à chaque fois être modifié dans une branche du programme dont l'évaluation est conditionnée par la valeur de `y`. C'est pourquoi le contenu de `r` est susceptible, à la fin de l'exécution, de porter de l'information sur `y`. Dans cette situation, on parle d'un flot *indirect* de `y` vers `r`. Le dernier des quatre exemples appelle un commentaire additionnel : la référence `r` n'est jamais modifiée sous une condition dépendant de `y` dans le cas où ce Booléen est `false`. Cependant, le flot d'information existe toujours : il est en effet possible de faire fuir de l'information à travers l'*absence* d'un certain effet. Ceci illustre la difficulté de détecter les flots d'information illégaux à l'exécution, et plaide donc en faveur d'une

analyse statique.

3.1.2 Références

Comme celles de Caml, les références de Flow Caml sont un cas particulier d'enregistrements à champ mutable (section 4.1.2, page 68), lequel est déclaré dans le module `Pervasives`. Il me semble cependant plus simple, dans un premier temps, d'expliquer comment elles sont typées indépendamment de cette définition, en utilisant les trois opérations prédéfinies `ref`, `:=` et `!`, qui permettent respectivement de créer une nouvelle référence, de modifier son contenu et de le lire. En Flow Caml, le constructeur de type pour les références, `ref`, a deux arguments :

```
#lookup_type "ref";;
type (='a:type, #'b:level) ref = ...
```

Le premier est le type de la valeur stockée dans la référence. Puisque le contenu d'une référence est accessible à la fois en lecture et en écriture, il doit être à la fois co- et contravariant, il est donc *invariant*. Le deuxième argument de `ref` est un niveau, qui est gardé et covariant. Il décrit la quotité d'information attachée à l'*identité* de la référence, c'est-à-dire à son adresse mémoire.

Définissons deux références `r1` et `r2` :

```
let r1 : (!alice bool, 'a) ref = ref true;;
val r1 : (!alice bool, 'a) ref
let r2 : (!bob bool, 'a) ref = ref true;;
val r2 : (!bob bool, 'a) ref
```

La référence `r1` a un contenu de type `!alice bool`. Cela signifie qu'il peut recevoir tout Booléen dont le niveau est inférieur ou égal à `!alice`, *i.e.* un Booléen que *Alice* est autorisée à consulter. Par exemple, la constante `y1`, définie à la section 2.1 (page 35), vérifie cette condition. Ainsi, elle peut légalement être affectée comme contenu de `r1` :

```
r1 := y1;;
- : unit
```

Cette expression ne produit qu'un effet de bord, de telle sorte qu'elle a le type `unit`. Puisque ce type n'est habité que par une seule valeur, la constante `()`, le résultat d'une expression de type `unit` ne peut être le porteur d'aucune information. Par conséquent, le constructeur de type `unit` n'est pas annoté. Le Booléen `y2` a été déclaré de niveau `!bob`. Puisque le transfert d'information de `!bob` vers `!alice` n'est pas autorisé (section 4.2, page 70), le système de type refuse son affectation à `r1` :

```
r1 := y2;;
This expression generates the following information flow:
  from !bob to !alice
which is not legal.
```

De manière similaire, la référence `r1` peut être modifiée dans un contexte dont l'exécution dépend de `y1`, mais pas de `y2` :

```
if y1 then r1 := false else r1 := true;;
- : unit
if y2 then r1 := false else r1 := true;;
This expression generates the following information flow:
  from !bob to !alice
which is not legal.
```

Enfin, la lecture du contenu de `r1` produit comme attendu un Booléen de niveau `!alice` :

```
!r1;;
- : !alice bool
```

Comment le système de type est-il capable de détecter les flots d'information décrits dans les exemples précédents ? Flow Caml associe à chaque point d'un programme (*i.e.* à chaque contexte d'évaluation) un niveau décrivant la quotité d'information que la sous-expression correspondante

acquiert lorsqu'elle est exécutée. Dans la littérature, ce niveau est usuellement noté *pc*, en référence à la notion de *compteur de programme* (*program counter* en anglais). Sans entrer dans les détails, on peut dire qu'à chaque fois qu'une construction conditionnelle est traversée, ce niveau est augmenté de l'annotation portée par la condition, comme illustré ci-après :

```
if y1 (* y1 a le type !alice bool *) then
  ... (* cette branche est typée sous le niveau !alice *)
else
  if y2 (* y2 a le type !bob bool *) then
    ... (* cette branche est typée sous le niveau [> !alice, !bob] *)
  else
    ... (* cette branche est typée sous le niveau [> !alice, !bob] *)
```

De plus, lorsque des données sont écrites dans une référence, le système contraint le niveau attaché à son contenu à être supérieur ou égal à celui associé au contexte d'évaluation. Cela reflète que, du fait de sa modification, le contenu de la référence est susceptible de donner de l'information à propos de chacun des tests effectués pour atteindre ce point du programme.

La définition de fonctions qui effectuent des effets de bords apporte une difficulté supplémentaire : puisque le corps de la fonction est exécuté au point du programme où elle est appelée — et non à celui où elle est définie — elle doit être typée à un niveau qui est *a priori* différent de celui où elle apparaît dans le code source. C'est la raison de l'existence de la première annotation des types flèches (section 2.2.5, page 49). Par exemple, considérons une fonction qui écrit le Booléen `false` dans la référence `r1` :

```
let reset_r1 () =
  r1 := false
;;
val reset_r1 : unit -{!alice ||}-> unit
```

De même que le contenu de `r1` ne peut être modifié que dans un contexte de niveau inférieur ou égal à `!alice`, de même la fonction `reset_r1` ne peut être appelée que dans un contexte dont le niveau est inférieur ou égal à `!alice`. Cela est reflété par l'annotation `!alice` qui apparaît à l'intérieur de la flèche : ce niveau donne une borne inférieure sur les effets réalisés par la fonction. Il constitue donc également une borne supérieure sur les contextes d'où la fonction peut être appelée.

Le rôle du deuxième argument du constructeur de type `ref` apparaît lorsqu'une référence est utilisée comme valeur de première classe, *e.g.* comme résultat d'une fonction. À titre d'illustration, je considère une version de la fonction `choose` spécialisée aux références :

```
let choose_ref y r1 r0 : (_, _) ref =
  if y then r1 else r0
;;
val choose_ref : 'a bool ->
                 ('b, 'a) ref -> ('b, 'a) ref -> ('b, 'a) ref
```

Le niveau de la référence retournée par cette fonction doit être supérieur ou égal à celui du Booléen donné comme argument. En effet, déterminer quelle référence est retournée par la fonction peut révéler de l'information sur la valeur de la condition `y`. Une telle observation peut être effectuée en modifiant le contenu de la référence, en utilisant par exemple la fonction suivante :

```
let reset r =
  r := false
;;
val reset : ('a bool, 'a) ref -{'a ||}-> unit
```

La fonction `reset` prend une référence comme argument, et rend son contenu égal à `false`. Le système de type contraint le niveau du contenu de la référence à être supérieur ou égal au niveau attaché à l'identité de la référence *et* à celui correspondant au contexte où la fonction est appliquée.

Pour terminer ce rapide tour d'horizon du typage des références en Flow Caml, je donne une nouvelle écriture de la fonction calculant la longueur d'une liste, `length`, en adoptant un « style

impératif » :

```
let length' list =
  let counter = ref 0 in
  let rec loop = function
      [] -> ()
    | _ :: tl ->
      incr counter;
      loop tl
  in
  loop list;
  !counter
;;
val length' : ('a, 'b) list -{'b ||}-> 'b int
```

Comme pour `length`, le niveau de l'entier retourné par cette fonction doit être supérieur ou égal à celui de la liste passée en argument. Cependant, le nouveau schéma de type porte une contrainte supplémentaire : avec `length'`, le niveau d'information du résultat doit être supérieur ou égal à celui du contexte dans lequel la fonction est appelée. Il s'agit d'un problème de *sur-spécification* : le type obtenu pour cette fonction donne trop d'indications sur la manière dont elle est implantée. Cependant, cette différence est seulement superficielle : on peut vérifier que les deux types ont en fait exactement le même pouvoir expressif. Effectuer cette identification de manière automatique reste un travail à réaliser.

3.1.3 Tableaux, chaînes de caractères et boucles

Je conclue cette section par quelques mots sur les tableaux et les chaînes de caractères mutables. En Flow Caml, le constructeur de type pour les tableaux, `array`, porte deux paramètres :

```
[|0; 1; 2|];;
- : ('a int, 'b) array
```

Leurs rôles respectifs sont analogues à ceux des paramètres de `ref` : le premier donne le type des éléments contenus dans le table, le second est un niveau relié à l'identité du tableau. Une légère nouveauté réside dans le fait qu'il décrit également l'information attachée à la longueur du tableau, comme le montre le type de la fonction `Array.length` :

```
Array.length;;
- : ('a, 'b) array -> 'b int
```

qui est comparable à celui de la fonction `length` sur les listes.

Dans le langage Caml, la seule différence entre les chaînes de caractères mutables et les tableaux de caractères réside dans leur représentation à l'exécution. Par conséquent, les types correspondant à ces deux structures de données en Flow Caml sont isomorphes : le type `('a, 'b) charray` doit ainsi être lu de la même façon que `('a char, 'b) array`.

```
[|'a'; 'b'; 'c'|];;
- : ('a char, 'b) array
"abc";;
- : ('a, 'b) charray
```

3.2 Exceptions

Les exceptions constituent pour le programmeur un outil puissant pour signaler et traiter les situations exceptionnelles. Comme en Objective Caml, les *noms d'exceptions* sont déclarés en utilisant le mot-clef **`exception`**, et les exceptions levées par l'opérateur **`raise`** :

```
exception X;;
exception X
exception Y;;
```

```
exception Y
raise X;;
- : 'a
```

Le mécanisme d'exception fourni par Flow Caml est toutefois légèrement restreint par rapport à celui d'Objective Caml : les exceptions ne sont pas des entités de première classe du langage Flow Caml. Autrement dit, un nom d'exception (comme `X` dans l'exemple précédent) n'est pas une valeur, et ne peut donc être lié à un identificateur ou passé comme argument à une fonction — tandis qu'en Objective Caml, il s'agit d'une valeur normale de type `exn`. De même, en Objective Caml, **`raise`** est (pour ce qui concerne l'utilisateur du langage) une fonction normale, alors que, en Flow Caml, il s'agit d'un mot-clef qui requiert que le nom de l'exception à lever soit explicitement donné. Par exemple, l'expression suivante, valide en Objective Caml, ne peut pas être écrite en Flow Caml :

```
let f x =
  raise (if x then X else Y)
;;
```

Dans ce cas particulier, elle peut toutefois être ré-écrite comme suit :

```
let f x =
  if x then raise X else raise Y
;;
```

Bien que possible d'un point de vue théorique, la présence d'exceptions de première classe dans le langage Flow Caml nécessiterait un système de types plus complexe, faisant en particulier intervenir des contraintes conditionnelles (section 9.2, page 160). L'expérience montre que l'usage des exceptions comme valeurs dans les programmes écrits en Caml est plutôt restreint, et généralement limité à certains idiomes bien particuliers. Ce sont les raisons pour lesquelles il m'a semblé judicieux de se limiter à des exceptions de deuxième classe, tout en fournissant deux constructions supplémentaires, **`finally`** et **`propagate`** (section 3.2.3, page 61), qui compensent partiellement la perte de puissance sur la forme **`try`** ... **`with`**. Je crois qu'il s'agit d'un bon compromis entre l'expressivité du langage et la complexité du système de types.

3.2.1 Rangées

Les exceptions qu'une expression est susceptible de lever sont décrites, dans le système de type de Flow Caml, en utilisant des *rangées*. Une rangée est une fonction des noms d'exceptions vers les niveaux d'information : pour chaque nom d'exception, la rangée donne la quotité d'information qui est transmise si l'expression lève effectivement une exception de ce nom. Puisque l'ensemble des noms d'exception est ouvert (dans le sens où le programmeur peut en définir un nombre arbitraire de manière incrémentale), les rangées doivent parcourir un domaine potentiellement infini. Pour représenter ces objets dans la syntaxe concrète du langage, Flow Caml fait intervenir des *variables de rangées* et utilise la syntaxe des rangées de Rémy [Rém90]. Par exemple, `X: 'a; Y: 'b; 'c` dénote la rangée qui associe le niveau `'a` à l'exception `X`, le niveau `'b` à `Y` et dont la valeur pour les autres exceptions est donnée par `'c`. Ici, `'a` et `'b` sont des variables de sorte niveau, tandis que `'c` est une variable de rangée dont le domaine est le complémentaire de `{X, Y}` : elle représente en effet une rangée définie sur tous les noms d'exceptions sauf `X` et `Y`. L'ordre dans lequel les champs d'une rangée apparaît n'est pas significatif : par exemple, la rangée précédente est égale à `Y: 'b; X: 'a; 'c`. Les variables de rangées peuvent apparaître dans les inégalités : l'ordre de sous-typage est étendu sur les rangées point à point, de manière covariante. Ainsi, si `'c1` et `'c2` sont deux rangées de même domaine, la contrainte `'c1 < 'c2` est valide si et seulement si chaque entrée de `'c1` est inférieure ou égale à celle qui lui correspond dans `'c2`. Ainsi, les inégalités reliant deux expressions de rangées mentionnant des noms d'exception identiques peuvent être décomposées : par exemple

```
X: 'a1; Y: 'b1; 'c1 < X: 'a2; Y: 'b2; 'c2
```

est équivalent à

```
'a1 < 'a2 and 'b1 < 'b2 and 'c1 < 'c2
```

Enfin, dans le but d'obtenir des types aussi concis que possible, Flow Caml n'affiche pas, dans les schémas de type, les variables de rangées qui ne sont pas contraintes : par exemple, `A: 'a; Y: 'b` est une abréviation pour `X: 'a; Y: 'b; 'c` où `'c` est une variable de rangée fraîche.

Puisque les exceptions constituent une forme observable de résultat pour les fonctions, elles doivent être prises en compte dans les types de ces dernières. Définissons par exemple une fonction triviale qui lève l'exception `X` :

```
let raise_X () =
  raise X
;;
val raise_X : unit -{'a | X: 'a |}-> 'b
```

La deuxième annotation qui apparaît sur la flèche est, comme annoncé section 2.2.5 (page 49), une rangée qui décrit les exceptions que la fonction est susceptible de lever lorsqu'elle est appelée. Ici, `X : 'a` indique que la fonction `raise_X` peut déclencher une exception de nom `X`, et que l'information obtenue en rattrapant cette dernière à un niveau borné par `'a`. En effet, cette exception fait fuir de l'information sur le contexte dans lequel la fonction `raise_X` est appelée, lequel est également de niveau `'a`. Dans l'exemple suivant :

```
let raise_X' y =
  if y then raise X
;;
val raise_X' : 'a bool -{'a | X: 'a |}-> unit
```

le rattrapage de l'exception `X` peut donner de l'information à la fois sur le contexte où `raise_X'` est appelé *et* sur le Booléen donné comme argument à la fonction. Les deux niveaux correspondants doivent donc être inférieurs ou égaux à celui associé à l'exception `X`.

Quand une fonction est susceptible de lever des exceptions de différents noms, sa rangée comporte une entrée pour chacun d'entre eux :

```
let raise_X_or_Y x y =
  if x then raise X;
  if y then raise Y
;;
val raise_X_or_y : 'a bool -> 'b bool -{'a | X: 'a; Y: 'b |}-> unit
                   with 'a < 'b
```

Le schéma de type inféré par le système distingue un niveau d'information pour chaque nom d'exception : rattraper `X` donne de l'information sur le premier argument de la fonction, `x`, tandis que rattraper `Y` donne de l'information sur les deux arguments, `x` et `y`.

Pour continuer, considérons une fonction qui prend pour argument un entier, lève `X` s'il est égal à zéro ou, sinon, retourne `false` :

```
let test_zero x =
  if x = 0 then raise X;
  false
;;
val test_zero: 'a int -> {'a | X: 'a |}-> 'b bool
```

D'après ce schéma de type, le résultat booléen de `test_zero` ne dépend pas de l'argument passé : en effet, si la fonction produit effectivement une valeur, alors il s'agit nécessairement de la constante `false`. Cependant, cette fonction peut faire fuir de l'information *via* ses résultats exceptionnels, ce qui se traduit dans son type par le niveau associé à l'exception `X` : ce dernier doit être supérieur ou égal à ceux du contexte d'appel et de l'entier passé en argument.

Les exceptions peuvent être rattrapées, comme en Objective Caml, à l'aide de la construction **try** ... **with** :

```
try
  test_zero x1
with
  X -> true
;;
- : !alice bool
```

Cet exemple montre que le résultat d'une construction **try** ... **with** doit être gardé par le niveau de chaque exception qu'elle peut rattraper : ici, la sous-expression `test_zero x1` peut lever l'exception `X` avec le niveau `!alice` qui est immédiatement rattrapée. Le résultat booléen doit donc aussi avoir le niveau `!alice`.

La deuxième partie d'une expression **try** ... **with** est une sorte de filtrage sur les noms d'exceptions. (Il ne s'agit toutefois pas d'un filtrage de la même classe syntaxique que celui introduit par **match** ... **with** puisque les exceptions ne sont pas des valeurs.) En particulier, une construction **try** ... **with** peut rattraper plusieurs noms d'exceptions — ou même tous les noms d'exceptions grâce au motif _ — comme illustré dans l'exemple suivant :

```
let f7 x y =
  try
    raise_X_or_Y x y;
    0
  with
    X -> 1
  | Y -> 2
;;
val f7 : 'a bool -> 'a bool -{'a ||}-> 'a int
```

De nombreuses fonctions de la bibliothèque standard lèvent une exception quand elles ne peuvent terminer normalement. Par exemple, l'opération de division entière produit l'exception `Division_by_zero` quand son deuxième argument est l'entier nul, comme le reflète son type :

```
val ( / ) : 'a int -> 'b int -{'c | Division_by_zero: 'c |}-> 'a int
            with 'b < 'c, 'a
```

Il est remarquable qu'il n'y ait pas de relation entre le niveau du premier argument et celui de l'exception `Division_by_zero`. Cela reflète le fait que l'implémentation de la division n'a pas besoin d'examiner le premier argument avant de déterminer si l'exception doit être levée.

Pour obtenir une analyse précise des flots d'information, l'ordre d'évaluation des expressions doit faire partie de la définition du langage Flow Caml. La spécification d'Objective Caml ne précise pas cet ordre, cependant, son (unique) implémentation actuelle effectue l'évaluation de droite à gauche. C'est naturellement cette stratégie que j'ai retenue pour Flow Caml. Dans l'exemple suivant, le système de type tient compte du fait que le deuxième argument de `+` est évalué avant le premier :

```
let f8 x y =
  (if x = 0 then raise X else x) + (if y = 0 then raise Y else y)
;;
val f8 : 'a int -> 'b int -{'c | X: 'd; Y: 'e |}-> 'f int
         with 'b < 'd, 'e, 'f
         and  'c < 'd, 'e
         and  'a < 'd, 'f
```

Le schéma de type obtenu montre en effet que la condition `y = 0` est considérée avant `x = 0`, de telle sorte que si l'exception `X` est levée, celle-ci est porte de l'information sur `x` et `y`. À l'inverse, l'exception `Y` n'est reliée qu'à la valeur de `y`. En supposant un ordre d'évaluation de gauche à droite, on obtiendrait le typage suivant, où les rôles des variables `'a` et `'b` sont échangés :

```
'a int -> 'b int -{'c | X: 'd; Y: 'e |}-> 'f int
with 'b < 'd, 'f
and  'c < 'd, 'e
and  'a < 'd, 'e, 'f
```

Enfin, si l'ordre d'évaluation n'était pas spécifié, le système de type devrait prendre en compte toutes les stratégies possibles. Dans ce cas, le schéma de type pour `f8` serait moins précis :

```
'a int -> 'a int -{'b | X: 'b; Y: 'b |}-> 'a int
with 'a < 'b
```

En effet, les niveaux d'information associés aux deux exceptions ne sont plus différenciés, et doivent tous deux être supérieurs ou égaux à ceux des deux arguments de la fonction. Je reviendrai sur ce point dans la deuxième partie de la thèse, à la section 9.1 (page 159).

3.2.2 Exceptions et effets de bord

La levée d'une exception peut également être observée par la non-réalisation d'un effet de bord ultérieur. Pour illustrer ce phénomène, considérons la fonction suivante :

```
let f9 y r =
  try
    if y then raise X;
    r := 0
  with
    X -> ()
;;
val f9 : 'a bool -> ('a int, 'a) ref -{'a ||}-> unit
```

Cette fonction affecte le contenu de la référence `r` à `0` si le Booléen `y` est faux. Elle pourrait naturellement être écrite de manière directe :

```
let f9' y r =
  if not y then r := 0
;;
val f9' : 'a bool -> ('a int, 'a) ref -{'a ||}-> unit
```

ce qui donnerait exactement le même type : puisque la référence `r` est modifiée sous une condition portant sur `y`, le niveau de son contenu doit être supérieur ou égal à celui du Booléen. La première écriture de la fonction est cependant intéressante pour expliquer comment le flot d'information entre le Booléen `y` et le contenu de la référence `r`, engendré par la combinaison d'une exception et d'un effet de bord, est détecté. En effet, dans la fonction `f9`, la référence `r` n'est pas modifiée sous une condition dépendant de `y`. Cependant, l'instruction `r := 0` apparaît de manière séquentielle *après* une expression susceptible de lever l'exception `X` : en particulier, quand elle est exécutée, elle observe que l'exception `X` n'a pas été levée. Elle doit donc être typée à un niveau augmenté de celui auquel l'exception `X` a été levée, c'est-à-dire celui du Booléen `y`. De manière générale, tout fragment de programme situé entre un point de déclenchement potentiel d'une exception et le rattrapage de celle-ci doit être typé à un niveau augmenté du niveau du premier point.

```
let f10 y1 y2 r1 r2 =
  try
    begin try
      if y1 then raise X;
      if y2 then raise Y;
      r1 := 0;
    with
      Y -> ()
    end;
    r2 := 0
  with X -> ()
;;
val f10 : 'a bool ->
          'b bool -> ('b int, 'b) ref -> ('c int, 'c) ref -{'a ||}-> unit
          with 'a < 'b, 'c
```

Dans cette deuxième fonction, les exceptions `X` et `Y` sont respectivement levées dans des contextes de niveaux `'a` et `'b`. La référence `r1` est modifiée à un point du programme situé entre celui où `X` et `Y` sont levées et ceux où elles sont rattrapées : son contenu doit donc avoir un niveau supérieur ou égal à celui des deux exceptions, c'est-à-dire `'b` (puisque `'a < 'b`). À l'inverse, `r2` est modifiée après le rattrapage de l'exception `Y`, de telle sorte que son contenu n'est gardé que par le niveau de `X`, *i.e.* a le niveau `'a`.

Considérons maintenant un exemple de fonctionnelle d'ordre supérieur, c'est-à-dire une fonction prenant pour argument une autre fonction :

```
let rec iter f = function
    [] -> ()
  | hd :: tl -> f hd; iter f tl
;;
val iter : ('a -{'b | 'c | 'b}-> 'd) -> ('a, 'b) list -{'b | 'c |}-> unit
           with content('c) < 'b
```

Il s'agit de la fonction `List.iter` de la bibliothèque standard. Elle prend deux arguments, une fonction `f` et une liste, puis applique la fonction `f` à chacun des éléments de la liste. Puisque l'argument `f` peut avoir des effets arbitraires et être issue d'un calcul, les trois annotations du type flèche correspondant doivent apparaître dans le type de `iter`. La variable de rangée `'c` décrit les exceptions potentiellement levées par `f`. Puisque `iter` ne lève pas d'exception d'elle-même, `'c` est également la rangée portée par sa flèche. Le niveau `'b` décrit l'information transmise à `f` par ses contextes d'appel : il doit naturellement dominer le niveau du contexte dans lequel `iter` est appelée (puisque `f` est invoquée par `iter`), le niveau d'information de la liste (puisque, lorsqu'elle est exécutée, `f` apprend en particulier que la liste n'est pas vide). La contrainte **`content('c) < 'b`** assure que ce niveau est supérieur ou égal à chacune des entrées de la rangée `'c` : à chaque fois qu'elle est appelée, la fonction `f` reçoit comme information que ses invocations précédentes ont terminé.

3.2.3 Les constructions `propagate` et `finally`

En plus de l'habituel **`try`** ... **`with`**, le langage Flow Caml est muni de deux autres constructions pour rattraper les exceptions. Leur introduction a été motivée par deux objectifs : tout d'abord, elles compensent partiellement la perte d'expressivité résultant du choix de faire des exceptions des entités de seconde classe ; elles permettent d'autre part un typage plus précis de certains idiomes courants.

Chacune des clauses qui apparaît dans la deuxième partie d'une construction **`try`** ... **`with`** peut se terminer par le mot-clef **`propagate`**. Dans ce cas, la clause lève à nouveau, à la fin de son exécution, l'exception qui l'a initiée. Par exemple :

```
try
  e
with
  X | Y -> e'; propagate
```

évalue `e`. Si cette expression lève `X` ou `Y` alors `e'` est exécutée et, une fois son évaluation terminée, l'exception rattrapée, `X` ou `Y`, est à nouveau levée. Cette construction est analogue au `reraise` qui était présent dans d'anciennes versions de Caml « Lourd » [WAL+90]. En Objective Caml, elle peut être implémentée en liant l'exception rattrapée à un identificateur :

```
try
  e
with
  X | Y as exn -> e'; raise exn
```

Outre le fait que ce codage n'est pas possible avec les exceptions de seconde classe de Flow Caml, la fourniture d'une construction spécifique permet un typage fin de cet idiome : l'exception est propagée par la clause avec exactement le niveau qu'elle avait dans `e`.

La construction **try** ... **finally** de Flow Caml est similaire au `unwind-protect` de Lisp ou au `finally` de Java. En effet

```
try
  e1
finally
  e2
```

évalue tout d'abord `e1`, de manière à obtenir soit une valeur soit une exception. Dans les deux cas, `e2` est évaluée. Son résultat est abandonné et celui de `e1` retourné. À nouveau, cette construction peut être codée en Caml comme suit :

```
try
  let r = e1 in e2; r
with
  exn -> e2; raise exn
```

(en supposant que `e2` ne lève pas d'exception, ce que le typage effectué par Flow Caml assure). Cependant, cette construction dédiée permet un meilleur typage de l'expression `e2` : cela rend en effet explicite le fait que l'expression `e2` est *toujours* exécutée, quelle que soit la nature du résultat produit par `e1`. Ainsi, cette dernière peut être typée dans un contexte de même niveau que celui de `e1` sans l'augmenter des niveaux des exceptions potentiellement levées. Par exemple, le fragment de code suivant est accepté par le système :

```
try
  if y1 then raise X
finally
  r2 := false
;;
```

tandis que son codage est rejeté :

```
try
  if y1 then raise X;
  r2 := false
with
  _ -> r2 := false; propagate
;;
This expression generates the following information flow:
  !alice < !bob
which is not legal.
```

3.2.4 Noms d'exceptions paramétrés

Dans le langage Caml, il est possible de déclarer des noms d'exceptions qui prennent un argument lorsqu'ils sont levés. Le type de l'argument est donné lors de la déclaration du nom d'exception. Il peut par exemple s'agir d'un entier représentant un code d'erreur :

```
exception Error of int;;
```

Une telle définition est également possible en Flow Caml. Cependant, le type des entiers est paramétré par un niveau d'information qui doit par conséquent être donné dans la déclaration de l'exception. Par exemple, on peut définir une exception `ErrorAlice` paramétrée par un entier de niveau `!alice` :

```
exception ErrorAlice of !alice int;;
```

Cette exception peut être levée avec n'importe quel entier dont le niveau est connu pour être inférieur ou égal à `!alice` :

```
raise (ErrorAlice 0);;
raise (ErrorAlice x1);;
```

Une telle déclaration peut toutefois ne pas être satisfaisante dans certains cas. Tout d'abord, lorsque l'exception est rattrapée, son argument est considéré de type `!alice int`, même si son niveau était en réalité inférieur :

```
try
  raise (ErrorAlice 0)
with
  ErrorAlice x -> x
;;
- : !alice int
```

De plus, si on souhaite signaler, à un autre point du programme, une erreur dont le code dépend d'un autre principal, par exemple Bob, il est nécessaire de définir un deuxième nom d'exception :

```
raise (ErrorAlice x2);;
This expression generates the following information flow:
  from !bob to !alice
which is not legal.

exception ErrorBob of !bob int
;;
raise (ErrorBob x2);;
```

Cela n'est pas tout-à-fait satisfaisant en pratique. C'est la raison pour laquelle j'ai autorisé, en Flow Caml, le type des arguments d'exceptions à être paramétrés par un niveau d'information :

```
exception Error : 'a of 'a int;;
```

Pour expliquer la manière dont ce niveau d'information est pris en compte dans les types, considérons la fonction suivante :

```
let error code =
  raise (Error code)
;;
val error: 'a int -{'a | Error: 'a |}-> 'b
```

Le niveau associé à l'exception `Error` dans cette fonction est la combinaison de deux informations : le niveau associé au contexte dans lequel l'exception est levé *et* celui de son argument entier. La fusion de ces deux annotations en une seule est relativement *ad hoc* ; cependant elle permet de conserver des types fonctionnels concis, et elle fonctionne bien en pratique. Il serait également possible de fournir un mécanisme plus complexe permettant de paramétrer les types d'exceptions par plusieurs niveaux d'information, qui apparaîtraient également dans les rangées, en plus des niveaux associés aux contextes. Cependant, cela pourrait accroître notoirement la verbosité des types de fonction.

Pour terminer cette section sur les exceptions, je souhaite mentionner que quelques une des exceptions prédéfinies du système Objective Caml ne sont pas accessibles en Flow Caml. C'est le cas par exemple de `Out_of_memory` et `Stack_overflow` qui sont respectivement levées par le glaneur de cellules quand la mémoire est saturée ou l'interpréteur de code lorsque la pile d'évaluation atteint sa taille maximale. En effet, analyser ces exceptions en Flow Caml aurait peu de sens, puisque, en l'absence d'une analyse sophistiquée de l'usage de la mémoire et de la pile, on devrait supposer qu'elles sont potentiellement levées en tout point du programme. Mon choix a plutôt été de rendre ces exceptions irrattrapables : leur déclenchement termine immédiatement l'exécution, de telles sortes qu'elles deviennent inobservables par le programme lui-même.

CHAPITRE 4 QUATRE

Définitions de types et de modules

4.1 Définition de types de données

En Flow Caml, le programmeur peut définir de nouvelles structures de données grâce aux *types de données algébriques*, variants et enregistrements. Comme en Objective Caml, ces types sont définis grâce aux déclarations introduites par le mot-clef **type**. La forme de celles-ci est très voisine de celle d'Objective Caml. Elles comportent toutefois quelques annotations supplémentaires, requises par l'analyse de flots d'information.

4.1.1 Variants

Je commence cette présentation des types de données avec quelques exemples de types variants. Voici un type à quatre variantes permettant de représenter les points cardinaux :

```
type 'a cardinal =
    North
  | West
  | South
  | East
  # 'a
;;
type (#'a:level) cardinal = North | West | South | East # 'a
```

Dans le système de type de Flow Caml, l'information portée par une valeur de type `cardinal` est décrite par un niveau — exactement comme pour les types énumérés prédéfinis tels que les entiers ou les caractères. Ce niveau est précisé par la clause `# 'a` qui termine la définition du type somme : il s'agit ici de l'unique paramètre du constructeur de type `cardinal`.

Lorsqu'un type est défini, la boucle interactive reproduit sa déclaration en précisant explicitement la signature du constructeur de type (cette dernière est automatiquement inféré par le système à partir de la déclaration entrée par le programmeur). Cette signature donne la sorte et la variance de chaque paramètre : dans l'exemple précédent, `#'a:level` signifie que `'a` est un paramètre de sorte **level**, covariant et gardé.

```
let p0 = North;;
val p0 : 'a cardinal
let p1 : !alice cardinal = North;;
```

```
val p1 : !alice cardinal
let p2 = if y2 then North else South;;
val p2 : !bob cardinal
```

Je définis maintenant une fonction `rotate` qui prend pour argument un point cardinal et retourne son successeur dans l'ordre des aiguilles d'une montre :

```
let rotate = function
    North -> East
  | West -> North
  | South -> West
  | East -> South
;;
val rotate: 'a cardinal -> 'a cardinal
```

Comme reflété par le schéma de type inféré, cette fonction prend un point cardinal de niveau quelconque, et retourne un autre point cardinal du même niveau.

Dans les sections 2.1.3 et 2.1.4, nous avons rencontré deux exemples de types variants, qui sont prédéfinis dans le langage Flow Caml : les listes et les options. Bien entendu, ces types n'ont rien de particulier (excepté le lieu de leur définition), et peuvent être définis comme suit :

```
type ('a, 'b) option =
    None
  | Some of 'a
  # 'b
;;
type (+'a:type, #'b:level) option = None | Some of 'a # 'b
```

La définition liste toutes les formes possibles d'une valeur de type `('a, 'b) option` : il s'agit soit de la constante `None` soit du constructeur `Some` avec un argument de type `'b`. La quatrième ligne de la déclaration, `# 'b`, indique que le paramètre `'b` est le niveau d'information associé à la connaissance de la forme de l'option, *i.e.* `None` ou `Some`.

La réponse produite par la boucle interactive lorsque cette déclaration est entrée donne à nouveau la signature du constructeur de type défini : `+'a:type` signifie que le premier paramètre de `option` est covariant et de sorte type ; tandis que `#'b:level` signifie que le second paramètre est un niveau, covariant et gardé.

La déclaration du type `list` est naturellement récursive, mais cela n'a pas d'incidence particulière. Elle est ainsi similaire à la précédente :

```
type ('a, 'b) list =
    []
  | :: of 'a * ('a, 'b) list
  # 'b
;;
type (+'a:type, #'b:level) list = [] | (::) of 'a * ('a, 'b) list # 'b
```

En suivant ce modèle, on peut également définir un type de données pour construire des arbres binaires étiquetés :

```
type ('a, 'b) tree =
    Leaf
  | Node of ('a, 'b) tree * 'a * ('a, 'b) tree
  # 'b
;;
type (+'a:type, #'b:level) tree =
    Leaf
  | Node of ('a, 'b) tree * 'a * ('a, 'b) tree
  # 'b
```

puis écrire une fonction qui calcule la hauteur d'un arbre :

```
let rec height = function
    Leaf -> 0
  | Node (tl, _, tr) -> max (height tl) (height tr)
;;
val height: ('a, 'b) tree -> 'b int
```

Comme pour la fonction qui calcule la longueur d'une liste, le schéma de type inféré indique que la hauteur d'un arbre dépend de sa structure (*i.e.* l'imbrication des constructeurs `Leaf` et `None`), mais pas des valeurs stockées dans les nœuds.

Munir une définition de type extraite d'un programme Objective Caml des annotations nécessaires pour le langage Flow Caml nécessite généralement de choisir entre plusieurs alternatives, en fonction des propriétés de sécurité que l'on souhaite vérifier. Par exemple, considérons la définition en Objective Caml d'un type pour représenter des arbres binaires étiquetés par des entiers :

```
type int_tree =
    ILeaf
  | INode of int_tree * int * int_tree
;;
```

Il y a au moins deux manières pertinentes de « décorer » cette déclaration pour la transposer en Flow Caml. Une première solution consiste à spécialiser la définition du type `tree` donnée ci-avant au cas des entiers :

```
type ('a, 'b) int_tree =
    ILeaf
  | INode of ('a, 'b) int_tree * 'a int * ('a, 'b) int_tree
  # 'b
;;
type (+'a:level, #'b:level) int_tree =
    ILeaf
  | INode of ('a, 'b) int_tree * 'a int * ('a, 'b) int_tree
  # 'b
;;
```

Le type `('a, 'b) int_tree` est isomorphe à `('a int, 'b) tree`. Ainsi, le type d'un arbre est annoté par deux niveaux, `'a` et `'b` : le premier décrit l'information attachée aux entiers stockés dans l'arbre, tandis que le second est relié à la structure de l'arbre. Pour illustrer l'intérêt de cette distinction, définissons deux fonctions : `size`, qui calcule le nombre de nœuds d'un arbre, et `sum`, qui effectue la somme de ses étiquettes :

```
let rec size = function
    ILeaf -> 0
  | INode (tl, x, tr) -> size tl + 1 + size tr
;;
val size : ('a, 'b) int_tree -> 'b int
let rec sum = function
    ILeaf -> 0
  | INode (tl, x, tr) -> sum tl + x + sum tr
;;
val sum : ('a, 'a) int_tree -> 'a int
```

Ces deux fonctions reçoivent des types différents, puisque la première ne révèle de l'information que sur la structure de l'arbre qui lui est passé en argument, alors que la seconde utilise également la valeur des étiquettes.

Une alternative possible consiste à annoter le type des arbres binaires d'étiquettes entières avec un seul niveau d'information :

```
type 'a int_tree1 =
    ILeaf1
  | INode1 of 'a int_tree1 * 'a int * 'a int_tree1
```

```
  # 'a
;;
type (#'a:level) int_tree1 =
    ILeaf1
  | INode1 of 'a int_tree1 * 'a int * 'a int_tree1
  # 'a
;;
```

L'information portée par les étiquettes n'est plus ici distinguée de celle associée à la structure des arbres. Ce choix permet d'obtenir des types plus simples et plus concis, mais également moins précis. Par exemple, les deux fonctions calculant la taille et la somme des étiquettes d'un arbre reçoivent, avec cette déclaration, le même schéma de type :

```
let rec size1 = function
    ILeaf1 -> 0
  | INode1 (tl, x, tr) -> size1 tl + 1 + size1 tr
;;
val size1 : 'a int_tree1 -> 'a int
let rec sum1 = function
    ILeaf1 -> 0
  | INode1 (tl, x, tr) -> sum1 tl + x + sum1 tr
;;
val sum1 : 'a int_tree1 -> 'a int
```

Le type obtenu pour `size1` donne ainsi une description moins précise du comportement de la fonction que celui de `size` : il ne reflète pas l'absence de dépendance entre les étiquettes de l'arbre et sa taille, comme montré par les expressions suivantes :

```
size (INode (ILeaf, x1, ILeaf));;
- : 'a int
size1 (INode1 (ILeaf1, x1, ILeaf1));;
- : !alice int
```

4.1.2 Enregistrements

► **Enregistrements simples** Les enregistrements peuvent, dans un premier temps, être vus comme une forme améliorée de n-uplets dont les composantes sont accessibles par noms. Comme exemple, je définis un type enregistrement pour représenter des points ou vecteurs dans un espace à deux dimensions :

```
type 'a vector =
  { x: 'a int;
    y: 'a int
  }
;;
type (#'a:level) vector = { x: 'a int; y: 'a int }
```

Comme indiqué par la réponse de la boucle interactive, le constructeur de type `vector` a un argument, qui est un niveau covariant et gardé. Il s'agit du niveau commun des deux entiers formant le vecteur. Comme pour les n-uplets, il n'y a pas de niveau supplémentaire attaché à l'enregistrement lui-même, puisqu'il n'est pas directement observable dans le langage.

Quand un vecteur est construit à partir de deux entiers, son niveau est l'union de ceux associés aux entiers :

```
let v = { x = x1; y = x2 };;
val v : [> !alice, !bob] vector
```

On peut définir quelques opérations usuelles sur les vecteurs :

```
let add_vector v1 v2 =
  { x = v1.x + v2.x;
```

```
    y = v1.y + v2.y
  }
;;
val add_vector: 'a vector -> 'a vector -> 'a vector
let rot_vector v =
  { x = - v.y;
    y = v.x
  }
;;
val rot_vector: 'a vector -> 'a vector
```

La manière dont j'ai annoté le type `vector` à l'aide d'un seul niveau dans la déclaration précédente est quelque peu arbitraire. En effet, il aurait également été possible de distinguer l'information portée par chacune des coordonnées du vecteur en introduisant deux niveaux :

```
type ('a, 'b) vector2 =
  { x2: 'a int;
    y2: 'b int
  }
;;
type (#'a:level, #'b:level) vector = { x2: 'a int; y2: 'b int }
```

D'une manière similaire à ce que nous avons observé pour les variants, cette déclaration permet d'obtenir des types parfois plus précis, mais également plus verbeux :

```
let add_vector2 v1 v2 =
  { x2 = v1.x2 + v2.x2;
    y2 = v1.y2 + v2.y2
  }
;;
val add_vector2: ('a, 'b) vector -> ('a, 'b) vector -> ('a, 'b) vector
let rot_vector2 v =
  { x2 = - v.y2;
    y2 = v.x2
  }
;;
val rot_vector2: ('a, 'b) vector2 -> ('b, 'a) vector2
```

En particulier, le type de la fonction `rot_vector2` montre clairement qu'elle effectue quelque permutation entre les deux composantes du vecteur passé en argument.

▶ **Enregistrements mutables** Comme en Objective Caml, les enregistrements peuvent également avoir des champs *mutables* dont le contenu peut être modifié en place. Ils sont déclarés à l'aide du mot-clef **`mutable`** :

```
type ('a, 'b) mvector =
  { mutable mx: 'a int; mutable my: 'a int } # 'b
;;
type (='a:level, #'b:level) mvector = {
  mutable mx : 'a int;
  mutable my : 'a int;
} # 'b
```

Ce fragment de code définit un type pour des vecteurs dont les coordonnées sont mutables. Il appelle deux commentaires. Tout d'abord, un champ mutable est à la fois un canal d'entrée et de sortie pour une valeur, puisque son contenu est accessible en lecture et en écriture. Puisque ces deux canaux sont décrits par un seul type, celui-ci doit être considéré comme *invariant*. C'est pourquoi, dans la définition ci-dessus, le premier paramètre du constructeur de type `mvector`, qui apparaît dans le type des deux champs mutables, est invariant. D'autre part, un enregistrement comprenant un champ mutable ne peut plus être vu comme un simple n-uplet. L'information qu'il porte n'est

pas entièrement contenue dans ses composantes puisque son identité (*i.e.* sa matérialisation en mémoire) peut être observée dans le langage, par exemple en modifiant la valeur d'un de ses champs. C'est pourquoi, le type `mvector` doit porter un niveau supplémentaire, comme les types des références. Dans l'exemple ci-dessus, ce rôle est joué par le paramètre `'b` donné dans la clause `# 'b` à la fin de la définition.

Pour illustrer l'utilisation d'un tel type, définissons une fonction qui effectue une rotation *en place* d'un vecteur.

```
let rot_mvector v =
  let x = v.mx in
  v.mx <- v.my;
  v.my <- x
;;
val rot_mvector : ('a, 'a) mvector -{'a ||}-> unit
```

Puisqu'ils sont modifiés par cette fonction, les entiers du vecteur donné en argument sont susceptibles de porter de l'information à la fois sur le contexte d'appel de la fonction et l'identité du vecteur modifié lui-même.

Dans la section 3.1 (page 53), j'ai introduit les références. Toutefois il s'agit simplement d'un cas particulier d'enregistrement à un champ mutable, qui peut être défini comme suit :

```
type ('a, 'b) ref =
  { mutable contents: 'a } # 'b
;;
type ('a:type, #'b:level) ref = { mutable contents: 'a } # 'b
```

De plus, les trois opérations primitives `ref`, `:=` et `!` sur les références sont des fonctions normales qui sont implémentées dans le module `Pervasives` à partir de la représentation des références comme enregistrements :

```
let ref x =
  { contents = x }
;;
val ref : 'a -> ('a, _) ref
let (:=) r x =
  r.contents <- x
;;
val ( := ) : ('a, 'b) ref -> 'a -{'b ||}-> unit
             with 'b < level('a)
let ( ! ) r =
  r.contents
;;
val ( ! ) : ('a, 'b) ref -> 'c
            with 'b < level('c)
            and  'a < 'c
```

4.2 Interaction avec le monde extérieur

Un programme Flow Caml peut être considéré comme un processus qui reçoit de l'information d'une ou plusieurs sources extérieures, effectue des calculs et envoie ses résultats (intermédiaires ou finaux) à un ou plusieurs receveurs également extérieurs. Avec ce point de vue, le but ultime du système de types est de vérifier que chaque flot d'information potentiel entre une source et un receveur engendré par l'exécution du programme est légal au regard de la politique de sécurité du système informatique. Il me faut expliquer comment ces entités externes sont représentées en Flow Caml, et comment les règles de propagation de l'information peuvent être définies par le programmeur.

Dans la littérature, les détenteurs d'information sont généralement appelés *principaux*. Du point de vue d'un programme donné, chacun d'entre eux peut être une source, un receveur ou même les deux à la fois. Suivant les contextes, la notion de principal peut recouvrir une grande variété de concepts : des (groupes d')utilisateurs, des niveaux de confidentialité (*public*, *secret*, etc.), des sous-ensembles d'un espace de stockage, des canaux de communication *via* une interface réseau, etc. Cependant, Flow Caml n'est pas concerné par la matérialisation de ces entités, et les traite d'une manière uniforme : dans son système de type, les principaux sont représentés par les constantes de niveau d'information, comme `!alice`, `!bob` et `!charlie` que nous avons rencontré jusqu'à présent dans les exemples. Leur introduction était cependant relativement artificielle, puisque je n'ai pas précisé comment il était effectivement possible d'échanger de l'information avec les principaux qu'ils symbolisent. Je donne maintenant un exemple plus réaliste avec les canaux de communication établis par l'entrée et la sortie standards du programme.

4.2.1 L'exemple de l'entrée et de la sortie standards

L'entrée standard et la sortie standard sont des canaux de communications ouverts respectivement en lecture et en écriture. Ils sont par convention représentés par les deux niveaux d'information : `!stdin` et `!stdout`. Le module `Pervasives` de la bibliothèque standard fournit quelques primitives pour utiliser ces canaux. Par exemple, la fonction `print_int` imprime un entier sur la sortie standard :

```
print_int;;
- : !stdout int -{!stdout ||}-> unit
```

Puisque l'entier donné en argument est envoyé sur la sortie standard, son niveau doit être inférieur ou égal à `!stdout`. Le littéral `1` ayant le type `'a int` pour tout niveau `'a`, il peut être passé comme argument à `print_int` :

```
print_int 1;;
- : unit
```

À l'inverse, l'entier `x1` ayant le niveau `!alice`, il n'est pas possible — pour l'instant — de l'afficher sur la sortie standard :

```
print_int x1;;
This expression generates the following information flow:
  from !alice to !stdout
which is not legal.
```

En effet, ce fragment de code génère un flot d'information du principal « Alice » vers le principal « sortie standard ». Il requiert par conséquent l'inégalité `!alice < !stdout`. Celle-ci n'est pas satisfaite par le treillis par défaut où tous les principaux sont incomparables, *i.e.* ne peuvent échanger d'information. Cette politique de sécurité peut être relaxée en déclarant de nouvelles inégalités entre niveaux grâce aux déclarations introduites par le mot-clef **`flow`** :

```
flow !alice < !stdout;;
```

Cette déclaration modifie la structure du treillis sous-jacent de telle sorte que le point correspondant à `!alice` devienne inférieur ou égal à celui de `!stdout`. Concrètement, cela revient à autoriser les flots d'information du principal représenté par le premier (Alice) vers celui du second (la sortie standard). Ces déclarations sont transitives. Par exemple, si on déclare :

```
flow !bob < !alice;;
```

alors Bob peut envoyer de l'information à Alice, mais aussi, par transitivité, à la sortie standard :

```
print_int x2;;
- : unit
```

Il faut bien noter que ces niveaux d'informations sont *globaux*, de même que les déclarations qui les relient. Cela est naturel puisque les principaux et la politique de sécurité qu'ils représentent le sont également. Cependant, de manière à préserver son caractère incrémental, la boucle interactive

permet au programmeur de relaxer la politique de sécurité progressivement : il peut entrer une déclaration **flow** à chaque invite, laquelle reste valide jusqu'à la fin de la session. Cette flexibilité ne va pas à l'encontre de la correction du système, puisqu'un fragment de programme typé dans un certain treillis reste bien typé dans un affaiblissement de ce treillis.

Le niveau de sécurité `!stdin` représente le canal de communication relié à l'entrée standard dans le système de types. Par exemple, la fonction `read_line` a le type suivant :

```
read_line;;
- : unit -{[< !stdout, !stdin] | End_of_file: !stdin |}-> !stdin string
```

D'après la documentation de la librairie standard d'Objective Caml, la fonction `read_line` vide le tampon de la sortie standard, puis lit des caractères sur l'entrée standard jusqu'à ce qu'un retour à la ligne soit rencontré. Ainsi, une invocation de `read_line` produit des effets sur à la fois l'entrée et la sortie standard, ce qui explique la première annotation sur la flèche du type ci-dessus. De plus, si l'utilisateur envoie le caractère de fin de fichier (*e.g.* en tapant `^D`), la fonction lève une exception `End_of_file`, d'où la deuxième annotation sur la flèche. Enfin, la chaîne obtenue en lisant l'entrée standard doit avoir le niveau `!stdin` :

```
let s1 =
  try read_line ()
  with End_of_line -> ""
;;
val s1 : !stdin string
```

Remarquons que, si on veut écrire sur la sortie standard une chaîne lue sur l'entrée standard (ou toute valeur calculée à partir de celle-ci), la politique de sécurité doit autoriser les transferts d'information de la seconde vers la première. Pour cela, il suffit d'entrer la déclaration suivante :

```
flow !stdin < !stdout;;
```

On peut alors écrire une fonction `echo` qui répète sur la sortie standard les caractères lus sur l'entrée standard :

```
let echo () =
  try
    while true do
      let s = read_line () in
      print_string s
    done
  with
    End_of_file -> ();;
val echo : unit -{[< !stdout, !stdin] ||}-> unit
```

4.2.2 Principaux

Les programmes réels peuvent généralement communiquer avec l'extérieur en utilisant d'autres canaux que les simples entrée et sortie standards, comme le système de fichiers, une interface réseau ou un périphérique d'affichage. Cependant, la librairie standard de Flow Caml ne fournit pas de fonction permettant de telles communications : analyser ces opérations de bas niveau à l'aide de son système de types ne permettrait pas d'obtenir une description pertinente de leur contenu vis-à-vis de la politique de sécurité. Des considérations très raffinées sont généralement nécessaires pour prouver leur sûreté. C'est pourquoi l'interaction avec ces entités externes doit être modélisée, en Flow Caml, à un plus haut niveau, dépendant de la manière dont elles sont utilisées dans le programme.

C'est la raison pour laquelle un programme écrit et vérifié avec le système Flow Caml doit généralement être divisé en deux parties. La première fournit un modèle de haut niveau des principaux externes. Cela consiste généralement d'une série de fonctions permettant de communiquer avec ces entités, qui sont implémentées dans un ou plusieurs modules écrits en Objective Caml, en utilisant

par exemple les primitives d'entrée/sortie du module `Pervasives`, la librairie `Unix` ou une librairie graphique. Cette partie du code ne peut pas être vérifiée par Flow Caml : le programmeur doit fournir lui-même une interface (*i.e.* leurs types) pour ces fonctions de haut niveau, laquelle décrit leurs comportements supposés vis-à-vis de la politique de sécurité. Le reste du programme peut être écrit en Flow Caml. Il interagit avec l'extérieur *via* les fonctions de haut niveau fournies par la première partie. Cette seconde partie peut être typée par Flow Caml, de manière à la vérifier automatiquement. Dans la section 4.4 (page 80), je donnerai un exemple détaillé de la mise en œuvre de ce schéma. Celui-ci nécessite de découper le programme en plusieurs *unités de compilation*, lesquelles sont des cas particuliers de modules globaux. C'est pourquoi, avant d'aborder cette question, je présente dans la section suivante la couche modulaire du langage Flow Caml.

4.3 Le langage de modules

Les fondations du système de module de Flow Caml sont identiques à celles d'Objective Caml [Ler00] : les *structures* sont des séquences de définitions, les *signatures* sont des interfaces pour les structures, et les *foncteurs* sont des « fonctions » des structures vers les structures. Sa mise en œuvre a cependant nécessité quelques adaptations, telles que l'ajout des déclarations **`level`** (section 4.3.2). Son utilisation nécessite la prise en compte des annotations portées par les types, et introduit de ce fait quelques difficultés nouvelles. Je les explique dans les sections suivantes.

4.3.1 Structures et signatures

Une *structure* est consituée d'une séquence de définitions, à l'intérieur d'une construction **`struct`** ... **`end`**, et généralement nommée par une liaison **`module`**. Par exemple, on peut définir une structure implémentant des ensembles d'entiers par des arbres binaires :

```
module IntSet = struct

  type 'a t =
      Empty
    | Node of 'a t * 'a int * 'a t
    # 'a

  let empty = Empty

  let rec add x = function
      Empty -> Node (Empty, x, Empty)
    | Node (l, y, r) ->
        if x < y then Node (add x l, y, r)
        else Node (l, y, add x r)

  let rec mem x = function
      Empty -> false
    | Node (l, y, r) ->
        (x = y) || mem x (if x < y then l else r)

end;;
module IntSet : sig
  type (#'a:level) t = Empty | Node of 'a t * 'a int * 'a t # 'a
  val empty : 'a t
  val add : 'a int -> 'a t -> 'a t
  val mem : 'a int -> 'a t -> 'a bool
end
```

Cette structure inclut une définition de type — le type `t` des ensembles d'entiers — et trois valeurs : l'ensemble vide, `empty`, et deux fonctions opérant sur les ensembles, `add` et `mem`. La boucle

interactive répond en donnant la signature de la structure, qui est la liste de ses composantes avec leurs déclarations ou types.

```
IntSet.add x1 (IntSet.add x2 IntSet.empty);;
- : [> !alice, !bob] IntSet.t
```

Les signatures permettent d'abstraire quelques caractéristiques de l'implémentation d'une structure en cachant certains composants ou bien en les exportant avec des déclarations ou types restreints. On peut par exemple cacher la représentation concrète des ensembles d'entiers :

```
module type INTSET = sig
  type (#'a:level) t
  val empty: 'a t
  val add: 'a int -> 'a t -> 'a t
  val mem: 'a int -> 'a t -> 'a bool
end;;
module AbstractIntSet = (IntSet : INTSET);;
module AbstractIntSet : INTSET
```

Il faut remarquer que les paramètres des déclarations de types abstraits doivent être annotés par leurs sortes et variances dans les signatures : en l'absence de représentation concrète du type, ils ne peuvent plus être inférés par le système, mais leur connaissance est toutefois nécessaire au typage du reste du programme.

4.3.2 Foncteurs

Les foncteurs sont des « fonctions » des structures vers les structures qui permettent d'exprimer des structures paramétrées. Un exemple courant est la définition d'une librairie implémentant des ensembles, paramétrée par une structure donnant le type des éléments et une fonction d'ordre total sur ces derniers.

```
module type ORDERED_TYPE = sig
  type (#'a:level) t
  val compare : 'a t -> 'a t -> 'a int
end;;
```

Comme il est usuel en Caml, `compare x y` est supposée retourner `0` si `x` et `y` sont égaux, un entier négatif si `x` est strictement inférieur à `y` et un entier positif sinon. Dans cette signature, le type des éléments, `t`, est paramétré par un seul niveau d'information, qui décrit toute l'information potentiellement obtenue lors d'une comparaison, comme le reflète le type de `compare`. Cependant, cela n'empêche aucunement cette signature d'être utilisée avec des structures de données complexes dont le type porte plusieurs niveaux d'information :

```
module IntList : ORDERED_TYPE = struct
  type 'a t = ('a int, 'a) list
  let rec compare l1 l2 =
    match l1, l2 with
      [], [] -> 0
    | [], _ :: _ -> -1
    | _ :: _, [] -> 1
    | hd1 :: tl1, hd2 :: tl2 ->
      let c = Pervasives.compare hd1 hd2 in
      if c = 0 then compare tl1 tl2
      else c
end;;
module IntList : sig
  type (#'a:level) t
  val compare : 'a t -> 'a t -> 'a int
end
```

Dans ce module, le paramètre du type `t` représente l'*union* des deux annotations portées par le type des listes d'entiers. Cela permet à la fonction `compare` qui suit d'accéder à la fois à la structure des listes et à leurs éléments pour les comparer.

Je définis maintenant le foncteur qui réalise des ensembles sur des éléments de type arbitraire. Ce foncteur prend une structure `Elt` comme argument, laquelle doit avoir la signature `ORDERED_TYPE` :

```
module Set (Elt: ORDERED_TYPE) = struct

  type 'a element =
    'a Elt.t

  type 'a t =
      Empty
    | Node of 'a t * 'a element * 'a t
    # 'a

  let empty = Empty

  let rec add x = function
      Empty -> Node (Empty, x, Empty)
    | Node (l, y, r) ->
        if Elt.compare x y < 0 then Node (add x l, y, r)
        else Node (l, y, add x r)

  let rec mem x = function
      Empty -> false
    | Node (l, y, r) ->
        let c = Elt.compare x y in
        (c = 0) || mem x (if c < 0 then l else r)

end;;
module Set : functor (Elt : ORDERED_TYPE) -> sig
  type (#'a:level) element = 'a Elt.t
  type (#'a:level) t = Empty | Node of 'a t * 'a Elt.t * 'a t # 'a
  val empty : 'a t
  val add : 'a Elt.t -> 'a t -> 'a t
  val mem : 'a Elt.t -> 'a t -> 'a bool
end
```

Comme dans l'exemple de `IntSet`, un bon style de programmation consiste à cacher l'implémentation concrète du type des ensembles. Cela peut être effectué en restreignant le type du foncteur `Set`. Définissons tout d'abord la signature de la structure produite par le foncteur :

```
module type SET = sig

  type (#'a:level) element
  type (#'a:level) t

  val empty: 'a t
  val add: 'a element -> 'a t -> 'a t
  val mem: 'a element -> 'a t -> 'a bool

end;;
```

Le type du foncteur `Set` peut être restreint lors de sa définition, en annotant son en-tête comme suit :

```
module Set (Elt: ORDERED_TYPE)
```

```
            : (SET with type 'a element = 'a Elt.t) = struct
  ...
end
```

La contrainte de type `with type 'a element = 'a Elt.t` a le même rôle qu'en Objective Caml : elle raffine la signature `SET` de manière à exprimer le fait que les ensembles contiennent des éléments de type `'a Elt.t`. Pour conclure avec cet exemple, observons qu'il est possible d'obtenir une nouvelle implémentation des ensembles d'entiers, comme une instance du foncteur `Set`, qui a la même signature que celle obtenue de manière directe :

```
module IntSet' = Set (struct
  type 'a t = 'a int
  let compare = Pervasives.compare
end);;
module IntSet' : sig
  type 'a element = 'a int
  type 'a t
  val empty : 'a t
  val add : 'a element -> 'a t -> 'a t
  val mem : 'a element -> 'a t -> 'a bool
end
```

Dans les exemples précédents, l'interaction entre le langage de module et l'analyse de flots mise en œuvre par le système de type de Flow Caml est relativement simple. Elle nécessite essentiellement de fournir les annotations utiles dans les déclarations de valeurs et de types des signatures. Le typage de certaines structures et foncteurs nécessite toutefois d'introduire des déclarations de niveaux abstraits, introduites par le mot-clef **`level`**, qui ont un rôle comparable à celles des déclarations de types. Par exemple, on peut définir un type de module pour des structures implémentant un canal de communication ouvert en lecture comme suit :

```
module type IN = sig
  level Data
  level Prompt
  val read : unit -{Prompt ||}-> Data string
end;;
```

Cette signature fait intervenir deux niveaux d'information abstraits : `Data` est le niveau des données lues sur le canal ; et `Prompt` représente l'information potentiellement transmise sur le canal lorsqu'un processus effectue une lecture sur ce dernier. Dans cette signature, ces niveaux sont totalement inconnus : on dit qu'ils restent *abstraits*. Ils sont toutefois utilisés dans le type de la fonction `read` , qui est supposée lire une ligne de texte sur le canal sous-jacent : cette fonction a la possibilité d'effectuer un effet de niveau `Prompt` et retourne une chaîne de niveau `Data`. Remarquons que, avec ce type pour `read`, la lecture sur le canal est supposée ne jamais échouer. Voici une implémentation de cette signature pour l'entrée standard :

```
module Stdin = struct
  level Data = !stdin
  level Prompt less than !stdin, !stdout
  let read () =
    try read_line ()
    with End_of_file -> ""
end;;
module Stdin : sig
  level Data = !stdin
  level Prompt less than !stdout, !stdin
  val read : unit -{[< !stdout, !stdin] ||}-> !stdin string
end
```

Les chaînes lues sur l'entrée standard ont le niveau `!stdin` auquel `Data` est déclaré égal. De plus, une invocation de `read_line` affecte à la fois l'entrée standard et la sortie standard (puisque le

cache de cette dernière est vidé avant la lecture), de telle sorte que le niveau `Prompt` doit être inférieur ou égal à `!stdin` et `!stdout`. Ainsi, le module `Stdin` implémente la signature `IN`, ce qui peut être immédiatement vérifié par une contrainte de type :

```
module AbstractStdin = (Stdin : IN);;
module AbstractStdin : IN
```

De la même manière, je peux déclarer un type de module pour les canaux ouverts en écriture, et en définir une instance avec la sortie standard :

```
module type OUT = sig
  level Data
  val print : Data string -{Data ||}-> unit
end;;

module Stdout = struct
  level Data = !stdout
  let print = print_endline
end;;
module Stdout : sig
  level Data = !stdout
  val print : !stdout string -{!stdout ||}-> unit
end;;
```

Dans ce cas, un seul niveau est nécessaire, `Data`, qui représente l'information qui peut être envoyée sur le canal. (Je ne considère pas la possibilité de *recevoir* de l'information d'un canal ouvert en écriture, par exemple *via* sa saturation.)

Je cherche maintenant à écrire un foncteur paramétré par deux structures implémentant respectivement un canal d'entrée et un canal de sortie. Le corps de ce foncteur définira une fonction `copy` qui permet simplement de lire une ligne sur le canal ouvert en lecture et de la recopier sur celui ouvert en écriture. De ce fait, il n'est pas suffisant de déclarer les deux paramètres du foncteur comme étant de signatures respectives `IN` et `OUT` : en effet, la fonction `copy` génère un flot d'information entre les deux canaux. Le niveau d'information `Data` du premier doit donc être inférieur ou égal à celui du second :

```
module Copier (I : IN)
              (O : OUT with level Data greater than I.Data) = struct
  let copy () =
    O.print (I.read ())
end;;
module Copier :
  functor (I : IN) ->
    functor
      (O : sig
             level Data greater than I.Data
             val print : Data string -{Data ||}-> unit
           end) ->
      sig
        val copy : unit -{[< O.Data, I.Prompt] ||}-> unit
      end
```

Cela est réalisé ci-dessus grâce à la contrainte **`with level`** qui apparaît dans le type du second paramètre du foncteur. Son effet est similaire à celle des **`with type`** et **`with module`** d'Objective Caml : elle raffine la définition du niveau `Data` dans la signature du module `O`. Il faut noter que l'ordre dans lequel apparaissent les deux paramètres du foncteur introduit une certaine asymétrie, puisque la contrainte est appliqué au second. Il est naturellement possible de permuter `I` et `O` comme suit :

```
module Copier' (O : OUT)
               (I : IN with level Data less than O.Data) = struct
```

```
  let copy () =
    O.print (I.read ())
end;;
module Copier' :
  functor (O : OUT) ->
    functor
      (I : sig
             level Data less than O.Data
             level Prompt
             val read : unit -{Prompt ||}-> Data string
           end) ->
      sig
        val copy : unit -{[< I.Prompt, O.Data] ||}-> unit
      end
```

Pour terminer cet exemple, définissons une instance de `Copier` opérant sur l'entrée standard et la sortie standard. Il suffit pour cela d'entrer la définition suivante :

```
module StdCopier = Copier (Stdin) (Stdout);;
```

Celle-ci produit toutefois une erreur :

```
Signature mismatch:
Modules do not match:
  sig
    level Data = !stdout
    val print : !stdout string -{!stdout ||}-> unit
  end
is not included in
  sig
    level Data greater than Stdin.Data
    val print : Data string -{Data ||}-> unit
  end
Level declarations mismatch: the provided type declaration
  level Data = !stdout
is not included in the expected one
  level Data greater than Stdin.Data
The inequality Stdin.Data < Data is required but not provided
```

En effet, en l'état actuel des choses, le module `Stdout` ne satisfait pas la signature `OUT with level Data greater than Stdin.Data` puisque le niveau `Stdout.Data`, qui est égal au principal `!stdout` n'est pas supérieur ou égal à `Stdin.Data` qui vaut `!stdin`. En d'autres termes, il est nécessaire d'avoir l'inégalité `!stdin < !stdout` puisque la fonction `copy` fourni par cette instance de `Copier` engendre un flot d'information de l'entrée standard vers la sortie standard. Pour autoriser cela, il suffit de relaxer la politique de sécurité comme suit :

```
flow !stdin < !stdout;;
```

Désormais, `Stdin.Data` est inférieur à `Stdout.Data`, de telle sorte que `Stdin` et `Stdout` sont des arguments légaux pour le foncteur `Copier` :

```
module StdCopier = Copier (Stdin) (Stdout);;
module StdCopier :
  sig val copy : unit -{[< Stdout.Data, Stdin.Prompt] ||}-> unit
end
```

4.3.3 Effets de bord, exceptions et modules

Le typage d'une structure en Flow Caml présente une difficulté supplémentaire : il est nécessaire de prendre en compte les flots d'information potentiellement engendrés par l'évaluation en *séquence* des définitions. Dans une structure, seules les définitions `let` ont un contenu calculatoire

pouvant produire des effets de bord ou lever des exceptions. En plus d'un type pour chacun des identificateurs liés, le système de type de Flow Caml associe à chaque définition **let** deux listes de niveaux d'information appelées *bornes*. La première est une *borne inférieure* sur les effets de bords produits par l'évaluation du corps de la définition : elle reprend les niveaux des structures de données qui peuvent être modifiées par son exécution. La seconde est une *borne supérieure* sur les niveaux des exceptions qui peuvent potentiellement s'échapper. Au contraire des expressions, il n'est pas nécessaire de considérer une borne par nom d'exception : en effet, si une exception s'échappe d'une définition **let** alors elle termine nécessairement l'exécution du programme, puisqu'il n'existe aucune construction dans le langage de modules pour la rattraper. Ainsi, seule la levée d'une exception est observable à ce niveau du langage, via la terminaison, et pas le nom de l'exception levée.

Ce procédé de calcul de bornes est étendu à chacune des constructions du langage de modules. Puisqu'elles n'ont pas d'effet à l'exécution, les définitions **external**, **type**, **level**, **exception**, **module type**, **open** et **include** ont des bornes vides. Les bornes d'une définition **module** sont obtenues en considérant récursivement l'expression de module qui apparaît dans son membre droit. Par exemple, puisque l'évaluation de la structure **struct** $\texttt{def}_1$ ... $\texttt{def}_n$ **end** consiste à évaluer successivement chacune de ses définitions, ses bornes sont naturellement l'*union* de celles associées aux définitions $\texttt{def}_1$, ..., $\texttt{def}_n$. De plus, si l'une de ces définitions lève une exception, celle-ci termine l'exécution de la structure. Ainsi, lorsqu'elle est évaluée, la définition $\texttt{def}_{i+1}$ observe que les définitions précédentes, $\texttt{def}_1 \ldots \texttt{def}_i$, n'ont pas levé d'exception. Par conséquent, pour prévenir tout flot d'information illégal, les bornes supérieures de ces dernières doivent être inférieures ou égales à la borne inférieure de $\texttt{def}_{i+1}$.

Les définitions entrées dans la boucle interactive Flow Caml, comme toutes celles que nous avons rencontrées jusqu'à présent, peuvent être vues comme les définitions d'une structure, et ont donc des bornes. Celles-ci ne sont cependant pas directement affichées à l'utilisateur. À la place, à chaque invite, le système garde en mémoire la borne supérieure de toutes les définitions entrées jusqu'à ce point. Lorsque l'utilisateur entre une nouvelle définition, le système vérifie que sa borne inférieure est supérieure ou égale à la borne courante, puis augmente cette dernière de la borne supérieure de la nouvelle définition. Un message est affiché à chaque fois que le niveau du contexte global est modifié :

```
if x1 then raise X;;
Current evaluation context has level !alice
if x2 then raise X;;
Current evaluation context has level !alice, !bob
```

Une définition de foncteur n'effectue aucun calcul : son corps n'est évalué qu'aux points où le foncteur est appliqué. Si les bornes du foncteur lui-même sont de ce fait vides, celles du corps doivent être mémorisées sur son type flèche, de manière à pouvoir être prises en compte quand le foncteur est utilisé. Ces annotations correspondent, au niveau du langage de module, aux deux premières annotations apparaissant sur les types flèches du langage de base. (La troisième annotation des fonctions n'est cependant pas requise pour les foncteurs car une structure ne peut pas être le résultat d'un calcul dépendant d'une source extérieure.)

```
module type S = sig
  val x : 'a int
end;;

module F (X: S) = struct
  let _ =
    r1 := X.x;
    if X.x = x2 then raise Exit
end;;
module F : functor (X : S) -{!alice | !bob}-> sig  end
```

Ce type de module signifie qu'une application du foncteur `F` peut modifier le contenu d'une cellule de niveau `!alice` et peut lever une exception au niveau `!bob`. Ainsi, une application de `F` insère le principal `!bob` dans le niveau courant du contexte global :

```
module F0 = F (struct let x = 1 end);;
Current evaluation context has level !bob
```

De plus, il n'est possible d'instancier `F` qu'à un point du programme où le niveau du contexte est inférieur ou égal à `!alice`, ce qui n'est plus le cas après une première application du foncteur (dans la politique de sécurité vide) :

```
module F1 = F (struct let x = 1 end);;
This expression is executed in a context of level !bob
but has an effect at level !alice.
This yields the following information flow:
  from !bob to !alice
which is not legal.
```

4.4 Programmes autonomes

Tous les exemples donnés jusqu'à présent étaient entrés dans le système interactif. Cependant, on peut également écrire des programmes complets en Flow Caml. Les fichiers sources sont typés par le « compilateur » en ligne de commande `flowcamlc`, puis traduits par ce dernier en des fichiers sources Objective Caml normaux. Ces derniers peuvent alors être compilés en utilisant les compilateurs `ocamlc` ou `ocamlopt` pour produire un exécutable.

Dans cette section, je mets en œuvre ce procédé à partir d'un exemple de programme concret. Dans les systèmes `Unix` munis de « *shadow passwords* », les données relatives aux utilisateurs sont stockées dans deux fichiers. D'une part, le fichier `/etc/passwd` contient la liste des noms d'utilisateurs (*logins*), avec, pour chacun d'entre eux, un mot de passe et quelques informations administratives telles qu'un identifiant numérique ou le chemin du répertoire de l'utilisateur. D'autre part, le fichier `/etc/shadow` associe à chaque nom d'utilisateur un mot de passe qui est utilisé à la place de celui donné dans `/etc/passwd`. Le programme qui sert d'exemple dans cette section permet de synchroniser ces deux fichiers en générant une entrée dans `/etc/shadow` pour chaque utilisateur listé dans `/etc/passwd`. Son code source est donné pages 85 à 89. Dans les paragraphes suivants, j'explique comment il est organisé, puis comment il est vérifié et compilé avec le système Flow Caml. Le système de type aidera à montrer qu'une exécution de ce programme ne peut pas révéler à l'utilisateur qui lance la commande une information sur les mots de passe stockés dans les deux fichiers.

4.4.1 Unités de compilation et compilation en ligne de commande

Le code source d'un programme est généralement séparé en plusieurs fichiers définissant des *unités de compilation*, lesquelles peuvent être compilées séparément. En Flow Caml, la définition d'une unité de compilation `A` est constitué d'un ou deux fichiers parmi :

- une implémentation *a*`.fml`, qui contient une séquence de définitions analogue à celle d'une construction **`struct`** ... **`end`** ;
- une interface *a*`.fmli`, qui contient une séquence de spécifications analogue à celle d'une construction **`sig`** ... **`end`**.

Ces deux fichiers définissent une structure nommée `A`, comme si la définition suivante était entrée dans la boucle interactive :

```
module A : sig (* specifications du fichier a.fmli *) end
         = struct (* definitions du fichier a.fml *) end;;
```

Les fichiers définissant un ensemble d'unités de compilation peuvent être traités de manière séparée par la commande `flowcamlc`, en suivant, pour chaque unité de compilation `A`, l'un des trois schémas

suivants :

1. La compilation d'une unité `A` pour laquelle sont données à la fois une implémentation *a*.`fml` et une interface *a*.`fmli` est décrite par la figure 4.1. Tout d'abord, l'interface Flow Caml *a*.`fmli` est donnée au compilateur `flowcamlc`, qui vérifie sa bonne formation et produit une version compilée de celle-ci, *a*.`fcmi`, ainsi qu'une traduction en Objective Caml *a*.`mli`. Ensuite, l'implémentation *a*.`fml` peut être typée par `flowcamlc`. Le système détermine son interface la plus générale, et vérifie qu'elle satisfait celle déclarée (stockée dans *a*.`fcmi`). De plus, il traduit cette implémentation en Objective Caml, produisant le fichier *a*.`fml`. Enfin, les fichiers *a*.`fmli` et *a*.`fml` définissent une unité de compilation Objective Caml normale, qui peut être compilée avec `ocamlc` pour produire une interface compilée *a*.`cmi` et un fichier objet *a*.`cmo`.
2. Comme en Objective Caml, il est également possible de définir une unité `A` en fournissant seulement un fichier implémentation *a*.`fml`, mais pas d'interface. Cela donne lieu au schéma de compilation donné figure 4.2 : l'implémentation *a*.`fml` peut directement être traitée par `flowcamlc`. L'interface produite par le processus d'inférence de types est quant-à-elle enregistrée dans *a*.`fcmi`.
3. Enfin, comme je l'ai expliqué à la section 4.2.2 (page 72), certaines parties du code source d'un programme ne peuvent être typées de manière satisfaisante en Flow Caml. Celles-ci peuvent être données dans une unité de compilation définie par une interface *a*.`fmli` en Flow Caml mais seulement une implémentation en Objective Caml *a*.`ml`. Cette possibilité est illustrée par le schéma de compilation de la figure 4.3. Dans ce cas, le système ne vérifie pas que le code dans *a*.`ml` satisfait l'interface *a*.`fmli` (mais seulement *a*.`mli`). En d'autres termes, si les erreurs de typage habituelles sont toujours détectées, il est du ressort du programmeur de s'assurer que les définitions de *a*.`ml` satisfont la spécification *a*.`fmli` au regard de la politique de sécurité. Cependant, les définitions exportées dans l'unité (et enregistrées dans *a*.`fcmi`) seront disponibles pour la partie du programme écrite en Flow Caml.

Le programme exemple est divisé en quatre unités de compilations. Les deux premières, `Passwd` et `Shadow`, sont des modules de bas niveau qui implémentent des fonctions pour accéder aux fichiers `/etc/passwd` et `/etc/shadow` : leur implémentation est directement écrite en Objective Caml dans les fichiers `passwd.ml` et `shadow.ml`, et seules les interfaces sont fournies du côté Flow Caml (fichiers `passwd.fmli` et `shadow.fmli`). Ces interfaces donnent les niveaux d'informations des données manipulées par ces unités : les informations stockées dans `/etc/passwd` reçoivent le niveau `!passwd_file`, excepté les mots de passe qui ont le niveau `!password`. De la même manière, l'information extraite de `/etc/shadow` reçoit les niveaux `!shadow_file` et `!shadow_password`. L'unité `Verbose` fournit un mode verbeux : si l'utilisateur lance le programme exemple avec l'option `-v`, alors une trace de son déroulement est donnée sur la sortie standard. Le corps du programme est dans l'unité `Main`. Ces deux dernières unités sont intégralement écrites en Flow Caml : une implémentation (fichiers `verbose.fml` et `main.fml`) et une interface (fichiers `verbose.fmli` et `main.fmli`) sont données pour chacun d'eux.

4.4.2 Déclarations `flow`

J'ai expliqué, à la section 4.2.1 (page 71) comment les déclarations **`flow`** permettent de relaxer la politique de sécurité en déclarant des inégalités entre principaux. Dans le cas de la boucle interactive, ces déclarations peuvent être fournies par le programmeur à tout moment et restent valides jusqu'à la fin de la session. Dans les programmes écrits avec le compilateur en ligne de commande `flowcamlc`, chaque unité de compilation doit être munie de sa propre politique de sécurité, *i.e.* d'une déclaration **`flow`** suffisante pour typer son code source. Cette déclaration doit être donnée au début des fichiers d'implémentation et d'interface. Par exemple, dans le programme exemple, l'unité de compilation commence par la déclaration suivante :

```
flow !arg < !stderr, !stdout
```

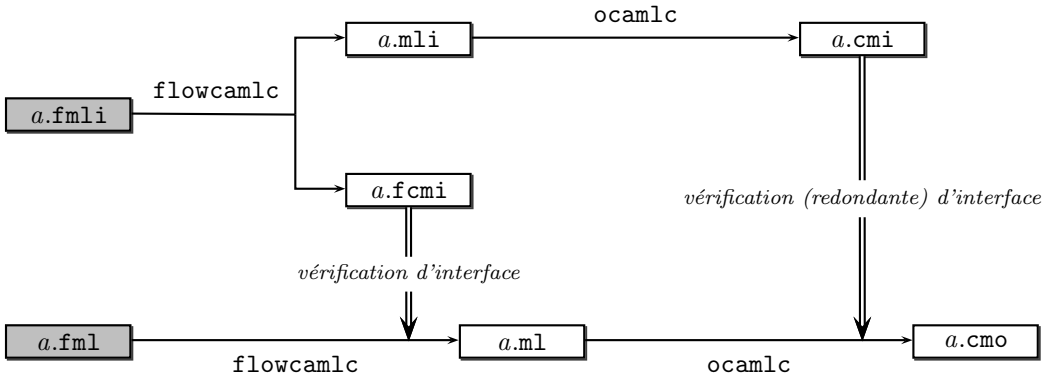


Figure 4.1 – Schéma de compilation d'une unité définie par a.fmli et a.fml

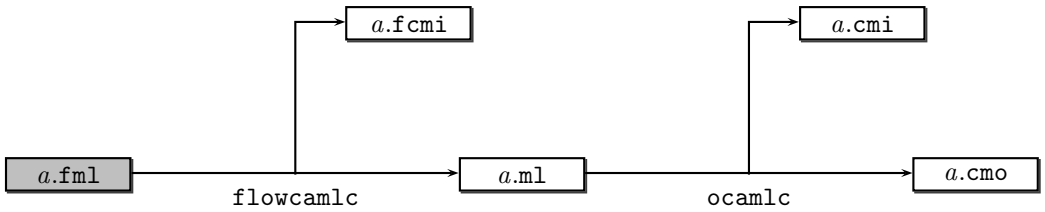


Figure 4.2 – Compilation d'une unité définie par a.fml

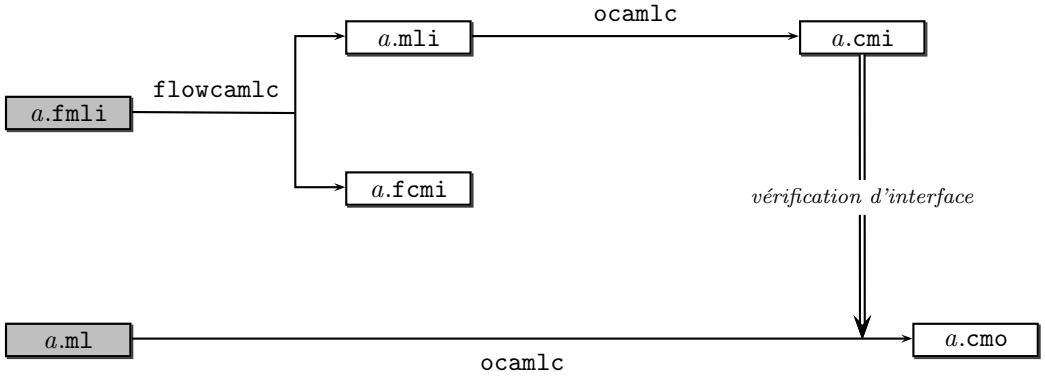


Figure 4.3 – Compilation d'une unité définie par a.fmli et a.ml

Par convention, les principaux `!stderr` et `!stdout` représentent respectivement la sortie d'erreur et la sortie standard du programme, et `!arg` est le niveau des arguments passés en ligne de commande. Cette déclaration est une abréviation pour

```
flow !arg < !stderr
and  !arg < !stdout
```

Elle signifie que l'unité de compilation est bien typée dans toute politique de sécurité qui satisfait les deux inégalités `!arg < !stderr` et `!arg < !stdout`, c'est-à-dire qui autorise les informations imprimées sur la sortie d'erreur et la sortie standard à dépendre des paramètres passés sur la ligne de commande du programme. Quand une unité de compilation ne comporte pas de déclaration **`flow`**, comme les unités `Passwd` et `Shadow` dans le programme exemple, cela signifie qu'elle est bien typée avec n'importe quelle politique de sécurité.

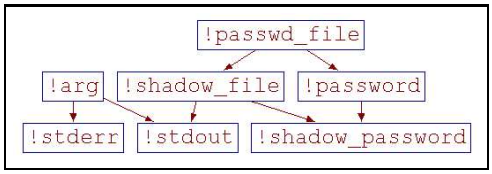
La politique de sécurité vis-à-vis de laquelle un programme consistant de plusieurs unités de compilation peut être considéré vérifié est tout simplement la superposition de celles déclarées dans les différentes unités. Ce principe de relaxation est correct puisque un fragment de code bien typé dans un certains treillis le reste dans un treillis plus flexible. Cependant, la possibilité de munir chaque unité de compilation d'une déclaration **`flow`** différente est essentielle pour la modularité de la programmation et la réutilisation du code. En effet, cela permet d'utiliser les mêmes librairies (comme la librairie standard) dans des programmes ayant des politiques de sécurité différentes, sans pour autant avoir à en définir des versions spécialisées ou à les recompiler pour chaque politique rencontrée.

Le système Flow Caml fournit un outil, `flowcamlpol`, qui permet de calculer la politique de sécurité minimale pour laquelle un programme tout entier a été vérifié. La commande `flowcamlpol` a un usage similaire à celle d'un assembleur de fichiers objets : elle doit être invoquée avec la liste des noms des interfaces compilées des différentes unités de compilation du programme, dans l'ordre dans lequel elles doivent être assemblées. La politique de sécurité respectée par le programme exemple est donnée comme suit :

```
flowcamlpol passwd.fcmi shadow.fcmi debug.fcmi main.fcmi
```

Cette commande donne un ensemble d'inégalités minimal entre principaux sous lequel le programme tout entier a été typé :

```
!shadow_file < !shadow_password
!shadow_file < !stdout
!passwd_file < !password
!passwd_file < !shadow_file
!password < !shadow_password
!arg < !stderr
!arg < !stdout
```



Une représentation graphique de ces inégalités peut être obtenue en invoquant `flowcamlpol` avec l'option `-graph`. La politique de sécurité est représentée par un graphe dont les nœuds sont les principaux et les arcs définissent la relation d'ordre, *i.e.* les flots d'information possibles. Pour le programme exemple, on remarque en particulier l'absence de chemin dans ce graphe des principaux `!password` et `!shadow_password` vers `!stdout` : cela montre que les informations imprimées sur la sortie standard ne dépendent pas des mots de passe stockés dans les fichiers `/etc/passwd` et `/etc/shadow`.

4.4.3 Déclarations `affects` et `raises`

L'exécution d'un programme consiste à évaluer successivement chacune de ses unités, dans l'ordre spécifié à la compilation. Une fois chacune de ses unités compilées, le programme exemple est assemblé par la commande suivante :

```
ocamlc -o passwd2shadow passwd.cmo shadow.cmo verbose.cmo main.cmo
```

L'exécution du binaire `passwd2shadow` évalue le corps des unités `Passwd`, `Shadow`, `Verbose` puis `Main`, jusqu'à ce qu'une exception non rattrapée soit rencontrée ou la fin du programme atteinte. Ainsi, lorsqu'elle prend le contrôle, chaque unité reçoit l'information que les précédentes ont terminé normalement. Pour prendre en compte ces possibles flots d'information, il faut comparer les bornes des structures sous-jacentes, comme si le programme était défini dans une seule unité comme

```
module Passwd = struct
  ...
end

module Shadow = struct
  ...
end

module Verbose = struct
  ...
end

module Main = struct
  ...
end
```

Dans ce but, chaque fichier interface doit mentionner, si elles ne sont pas vides, les bornes inférieures et supérieures de la structure définie dans le fichier implémentation correspondant, grâce aux déclarations **`affects`** et **`raises`** qui doivent apparaître dans son en-tête. Lorsque l'implémentation est typée, les bornes sont inférées et comparées à celles données dans l'interface. Par exemple, l'interface de l'unité `Verbose` déclare les bornes suivantes :

```
affects !arg
raises !arg
```

qui signifient que l'unité `Verbose` a des effets de bord de niveau `!arg` et peut lever des exceptions de ce même niveau.

Lors de l'assemblage d'une série d'unité de compilations $\texttt{Unit}_1$, ..., $\texttt{Unit}_n$, il faut vérifier que, pour chaque unité $\texttt{Unit}_{i+1}$, les bornes supérieures de $\texttt{Unit}_1$, ..., $\texttt{Unit}_i$ sont inférieures ou égales à la borne inférieure de $\texttt{Unit}_{i+1}$. Cela est effectué par la commande `flowcamlpol`, en même temps qu'elle calcule la politique de sécurité :

```
flowcamlpol passwd.fcmi shadow.fcmi debug.fcmi main.fcmi
```

C'est la raison pour laquelle les unités de compilation doivent lui être données dans le même ordre que pour l'assemblage du programme.

Un exemple complet

passwd.fmli

```
flow !passwd_file < !password

(* An entry of "/etc/passwd" is represented by a record of
   type [(!passwd_file, !password) entry] *)
type ('a, 'b) entry =
    { login: 'a string;
      password: 'b string
    }

(* Input from "/etc/passwd" *)
type (#'a:level) in_channel
val open_in: unit -{!passwd_file ||}-> !passwd_file in_channel
val input_entry: [< !passwd_file] in_channel
   -{!passwd_file | End_of_file: !passwd_file |}->
   (!passwd_file, !password) entry
val close_in: [< !passwd_file] in_channel -{!passwd_file ||}-> unit
```

passwd.ml

```
type entry =
    { login: string;
      password: string
    }

type in_channel = Pervasives.in_channel

let open_in () =
  Pervasives.open_in "/etc/passwd"

let rec input_entry chan =
  let line = input_line chan in
  try
    let i1 = String.index line ':' in
    let i2 = String.index_from line (i1 + 1) ':' in
    { login = String.sub line 0 i1;
      password = String.sub line (i1 + 1) (i2 - i1 - 1)
    }
  with
    Not_found -> input_entry chan

let close_in chan =
  Pervasives.close_in chan
```

shadow.fmli

```
flow !shadow_file < !shadow_password

(* An entry of "/etc/shadow" is represented by a record of
   type [(!shadow_file, !shadow_password) entry] *)
type ('a, 'b) entry =
    { login: 'a string;
```

```
      password: 'b string;
      rem: 'b string;
    }

(* Input from "/etc/shadow" *)
type (#'a:level) in_channel
val open_in: unit -{!shadow_file ||}-> !shadow_file in_channel
val input_entry: !shadow_file in_channel
   -{!shadow_file | End_of_file: !shadow_file |}->
   (!shadow_file, !shadow_password) entry
val close_in: !shadow_file in_channel -{!shadow_file ||}-> unit

(* Output to "/etc/shadow" *)
type noneq out_channel
val open_out: unit -{!shadow_file ||}-> out_channel
val output_entry:
   out_channel -> (!shadow_file, !shadow_password) entry
   -{!shadow_file ||}-> unit
val close_out: out_channel -{!shadow_file ||}-> unit
```

shadow.ml

```
type entry =
    { login: string;
      password: string;
      rem: string;
    }

type in_channel = Pervasives.in_channel

let open_in () =
  Pervasives.open_in "/etc/shadow"

let rec input_entry chan =
  try
    let line = input_line chan in
    let i1 = String.index line ':' in
    let i2 = String.index_from line (i1 + 1) ':' in
    let ln = String.length line in
    { login = String.sub line 0 i1;
      password = String.sub line (i1 + 1) (i2 - i1 - 1);
      rem = String.sub line (i2 + 1) (ln - i2 - 1)
    }
  with
    Not_found -> input_entry chan

let close_in chan =
  Pervasives.close_in chan

type out_channel = Pervasives.out_channel

let open_out () =
  Pervasives.open_out "/etc/shadow"

let output_entry chan e =
  Printf.fprintf chan "%s:%s:%s\n" e.login e.password e.rem
```

```
let close_out chan =
  Pervasives.close_out chan
```

verbose.fmli

```
flow !arg < !stderr
and  !arg < !stdout


affects !arg
raises !arg


val message : !stdout string -{!stdout ||}-> unit
```

verbose.fml

```
flow !arg < !stderr, !stdout

(** [!verbose_mode] is true if the verbose mode is
    active. *)
let verbose_mode : (!arg bool, _) ref = ref false

(** Parse command-line arguments.  If the option "-v" if
    found then [verbose_mode] is set to true.  If any other
    option is encountered then an error message is printed
    and the exception [Exit] is raised. *)
let _ =
  for i = 1 to Array.length Sys.argv - 1 do
    match Sys.argv.(i) with
      "-v" -> verbose_mode := true
    | option ->
        prerr_string "Invalid␣option␣";
        prerr_endline option;
        raise Exit
  done

(** [print message] print a message on the standard output
    if the verbose mode is enabled. Otherwise, it does
    nothing. *)
let message s =
  if !verbose_mode then print_endline s
```

main.fml

```
flow !passwd_file < !shadow_file
and  !passwd_file, !shadow_file < !stdout
and  !passwd_file, !shadow_file < !shadow_password
and  !password < !shadow_password

(** The module [StringMap] implements association tables
    indexed by strings. *)
module StringMap = Map.Make (struct
```

```
  type 'a t = 'a string
  let compare = Pervasives.compare
end)



(** [read_shadow ()] reads the content of /etc/passwd
    and returns a map associating each login to its
    entry. *)
let read_shadow () =

  let in_chan = Shadow.open_in () in

  let rec loop accu =
    try
      let entry = Shadow.input_entry in_chan in
      loop (StringMap.add entry.Shadow.login entry accu)
    with End_of_file ->
      Shadow.close_in in_chan;
      accu
  in

  loop StringMap.empty


(** [read_passwd shadow_map] generates /etc/shadow from
    /etc/passwd and the entries in [shadow_map] *)
let read_passwd shadow_map =

  let in_chan = Passwd.open_in ()
  and out_chan = Shadow.open_out () in

  let rec loop () =

    try

      let passwd_entry = Passwd.input_entry in_chan in
      Verbose.message passwd_entry.Passwd.login;

      let shadow_entry =
        try
          StringMap.find passwd_entry.Passwd.login shadow_map
        with
          Not_found ->
            Verbose.message "␣␣creating␣an␣entry";
            { Shadow.login = passwd_entry.Passwd.login;
              Shadow.password = passwd_entry.Passwd.password;
              Shadow.rem = ""
            }
      in

      Shadow.output_entry out_chan shadow_entry

    with

      End_of_file -> ()
```

```
  in

  loop ();

  Passwd.close_in in_chan;
  Shadow.close_out out_chan


let _ =
  let shadow_map = read_shadow () in
  read_passwd shadow_map
```

DEUXIÈME PARTIE

Une analyse typée de flots d'information pour ML

Cette deuxième partie formalise l'analyse typée de flots d'information mise en œuvre dans le système Flow Caml, pour un noyau du langage ML, que je dénomme Core ML. Il s'agit d'un λ-calcul en appel par valeur muni de polymorphisme « let », d'exceptions et de constantes. J'ai cherché à donner une présentation du langage et du système de types aussi modulaire que possible en laissant ouverte la définition de ses constructeurs et destructeurs. Je montre toutefois comment le langage peut être muni — par un choix approprié de ces constantes — de fonctionnalités usuelles : arithmétique, références, structures de données.

*Les systèmes de types sont généralement utilisés pour garantir la préservation d'un certain invariant lors de l'évaluation d'*une *expression, permettant ainsi d'établir* statiquement *des propriétés* dynamiques *des programmes. L'absence d'erreur à l'exécution est certainement l'exemple le plus habituel de telle propriété. Le système que je présente ici établit quant à lui un résultat de* non-interférence *[GM82]. Cependant, l'énoncé et la preuve d'une telle propriété nécessitent de considérer* deux *exécutions* a priori *indépendantes d'un même programme, initialisées avec des valeurs d'entrées différentes, pour montrer qu'elle produisent le même résultat. C'est pourquoi, son traitement est généralement plus délicat et des approches variées sont proposées dans la littérature.*

Abadi, Lampson et Lévy [ALL96] ont donné une sémantique opérationnelle étiquetée du λ-calcul, où la quotité d'information *portée par les expressions est indiquée par des étiquettes qui leur sont attachées. L'exécution d'un programme dans une telle sémantique permet d'effectuer une analyse de flots d'information dynamique. Pottier et Conchon [PC00] ont ensuite montré comment une analyse statique typée pouvait être dérivée et prouvée correcte à partir de cette sémantique. Toutefois, dans une langage autorisant, comme Core ML, l'évaluation des expressions à produire des effets de bords, un flot d'information peut avoir pour origine l'*absence *d'un certain effet. Considérons par exemple le fragment de code suivant :*

$$\mathsf{if}\ x = 1\ \mathsf{then}\ y := 1$$

Si, après l'évaluation de cette expression, la référence y ne contient pas l'entier 1 *alors x ne peut être l'entier* 1 *lui-même. Dans ce cas, l'exécution transfère de l'information de x vers la référence y bien que le contenu de cette dernière ne soit pas modifié. Il semble difficile d'étendre une sémantique étiquetée, telle que celle d'Abadi, Lampson et Lévy, pour capturer ce type de phénomène. Cela m'amène à envisager une autre approche.*

Les preuves de non-interférence directes*, bien que relativement aisées dans le cas de langages de programmation simples [VSI96], deviennent de plus en plus complexes en présence de traits avancés dans le langage étudié — tels que l'allocation mémoire, les mécanismes d'exceptions ou les clôtures — ou bien dans le système de type utilisé — tels que le polymorphisme. En effet, elles peuvent être vue comme des preuves de* bisimulation *et nécessitent ainsi la considération d'invariants complexes, comme par exemples ceux manipulés par Zdancewic et Myers [ZM01a, ZM02].*

Pour contourner ces difficultés, j'ai décomposé mon approche en plusieurs étapes indépendantes. Tout d'abord, dans le chapitre 5 (page 97), je définis une extension ad hoc *particulière de Core ML, dénommée Core ML*^{2}*, qui permet de raisonner explicitement sur les points communs et les différences entre* deux *configurations d'un programme. Je prouve que cette extension est correcte dans un certain sens en donnant un énoncé de* simulation *(théorème 5.13, page 110). Ensuite, le chapitre 6 (page 113) présente un système de type pour ce langage étendu (et par là même pour Core ML qui en est un sous-ensemble). Je montre alors une propriété standard de stabilité du typage par réduction (théorème 6.17, page 128). La propriété de non-interférence pour le langage de base (théorème 6.21, page 134) est obtenue alors de manière élémentaire en combinant les deux résultats précédents. En d'autres termes, j'ai réduit le problème initial de* non-interférence *pour Core ML en une propriété de préservation du typage pour Core ML*^{2}*, grâce à l'expression de l'invariant de bisimulation dans le système de types lui-même.*

Dans la tradition de ML, le système de type est équipé de polymorphisme « let *» et d'un algorithme de synthèse des types. Ces deux caractéristiques se révèlent fondamentales pour son utilisation pratique : outre la forme des valeurs produites, les types associés aux expressions décrivent les effets produits par leur exécution et la quotité d'information attachée au résultat. Ainsi, le polymorphisme permet d'écrire du code générique vis-à-vis de chacune de ces trois composantes, évitant par exemple la duplication d'une fonction devant être utilisée avec des données de niveaux différents. L'inférence de types permet au programmeur de vérifier son code sans l'annoter, ce qui ne serait pas réaliste étant donné la verbosité et la complexité des types. Le système est également équipé de* sous-typage*, ce qui permet d'obtenir une description précise car* orientée *des flots d'information. Il est par conséquent à base de contraintes. Cependant, si j'établissais le résultat de non-interférence directement sur ce système, il me faudrait prendre en compte les mécanismes syntaxiques de généralisation et d'instanciation, ainsi que la gestion des contraintes, au sein même de la preuve relative à la correction de l'analyse. Cette dernière serait très certainement compliquée et les points nouveaux relatifs à l'analyse de flots obscurcis par des problèmes techniques, comme la gestion des noms, mais bien connus. Pour contourner cette dernière difficulté, j'ai adopté l'approche*

semi-syntaxique *de Pottier [Pot01a], qui consiste en deux étapes. Tout d'abord, dans le chapitre 6 (page 113), j'étudie un système de types équipé d'une forme extensionelle de polymorphisme, dont le traitement est très simple puisqu'il ne fait intervenir que des types bruts, c'est-à-dire des types sans variables. Ensuite, dans le chapitre 7 (page 135), je construis un système à base de contraintes, dans le style de HM(X) [OSW99], pourvu d'un algorithme d'inférence. Je prouve la correction de ce système en traduisant ses jugements dans le précédent.*

Les résultats présentés dans cette partie ont été co-publiés avec François Pottier, en anglais, dans les actes de la conférence *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002)* [PS02] puis dans le journal *ACM Transactions on Programming Languages and Systems (TOPLAS)* [PS03].

CHAPITRE 5 CINQ

Core ML et Core ML^2 Syntaxe et sémantique

5.1 Le langage Core ML

5.1.1 Présentation

La définition du langage Core ML est paramétrée par deux ensembles disjoints et dénombrables de ***constructeurs*** k et de ***destructeurs*** (ou ***primitives***) f. J'utilise le nom *constante* pour désigner indistinctement un constructeur ou un destructeur. La plus grande partie du développement de cette thèse est indépendant de leur spécification ; je montrerai toutefois comment constructeurs et destructeurs permettent de munir Core ML de *constantes et opérations arithmétiques*, *références*, *sommes* et *produits* ainsi que de *primitives génériques*.

Je suppose donnés des ensembles infinis et dénombrables de ***variables de programme*** x, d'***adresses mémoire*** m et de ***noms d'exceptions*** ξ. Les ensembles, finis ou co-finis, de noms d'exceptions sont notés Ξ. La syntaxe des ***valeurs***, ***résultats***, ***expressions***, ***contextes d'évaluation*** et ***clauses*** de Core ML est définie comme suit :

$$
\begin{array}{lll}
v ::= x \mid m \mid \lambda x.e \mid k\, v_1 \cdots v_{\mathrm{a}(k)} & & \text{(valeur)} \\
a ::= v \mid \mathsf{raise}\, \xi\, v & & \text{(résultat)} \\
e ::= a \mid v_1\, v_2 \mid f\, v_1 \cdots v_{\mathrm{a}(f)} \mid \mathsf{let}\, x = v\, \mathsf{in}\, e \mid \mathbb{E}[e] & & \text{(expression)} \\
\mathbb{E} ::= \mathsf{bind}\, x = [\,]\, \mathsf{in}\, e \mid [\,]\, \mathsf{handle}\, \vec{h} \mid [\,]\, \mathsf{finally}\, e & & \text{(contexte d'évaluation)} \\
h ::= \Xi\, x \to e \mid \Xi\_ \to e \mid \Xi\, x \to e\, \mathsf{pgt} \mid \Xi\_ \to e\, \mathsf{pgt} & & \text{(clause)}
\end{array}
$$

Les valeurs comprennent les variables, les adresses mémoires, les λ-abstractions et les applications totales de constructeurs. Par convention, les adresses mémoires n'apparaissent pas dans les programmes sources entrés par le programmeur : elles sont introduites lors de la réduction, quand de nouveaux blocs sont alloués en mémoire.

Un résultat représente l'aboutissement de l'évaluation d'un programme : il s'agit soit d'une valeur soit d'une exception non rattrapée, de la forme $\mathsf{raise}\, \xi\, v$ où ξ est le nom de l'exception et v son argument. Une expression peut être un résultat, une application d'une fonction à un argument, une application totale de destructeur, une construction let ou une autre expression

placée dans un contexte d'évaluation. Il faut noter que les deux membres d'une application de fonction, ainsi que les arguments passés à un destructeur, doivent être — selon la syntaxe de Core ML — des valeurs, et non des expressions arbitraires : cette restriction syntaxique, qui est dérivée des *formes A-normales* de Flanagan, Sabry, Duba et Felleisen [FSDF93], offre plusieurs avantages. Tout d'abord, elle permet une formulation plus simple du système de types : en effet, puisque les valeurs sont des formes normales, les seuls effets résultant de l'évaluation d'une application sont ceux produits par la fonction ou la primitive elle-même. D'autre part, cette restriction me permet de rester indépendant de la stratégie d'évaluation adoptée (de gauche à droite ou de droite à gauche par exemple). Cependant, elle ne réduit pas l'expressivité de Core ML : les programmes habituels, écrits dans une syntaxe plus libérale, peuvent, dès lors qu'une stratégie d'évaluation est choisie, être traduits dans cette syntaxe. Je reviendrai sur ce point à la section 9.1 (page 159).

La construction $\mathsf{let}\, x = v \;\mathsf{in}\; e$ a la même sémantique que l'application $(\lambda x.e)\, v$; cependant, comme il est usuel pour ML [WF94], elle indique au vérificateur de types de généraliser le type attribué à x avant de considérer e. Suivant l'approche de Wright [Wri93, Wri95], j'impose que l'expression liée à x soit une valeur afin de préserver la correction du typage en présence de traits impératifs. On pourrait naturellement étendre sans difficulté la généralisation des types à une classe plus large d'expressions *non-expansives* [Tof88].

Les contextes d'évaluation permettent d'assembler les expressions en spécifiant leur ordre d'évaluation. L'expression $\mathsf{bind}\, x = e_1 \;\mathsf{in}\; e_2$ évalue e_1, lie la variable x à sa valeur et évalue e_2. Malgré leur ressemblance, les rôles des constructions let et bind sont bien distincts : la première ne peut lier que des valeurs mais permet la généralisation, alors que la seconde permet d'exprimer la séquentialité. J'écris $e_1; e_2$ pour $\mathsf{bind}\, x = e_1 \;\mathsf{in}\; e_2$ quand x n'apparaît pas dans e_2. Les autres contextes d'évaluation fournissent différents moyens pour rattraper les exceptions. Le contexte $[\,]\;\mathsf{handle}\;\vec{h}$ est similaire à la construction try . . . with du langage Caml [LDG+b]. Elle comporte une liste de clauses $h_1, \ldots, h_n$. Chaque clause h_i rattrape un ensemble de noms d'exceptions handled(h_i), qui est naturellement l'ensemble Ξ indiqué à sa tête. Si l'expression placée dans le trou d'un contexte handle se réduit en une valeur, ce contexte n'a naturellement pas d'effet. Sinon, l'exception levée, $\mathsf{raise}\, \xi\, v$, est confrontée aux clauses. L'exécution se poursuit alors avec une des clauses h_i qui attrape le nom d'exception ξ (*i.e.* telle que $\xi \in \text{handled}(h_i)$) — s'il en existe une. Sinon, dans le cas où l'exception n'est attrapée par aucune des clauses, elle est tout simplement propagée. Je note ainsi escape($h_1 \cdots h_n$) l'ensemble des noms d'exceptions propagés par le contexte $[\,]\;\mathsf{handle}\;(h_1 \cdots h_n)$, c'est-à-dire le complémentaire de l'ensemble handled(h_1) $\cup \cdots \cup$ handled(h_n).

Quatre sortes de clauses, correspondant à quatre sémantiques différentes, sont disponibles. Les formes $\Xi\, x \to e$ et $\Xi\, x \to e\,\mathsf{pgt}$ lient la variable x à l'argument de l'exception rattrapée, v, avant d'évaluer e. Comme en Caml, le typage restreint leur usage aux cas où toutes les exceptions mentionnées dans Ξ ont le *même* type. Dans les autres situations, on ne peut utiliser que les formes $\Xi\_ \to e$ et $\Xi\_ \to e\,\mathsf{pgt}$ qui ne permettent pas d'accéder à l'argument de l'exception rattrapée. Orthogonalement, l'évaluation des clauses $\Xi\, x \to e$ et $\Xi\_ \to e$ retourne le résultat produit par e, alors que celle de $\Xi\, x \to e\,\mathsf{pgt}$ et $\Xi\_ \to e\,\mathsf{pgt}$ propage l'exception initialement rattrapée, en la levant après l'évaluation de e (le mot-clef pgt est une abréviation de l'anglais *propagate*). Notons enfin que l'ensemble Ξ des exceptions attrapées par une clause peut pour l'instant être une partie finie ou co-finie arbitraire de $\mathcal{E}$. Au chapitre 6 (page 113), je supposerai Ξ *fini* dans les clauses des formes $\Xi\, x \to e$ et $\Xi\, x \to e\,\mathsf{pgt}$, pour formuler l'algorithme d'inférence de manière simple. Cette restriction correspond aux possibilités de la construction try . . . with du langage Caml.

Le contexte $[\,]\;\mathsf{finally}\;e$ évalue l'expression à la place de $[\,]$ puis, quelle que soit la forme du résultat, poursuit en réduisant e. Si e lève une exception, celle-ci est propagée, sinon la valeur produite par e est mise de côté, et le résultat de la première expression retourné. Ce contexte, qui n'est pas fourni primitivement dans les implémentations actuelles de ML [sml, LDG+b] est similaire aux constructions unwind-protect de Lisp et try-finally de Java. En faisant de lui une construction de base du langage, je permets un typage plus fin de cet idiome reflétant le fait que l'expression e

Réductions de base

$$
\begin{array}{rcll}
(\lambda x.e)\, v \,/\, \mu & \to & e[x \leftarrow v] \,/\, \mu & (\beta) \\
\mathsf{let}\, x = v \,\mathsf{in}\, e \,/\, \mu & \to & e[x \leftarrow v] \,/\, \mu & (\mathrm{let}) \\
f\, \vec{v} \,/\, \mu & \to & e \,/\, \mu \circledcirc \mu' & (\delta) \\
 & & \textit{si}\ f\, \vec{v} \,/\, \mu \multimap_f e + \mu' & \\
\mathbb{E}[a] \,/\, \mu & \to & \mathbb{E} \ll a \,/\, \mu & (\mathrm{pop})
\end{array}
$$

Réduction sous un contexte

$$
\begin{array}{rcll}
\mathbb{E}[e] \,/\, \mu & \to & \mathbb{E}[e'] \,/\, \mu' & (\mathrm{context}) \\
 & & \textit{si}\ e \,/\, \mu \to e' \,/\, \mu' &
\end{array}
$$

Figure 5.1 – Sémantique opérationelle de Core ML

est *toujours* évaluée. Je reviendrai sur cela à la section 6.3.2 (page 117).

La nécessité d'introduire autant de constructions différentes pour rattraper les exceptions tient également à la décision de faire des noms d'exceptions des entités de seconde classe du langage : une exception n'est pas une valeur, de telle sorte qu'une variable ne peut être liée à une exception. Ainsi, afin de limiter la perte d'expressivité au minimum, des contextes couvrant les idiomes les plus courants ont dû être ajoutés. Je motiverai précisément ce choix dans la section 6.3.2 (page 117) et le discuterai dans la section 9.2 (page 160).

Les variables sont liées dans les valeurs, expressions, résultats et clauses de Core ML suivant les règles usuelles : dans $\lambda x.e$, $\mathsf{let}\, x = v \,\mathsf{in}\, e$, $\mathsf{bind}\, x = e' \,\mathsf{in}\, e$, $\Xi\, x \to e$ et $\Xi\, x \to e\, \mathsf{pgt}$, la variable x est liée à l'intérieur de e. Les adresses mémoire ne sont jamais liées dans les expressions. Les ensembles des variables libres et adresses mémoires libres d'une expression e sont respectivement notés $\mathrm{fpv}(e)$ et $\mathrm{fml}(e)$. Une expression e est *close* si et seulement si $\mathrm{fpv}(e) = \varnothing$.

5.1.2 Sémantique opérationnelle

Un ***bloc mémoire*** w est une valeur close ou la constante null.

$$w ::= v \mid \mathsf{null} \qquad (\text{bloc mémoire})$$

Un ***état mémoire*** μ est une fonction totale des adresses mémoires vers les blocs mémoires. Il représente dans la sémantique de Core ML la notion de *tas*, c'est à dire une collection de structures, chacune allouée à une adresse particulière de la mémoire et pouvant contenir des pointeurs vers d'autres entrées du tas. La constante null correspond naturellement à un bloc non alloué : on dit que m est *allouée* dans μ si et seulement si $\mu(m) \neq \mathsf{null}$. On note $\mathrm{fml}(\mu)$ l'ensemble des adresses mémoires qui apparaissent dans l'image de μ, c'est-à-dire $\cup_m \mathrm{fml}(\mu(m))$. Un état mémoire est *bien formé* si et seulement si seul un nombre fini d'adresses mémoire sont allouées et si toutes les adresses mentionnées dans ses blocs sont allouées, *i.e.* pour tout $m \in \mathrm{fml}(\mu)$, $\mu(m) \neq \mathsf{null}$.

Puisque nous souhaitons pouvoir munir Core ML de primitives effectuant des effets de bord, c'est-à-dire allouant, modifiant et lisant des entrées du tas, la sémantique opérationnelle du langage doit être définie comme une relation de réécriture entre configurations : une ***configuration*** est une paire formée d'une expression e et d'un état mémoire μ, notée $e \,/\, \mu$.

La figure 5.1 définit la relation de réduction, notée $\to$, entre configurations Core ML. On dit que $e \,/\, \mu$ se *réduit* en $e' \,/\, \mu'$ si et seulement si $e \,/\, \mu \to e' \,/\, \mu'$ est dérivable. Les règles du premier groupe définissent la sémantique des constructions de base du langage : (β) et (let) réduisent respectivement les applications dont le membre gauche est une λ-abstraction et les constructions let. Je note $e[x \leftarrow v]$ l'expression obtenue en substituant toutes les occurrences libres de x par v dans e. L'ensemble des constantes n'étant pas fixé, leur sémantique doit rester abstraite : elle est décrite par la donnée, pour chaque primitive f, d'une relation $-_f\!\to$. Si la primitive f a une arité n,

Contextes

$$
\begin{aligned}
(\mathsf{bind}\,[\,] = x\ \mathsf{in}\ e) \ll v &= e[x \leftarrow v] \\
(\mathsf{bind}\,[\,] = x\ \mathsf{in}\ e) \ll \mathsf{raise}\,\xi\,v &= \mathsf{raise}\,\xi\,v \\
([\,]\ \mathsf{handle}\ (h_1 \cdots h_n)) \ll v &= v \\
([\,]\ \mathsf{handle}\ (h_1 \cdots h_n)) \ll \mathsf{raise}\,\xi\,v &= h_i \ll \mathsf{raise}\,\xi\,v \\
&\quad si\ \xi \in \mathrm{handled}(h_i) \\
([\,]\ \mathsf{handle}\ (h_1 \cdots h_n)) \ll \mathsf{raise}\,\xi\,v &= \mathsf{raise}\,\xi\,v \\
&\quad si\ \xi \in \mathrm{escape}(h_1 \cdots h_n) \\
([\,]\ \mathsf{finally}\ e) \ll a &= e; a
\end{aligned}
$$

Clauses

$$
\begin{aligned}
(\Xi\, x \to e) \ll \mathsf{raise}\,\xi\,v &= e[x \leftarrow v] \\
&\quad si\ \xi \in \Xi \\
(\Xi\, \_ \to e) \ll \mathsf{raise}\,\xi\,v &= e \\
&\quad si\ \xi \in \Xi \\
(\Xi\, x \to e\ \mathsf{pgt}) \ll \mathsf{raise}\,\xi\,v &= e[x \leftarrow v]; \mathsf{raise}\,\xi\,v \\
&\quad si\ \xi \in \Xi \\
(\Xi\, \_ \to e\ \mathsf{pgt}) \ll \mathsf{raise}\,\xi\,v &= e; \mathsf{raise}\,\xi\,v \\
&\quad si\ \xi \in \Xi
\end{aligned}
$$

Figure 5.2 – Évaluation des contextes et des clauses

un jugement de $-f\!\!\rightarrow$ est de la forme $f\,v_1 \cdots v_n \,/\, \mu \,-f\!\!\rightarrow e + \dot{\mu}$, où $v_1, \ldots, v_n$ sont des valeurs closes. Il doit être lu : *dans l'état mémoire μ, l'application $f\,v_1 \cdots v_n$ a pour résultat e et effectue le(s) effet(s) de bord $\dot{\mu}$*. Cette interprétation est formalisée par la règle (δ). La meta-variable $\dot{\mu}$ dénote un ***fragment d'état mémoire***, qui est une fonction partielle des adresses mémoires vers les valeurs closes. Il représente les *modifications* produites sur l'état mémoire μ par l'appel de la primitive f, c'est-à-dire les allocations et les écritures effectuées. Ainsi, $\mu \circledcirc \dot{\mu}$ est l'état mémoire μ' défini par $\mu'(m) = \dot{\mu}(m)$ si $m \in \mathrm{dom}(\dot{\mu})$ et $\mu'(m) = \mu(m)$ sinon. Le lecteur peut naturellement se demander pourquoi je n'ai pas suivi une approche plus habituelle en définissant $-f\!\!\rightarrow$ comme une relation entre configurations (croissante pour l'ensemble des adresses mémoire allouées) et remplacé (δ) par la règle suivante :

$$\frac{f\,v_1 \cdots v_n \,/\, \mu \,-f\!\!\rightarrow e \,/\, \mu'}{f\,v_1 \cdots v_n \,/\, \mu \rightarrow e \,/\, \mu'}$$

La raison apparaîtra avec la définition de la sémantique de Core ML^2 (section 5.2.3, page 105) qui se trouve être simplifiée par ce choix technique. Enfin, la réduction des primitives ne doit pas être influencée par les noms arbitraires des adresses mémoires dans les configurations et doit préserver la bonne formation des configurations, comme exprimé par l'hypothèse suivante.

Hypothèse 5.1 *Les relations $-f\!\!\rightarrow$ sont stables par renommage des adresses mémoires (*i.e. *si $f\,\vec{v}\,/\,\mu \,-f\!\!\rightarrow\, e + \dot{\mu}$ alors, pour tout renommage ϕ des adresses mémoires, on a $f\,\phi\vec{v}\,/\,\phi\mu\phi^{-1} \,-f\!\!\rightarrow\, \phi e + \phi\dot{\mu}\phi^{-1}$). Si $f\,\vec{v}\,/\,\mu \,-f\!\!\rightarrow e + \dot{\mu}$ alors, pour tout $m \in \mathrm{fml}(e, \dot{\mu})$, on a $m \in \mathrm{dom}(\dot{\mu})$ ou $\mu(m) \neq \mathsf{null}$.* □

La règle (pop) élimine un contexte qui contient un résultat. L'expression obtenue est donnée par la relation $\ll$, définie figure 5.2, qui précise comment chaque forme de résultat est attrapée ou propagée par chaque contexte d'évaluation. La règle (context) termine la définition de $\rightarrow$ en autorisant la réduction à avoir lieu sous un contexte d'évaluation arbitraire.

Je souhaite montrer que la sémantique de Core ML est déterministe, c'est-à-dire qu'étant donnée $e \,/\, \mu$, il existe au plus une configuration $e' \,/\, \mu'$, à un renommage des adresses mémoire près, telle que $e \,/\, \mu \to e' \,/\, \mu'$. Ce choix naturel pour un langage séquentiel est en fait essentiel pour la non-interférence : dans le cadre d'une sémantique non-déterministe, il faut en effet considérer qu'un programme produit un *ensemble* de résultats, chacun d'entre eux étant éventuellement affecté d'une certaine probabilité. Une propriété de non-interférence devrait alors garantir que deux exécutions d'un même programme donnent les mêmes résultats avec les mêmes probabilités.

La formulation actuelle des règles de réduction présente deux sources potentielles de non-déterminisme. La première concerne les primitives du langage, je dois en effet m'assurer que la sémantique qui leur est attribuée par les relations $-f\!\to$ est bien déterministe :

Hypothèse 5.2 (Déterminisme des primitives) *Si $f\,\vec{v} \,/\, \mu \,-f\!\to e_1 + \dot{\mu}_1$ et $f\,\vec{v} \,/\, \mu \,-f\!\to e_2 + \dot{\mu}_2$ alors les paires $e_1 + \dot{\mu}_1$ et $e_2 + \dot{\mu}_2$ coïncident, modulo un renommage des adresses mémoires non allouées dans μ.* □

La deuxième source de non-déterminisme apparaît avec la réduction des constructions handle où un choix peut intervenir entre plusieurs clauses rattrapant la même exception. Pour que cette situation n'advienne pas, je supposerai que dans chaque contexte $[\,]$ handle $(h_1 \cdots h_n)$, les clauses $h_1, \ldots, h_n$ rattrapent des ensembles d'exceptions deux à deux disjoints, *i.e.* pour tous $j_1, j_2 \in [1, n]$ si $j_1 \neq j_2$ alors $\text{handled}(h_{j_1}) \,\#\, \text{handled}(h_{j_2})$. On pourrait également préciser la définition de $\ll$ afin que les clauses soient considérées dans un ordre prédéterminé, comme dans la construction try . . . with de Caml. Cependant, cette possibilité se ramène de manière systématique à l'hypothèse précédente en précisant les ensembles de noms d'exceptions effectivement attrapées par chaque clause.

Lemme 5.3 *Toute configuration de la forme $a \,/\, \mu$ est une forme normale pour $\to$.* □

⌜ *Preuve.* Par inspection des règles de la figure 5.1 (page 99), on vérifie que si $e \,/\, \mu$ est réductible alors e est soit une application de fonction ou de constante (règles (β) et (δ)), soit une construction let (règle (let)), soit une expression de la forme $\mathbb{E}[e']$ (règles (pop) et (context)). Ces formes d'expressions n'étant pas des résultats, on en déduit que toute configuration $a \,/\, \mu$ est irréductible. ⌟

Lemme 5.4 (Déterminisme) *La sémantique opérationnelle de Core ML est déterministe : si $e/\mu \to e_1/\mu_1$ et $e/\mu \to e_2/\mu_2$ alors les configurations e_1/μ_1 et e_2/μ_2 sont identiques, modulo un renommage des adresses mémoires non allouées dans μ.* □

⌜ *Preuve.* On procède par induction sur les dérivations de $e \,/\, \mu \to e_1 \,/\, \mu_1$ et $e \,/\, \mu \to e_2 \,/\, \mu_2$.

∘ *Si l'une de ces dérivations se termine par une instance de (context).* On peut supposer, sans perte de généralité, qu'il s'agit de la première. L'expression e est de la forme $\mathbb{E}[e']$ et la prémisse de (context) est $e' \,/\, \mu \to e'_1 \,/\, \mu_1$ **(1)**. Par le lemme 5.3, on en déduit que e' n'est pas une valeur. On vérifie alors que la configuration $\mathbb{E}[e'] \,/\, \mu$ ne peut être réduite que par application de (context), de telle sorte qu'il existe e'_2 tel que $e_2 = \mathbb{E}[e'_2]$ et $e' \,/\, \mu \to e'_2 \,/\, \mu_2$ **(2)**. Par l'hypothèse d'induction appliquée à (1) et (2), les configurations $e'_1 \,/\, \mu_1$ et $e'_2 \,/\, \mu_2$ sont identiques, modulo un renommage des adresses mémoires n'apparaissant pas dans μ. La configuration $\mathbb{E}[e'] \,/\, \mu$ étant bien formée, les adresses mémoires libres dans $\mathbb{E}$ sont allouées dans μ. On en déduit que le renommage reliant e'_1/μ_1 à $e'_2 \,/\, \mu_2$ n'affecte pas le contexte $\mathbb{E}$, ce qui permet de conclure.

∘ *Autres cas.* On vérifie tout d'abord que la configuration $e \,/\, \mu$ ne peut être réduite que par une seule règle parmi les règles restantes, celle-ci étant déterminée par la forme de l'expression e : (β) réduit une application de fonction, (let) une construction let, (δ) une application de destructeur et (pop) un contexte. Il reste à prouver que la configuration produite par l'application de chacune des règles précédentes (c'est-à-dire celle mentionnée dans le membre droit de la règle) est déterminée par la configuration initiale (celle du membre gauche), modulo un renommage des adresses mémoire

non initialement allouées. Cette propriété est immédiate pour les règles (β) et (let). Pour (δ), elle découle de l'hypothèse 5.2. Pour (pop), elle est assurée par la disjonction des ensembles d'exceptions attrapées par chaque clause d'un contexte handle, qui fait de $\ll$ une fonction totale. ⌟

5.1.3 Constantes

Pour conclure cette présentation de Core ML, il me reste à introduire quelques exemples de constructeurs et destructeurs dont le langage peut être muni. Je donne ici leur sémantique et poursuivrai leur traitement tout au long du développement de ce chapitre, puis des chapitres 6 et 7, afin d'illustrer chacune des notions introduites. D'autres exemples, dont le traitement est plus conséquent, seront introduits au chapitre 8 (page 149).

▶ **Entiers relatifs** Pour chaque entier relatif n, on peut supposer l'existence d'un constructeur d'arité 0, noté $\widehat{n}$. On peut ensuite introduire les opérateurs arithmétiques habituels sous forme de primitives. Par exemple, la primitive binaire $\widehat{+}$ correspond à l'addition. Son application est notée de manière infixe : $v_1 \mathbin{\widehat{+}} v_2$ représente l'expression $\widehat{+}\, v_1\, v_2$. Sa sémantique peut être définie par :

$$\widehat{n_1} \mathbin{\widehat{+}} \widehat{n_2} \,/\, \mu \longrightarrow \widehat{n_1 + n_2} + \varnothing \qquad \text{(add)}$$

Le membre gauche est l'application binaire du destructeur $\widehat{+}$ aux constructeurs $\widehat{n_1}$ et $\widehat{n_2}$ tandis que le membre droit est le constructeur $\widehat{n}$ tel que n est la somme des entiers n_1 et n_2.

▶ **Références** La présence d'un état mémoire dans les configurations de la sémantique opérationnelle est essentiellement justifiée par un exemple particulièrement important, la manipulation des références. On définit deux primitives unaires ref et ! pour l'allocation et la lecture d'adresses mémoires ainsi qu'une primitive binaire := pour la modification en place de leur contenu. Soit () un constructeur d'arité 0, la constante *unit*. La sémantique opérationnelle de ces primitives est définie par les trois règles suivantes :

$$\mathsf{ref}\ v \,/\, \mu \longrightarrow m + \{m \mapsto v\} \quad \textit{si } \mu(m) = \mathsf{null} \qquad \text{(ref)}$$

$$m := v' \,/\, \mu[m \mapsto v] \longrightarrow () + \{m \mapsto v'\} \qquad \text{(assign)}$$

$$!\, m \,/\, \mu[m \mapsto v] \longrightarrow v + \varnothing \qquad \text{(deref)}$$

Utilisées comme prémisse de (δ), elles donnent les réductions suivantes entre configurations :

$$\mathsf{ref}\ v \,/\, \mu \to m \,/\, \mu[\![m \mapsto v]\!] \quad \textit{si } \mu(m) = \mathsf{null}$$

$$m := v' \,/\, \mu[\![m \mapsto v]\!] \to () \,/\, \mu[\![m \mapsto v']\!]$$

$$!\, m \,/\, \mu[\![m \mapsto v]\!] \to v \,/\, \mu[\![m \mapsto v]\!]$$

L'évaluation de $\mathsf{ref}\ v$ alloue une nouvelle adresse mémoire, distincte de celles déjà allouées, et lui associe la valeur v. Par (assign), l'évaluation de $m := v'$ modifie la valeur liée à l'adresse m : cette adresse doit déjà être définie dans l'état mémoire, mais cela est assuré par la bonne formation de la configuration initiale. Elle retourne la constante (). Enfin, $!\, m$ retourne la valeur liée à l'adresse m dans l'état mémoire, sans modifier ce dernier.

▶ **Paires** On introduit un constructeur binaire $(\cdot,\cdot)$. Si v_1 et v_2 sont des valeurs, la double application $(\cdot,\cdot)\, v_1\, v_2$ est notée (v_1, v_2). Pour accéder aux composantes d'une paire, on considère deux destructeurs proj_1 et proj_2 appelés *projection gauche* et *projection droite*, respectivement. Leur sémantique est naturellement définie par, pour $j \in \{1, 2\}$:

$$\mathsf{proj}_j\ (v_1, v_2) \,/\, \mu \longrightarrow v_j + \varnothing \qquad \text{(proj)}$$

▶ **Sommes binaires** Symétriquement, on peut équiper Core ML de sommes binaires en introduisant deux constructeurs unaires, inj_1 et inj_2, appelés *injection gauche* et *injection droite*, respectivement, et un destructeur ternaire case dont la sémantique est définie par la règle suivante, où $j \in \{1, 2\}$:

$$(\mathsf{inj}_j\, v)\ \mathsf{case}\ v_1\, v_2 \,/\, \mu \longrightarrow_{\mathsf{case}} v_j\, v \,/\, \mu \tag{case}$$

Les valeurs v_1 et v_2 sont typiquement des λ-abstractions et représentent les deux branches d'une construction case ou $\mathsf{match} \ldots \mathsf{with}$ habituelle.

Les Booléens peuvent naturellement être vus comme un cas particulier de somme binaire : on représente les constantes true et false respectivement par $\mathsf{inj}_1\,()$ et $\mathsf{inj}_2\,()$. La construction $\mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ peut est alors un sucre syntaxique pour $v\ \mathsf{case}\ (\lambda x.e_1)\,(\lambda x.e_2)$, où x est une variable qui n'est libre ni dans e_1 ni dans e_2. Il est aisé de vérifier que l'on a alors les deux réductions suivantes :

$$\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 \longrightarrow^* e_1$$
$$\mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 \longrightarrow^* e_2$$

▶ **Point fixe** Les constructions de base de Core ML ne permettent pas la récursion. Celle-ci peut être introduite à l'aide d'une primitive binaire fix. En ML, la sémantique de cette constante est définie de telle manière que l'application $\mathsf{fix}\, v_1\, v_2$ produit l'expression $v_1\,(\mathsf{fix}\, v_1)\, v_2$. Cependant, cette dernière forme n'est pas une expression valide Core ML, à cause des restrictions posées sur la syntaxe. Il est nécessaire d'effectuer une η-expansion de l'application partielle $\mathsf{fix}\, v_1$, puis d'introduire une liaison bind pour nommer le résultat de l'application de v_1 à son premier argument, ce qui donne la règle suivante :

$$\mathsf{fix}\, v_1\, v_2 \,/\, \mu \longrightarrow_{\mathsf{fix}} \mathsf{bind}\, x_1 = v_1\,(\lambda x_2.\mathsf{fix}\, v_1\, x_2)\ \mathsf{in}\ x_1\, v_2 \,/\, \mu \tag{fix}$$

(Il n'est pas nécessaire de faire d'hypothèse de fraîcheur sur les variables x_1 et x_2, puisque les valeurs v_1 et v_2 sont nécessairement closes.) La primitive fix est généralement utilisée avec une λ-abstraction comme premier argument. En particulier, la construction usuelle $\mathsf{let\ rec}\, f x = e_1\ \mathsf{in}\ e_2$, qui permet de définir une fonction f de manière récursive, peut être codée par l'expression $\mathsf{let}\, f = \lambda y.\mathsf{fix}\,(\lambda f.\lambda x.e_1)\, y\ \mathsf{in}\ e_2$ (avec $y \notin \mathrm{fpv}(e_1)$).

5.2 Le langage Core ML²

5.2.1 Présentation

Prouver une propriété de non-interférence nécessite de considérer simultanément deux programmes pour établir qu'ils partagent des sous-expressions tout au long de leur réduction. Afin de rendre ce raisonnement plus simple, je vais représenter ces deux programmes comme *une seule* expression d'un langage étendu, appelé Core ML², au lieu d'une paire d'expressions Core ML. Les valeurs, résultats et expressions de Core ML² sont obtenus en étendant la syntaxe de Core ML comme suit :

$$\begin{array}{lll} v ::= \ldots \mid \langle v \mid v \rangle & & \text{(valeur)} \\ a ::= \ldots \mid \langle a \mid a \rangle & & \text{(résultat)} \\ e ::= \ldots \mid \langle e \mid e \rangle & & \text{(expression)} \end{array}$$

L'expression Core ML² $\langle e_1 \mid e_2 \rangle$ permet de représenter la paire d'expressions (e_1, e_2). Il faut noter que les crochets peuvent apparaître à une profondeur arbitraire dans une expression. Par exemple, si v est une valeur Core ML alors les expressions $\langle v_1 \mid v_2 \rangle\, v$ et $\langle v_1\, v \mid v_2\, v \rangle$ représentent toutes les deux la paire $(v_1\, v, v_2\, v)$. Cependant, la première est plus précise que la seconde puisqu'elle explicite le fait que le nœud de l'application et son argument v sont partagés entre les deux expressions. L'imbrication de constructions $\langle \ldots \mid \ldots \rangle$ n'est pas autorisée, puisque cela n'aurait pas de sens vis-à-vis de l'interprétation donnée aux crochets : ainsi, les sous-expressions e_1 et e_2 d'une expression $\langle e_1 \mid e_2 \rangle$ doivent être des expressions du sous-ensemble Core ML. La correspondance entre Core

ML et Core ML² est rendue explicite par deux ***projections***, notées $\lfloor\cdot\rfloor_i$ où i est un élément de $\{1,2\}$. Ces deux fonctions vérifient $\lfloor\langle e_1 \mid e_2\rangle\rfloor_i = e_i$ et sont des homomorphismes sur toutes les autres constructions du langage. On étend la définition de $\mathrm{fpv}(e)$ et $\mathrm{fml}(e)$ aux expressions Core ML².

J'ai choisi de ne pas introduire pour Core ML² des notations distinctes de celles de Core ML, dans le but de ne pas alourdir la présentation : ainsi, je conserve les mêmes meta-variable v, a et e et réutiliserai plusieurs des notations introduites à la section 5.1.2 (page 99) pour définir la sémantique de Core ML². Cependant, aucune ambiguïté n'en résultera puisque je considère désormais Core ML comme un sous-ensemble de Core ML².

Avant de poursuivre, il est peut-être utile d'expliquer informellement comment Core ML² permet de garder trace des différences entre deux expressions Core ML tout au long de leur réduction. Considérons par exemple la fonction $\lambda x.\widehat{0}$. Évidemment, son résultat ne révèle aucune information sur son argument, puisqu'il s'agit d'une constante entière. Le système de types que je vais présenter au chapitre 6 (page 113) établira que cette fonction associe des résultats « publics » à des entrées « secrètes ». Dès lors, pour prouver que le système de types est correct, je dois établir un résultat de non-interférence : *pour tous entiers* n_1 *et* n_2*, les deux programmes* $(\lambda x.\widehat{0})\,\widehat{n}_1$ *et* $(\lambda x.\widehat{0})\,\widehat{n}_2$ *produisent la même valeur*. Pour cela, je représenterai cette paire de programmes à l'aide d'une expression Core ML², $e = (\lambda x.\widehat{0})\,\langle\widehat{n}_1 \mid \widehat{n}_2\rangle$; ses deux projections étant naturellement les deux programmes originaux : pour tout $i \in \{1,2\}$, $\lfloor e\rfloor_i$ est e_i. Notons que les arguments « secrets » $\widehat{n}_1$ et $\widehat{n}_2$ apparaissent dans e sous des crochets, alors que l'application de $\lambda x.\widehat{0}$ est partagée, *i.e.* apparaît en dehors des crochets. Suivant la sémantique de Core ML² que je vais décrire à la section 5.2.3, l'expression $(\lambda x.\widehat{0})\,\langle\widehat{n}_1 \mid \widehat{n}_2\rangle$ se réduit en la valeur Core ML² $\widehat{0}$. Le fait que ce résultat ne comporte aucune construction $\langle\cdot \mid \cdot\rangle$ assure que ses deux projections coïncident, c'est-à-dire que les programmes originaux e_1 et e_2 produisent tous les deux la même valeur. La preuve du théorème de non-interférence développée au chapitre 6 (page 113) est basée sur le même principe : je montrerai que, sous des hypothèses de typage appropriées, le résultat d'une séquence de réductions Core ML² ne comporte pas de crochets.

La réduction $(\lambda x.\widehat{0})\,\langle\widehat{n}_1 \mid \widehat{n}_2\rangle \to \widehat{0}$ que nous venons d'étudier est très simple. Cependant, les réductions de Core ML² peuvent, en général, être plus complexes : plusieurs des règles de réduction que je vais introduire doivent faire remonter les crochets quand ils bloquent la réduction (j'appellerai ces règles des règles « *lift* »). Par exemple, puisque l'application $\langle\lambda x.x \mid \lambda x.\widehat{0}\rangle\,\widehat{1}$ n'est pas un β-redex, elle ne peut être réduite par (β). C'est pourquoi j'introduirai une nouvelle règle, (lift-β), pour la réduire en $\langle(\lambda x.x)\,\widehat{1} \mid (\lambda x.\widehat{0})\,\widehat{1}\rangle$. Il faut remarquer que cette étape de réduction n'affecte aucune des projections de l'expression en question. Ainsi, elle n'a pas de contenu calculatoire : le déplacement des crochets reflète seulement un flot d'information. Enfin, chaque membre de la nouvelle expression est maintenant un β-redex qui peut être réduit : nous obtenons ainsi $\langle(\lambda x.x)\,\widehat{1} \mid (\lambda x.\widehat{0})\,\widehat{1}\rangle \to^* \langle\widehat{1} \mid \widehat{0}\rangle$.

5.2.2 États mémoire et configurations

De même que la syntaxe des valeurs, résultats et expressions, je dois étendre celle des blocs mémoire, des états mémoire et des configurations pour donner une sémantique opérationnelle à Core ML². En effet, il faut garder trace du partage non seulement entre les expressions mais aussi entre les états mémoire de deux configurations. Cependant, deux états mémoire peuvent allouer des adresses mémoire distinctes ; c'est la raison pour laquelle la syntaxe des *blocs mémoire* doit être étendue comme suit :

$$w ::= \ldots \mid \langle v \mid \mathsf{null}\rangle \mid \langle \mathsf{null} \mid v\rangle \qquad \text{(bloc mémoire)}$$

Les blocs de la forme $\langle v \mid \mathsf{null}\rangle$ ou $\langle \mathsf{null} \mid v\rangle$ (où la valeur v doit être close) permettent de représenter les situations où une adresse mémoire m est allouée dans seulement un des deux états mémoire Core ML représentés par un état mémoire Core ML², qui est naturellement une fonction totale

Réductions de base

$$
\begin{array}{rcll}
(\lambda x.e)\, v \,/_i\, \mu & \to & e[x \leftarrow v] \,/_i\, \mu & (\beta) \\
\mathsf{let}\, x = v \,\mathsf{in}\, e \,/_i\, \mu & \to & e[x \leftarrow v] \,/_i\, \mu & (\mathrm{let}) \\
f\, \vec{v} \,/_i\, \mu & \to & e \,/_i\, \mu \circledcirc_i \mu' & (\delta) \\
 & & \textit{si } f\, \vec{v} \,/\, \lfloor \mu \rfloor_i \multimap\!\!\!\!\!\rightarrow e + \mu' & \\
\mathbb{E}[a] \,/\, \mu & \to & \mathbb{E} \ll a \,/\, \mu & (\mathrm{pop})
\end{array}
$$

Règles « lift »

$$
\begin{array}{rcll}
\langle v_1 \mid v_2 \rangle\, v \,/\, \mu & \to & \langle v_1 \lfloor v \rfloor_1 \mid v_2 \lfloor v \rfloor_2 \rangle \,/\, \mu & (\text{lift-}\beta) \\
f\, \vec{v} \,/\, \mu & \to & \langle f \lfloor \vec{v} \rfloor_1 \mid f \lfloor \vec{v} \rfloor_2 \rangle \,/\, \mu & (\text{lift-}\delta) \\
 & & \textit{si } \vec{v} \,/\, \mu \not\downarrow_f,\ \lfloor \vec{v} \rfloor_1 \,/\, \lfloor \mu \rfloor_1 \downarrow_f \textit{ et } \lfloor \vec{v} \rfloor_2 \,/\, \lfloor \mu \rfloor_2 \downarrow_f & \\
\mathbb{E}[\langle a_1 \mid a_2 \rangle] \,/\, \mu & \to & \langle \lfloor \mathbb{E} \rfloor_1 \ll a_1 \mid \lfloor \mathbb{E} \rfloor_2 \ll a_2 \rangle \,/\, \mu & (\text{lift-pop}) \\
 & & \textit{si } \mathbb{E} \textit{ n'accepte pas } \langle a_1 \mid a_2 \rangle &
\end{array}
$$

Réduction sous un contexte

$$
\begin{array}{rcll}
\mathbb{E}[e] \,/_i\, \mu & \to & \mathbb{E}[e'] \,/_i\, \mu' & (\mathrm{context}) \\
 & & \textit{si } e \,/_i\, \mu \to e' \,/_i\, \mu' & \\
\langle e_1 \mid e_2 \rangle \,/\, \mu & \to & \langle e'_1 \mid e'_2 \rangle \,/\, \mu' & (\mathrm{bracket}) \\
 & & \textit{si } e_{i_1} \,/_{i_1}\, \mu \to e'_{i_1} \,/_{i_1}\, \mu' \textit{ et } e_{i_2} = e'_{i_2} & \\
 & & \textit{avec } \{i_1, i_2\} = \{1, 2\} &
\end{array}
$$

Figure 5.3 – Sémantique opérationelle de Core ML^2

des adresses mémoire vers ces blocs mémoire. Les fonctions de projection sont étendues au blocs mémoire par les égalités

$$
\begin{array}{lll}
\lfloor \mathsf{null} \rfloor_1 = \mathsf{null} & \lfloor \langle \mathsf{null} \mid v \rangle \rfloor_1 = \mathsf{null} & \lfloor \langle v \mid \mathsf{null} \rangle \rfloor_1 = v \\
\lfloor \mathsf{null} \rfloor_2 = \mathsf{null} & \lfloor \langle \mathsf{null} \mid v \rangle \rfloor_2 = v & \lfloor \langle v \mid \mathsf{null} \rangle \rfloor_2 = \mathsf{null}
\end{array}
$$

puis point à point aux états mémoire : pour tous i et m, $\lfloor \mu \rfloor_i(m) = \lfloor \mu(m) \rfloor_i$. Un état mémoire μ de Core ML^2 est bien formé si et seulement si seul un nombre fini d'adresses mémoire est alloué dans μ, et si, pour tous $i \in \{1, 2\}$, $\mu \in \mathrm{fml}(\lfloor \mu \rfloor_i)$ implique $\lfloor \mu(m) \rfloor_i \neq \mathsf{null}$. Ce critère est équivalent à la bonne formation, au sens de Core ML, des deux états mémoire $\lfloor \mu \rfloor_1$ et $\lfloor \mu \rfloor_2$.

Une *configuration* Core ML^2, notée $e \,/_i\, \mu$, est un triplet formé d'une expression e, d'un état mémoire μ ainsi que d'un index $i \in \{\bullet, 1, 2\}$ dont j'expliquerai le rôle section 5.2.3. Notons toutefois que, dans le cas où i est 1 ou 2, e doit être une expression Core ML (c'est-à-dire une expression sans crochets). J'écris $e \,/\, \mu$ pour $e \,/_\bullet\, \mu$ et définis les projections d'une telle configuration par $\lfloor e \,/\, \mu \rfloor_i = \lfloor e \rfloor_i \,/\, \lfloor \mu \rfloor_i$.

Les règles de bonne formation des configurations données pour Core ML doivent être étendues à Core ML^2 comme suit :

- La configuration e/μ est bien formée si et seulement si μ est bien formé et pour tous $i \in \{1, 2\}$ et $m \in \mathrm{fml}(\lfloor e \rfloor_i)$, $\lfloor \mu(m) \rfloor_i \neq \mathsf{null}$. Ce critère est équivalent à la bonne formation, au sens de Core ML, des configurations $\lfloor e \,/\, \mu \rfloor_1$ et $\lfloor e \,/\, \mu \rfloor_2$.
- Pour pour $i \in \{1, 2\}$, la configuration $e \,/_i\, \mu$, est bien formée si et seulement si μ est bien formé et, pour tout $m \in \mathrm{fml}(e)$, $\lfloor \mu(m) \rfloor_i \neq \mathsf{null}$.

5.2.3 Sémantique opérationnelle

La sémantique opérationnelle à petits pas de Core ML^2 est donnée figure 5.3. Cette sémantique étend celle donnée pour Core ML (section 5.1.2, page 99). Les règles sont conçues de telle sorte que l'image d'une réduction par une projection soit également une réduction valide. L'évaluation

peut avoir lieu en dehors des crochets — les deux projections effectuant alors la même étape de réduction — ou bien avoir lieu à l'intérieur des crochets — l'une des projections progressant indépendamment, l'autre restant inchangée — ou bien faire remonter des crochets — perdant alors une partie du partage.

Les règles du premier groupe sont syntaxiquement identiques à celles de Core ML — à l'exception de quelques détails techniques expliqués dans le prochain paragraphe. Les meta-variable v, a, e et μ représentent maintenant des valeurs, résultats, expressions et états mémoire de Core ML^2 qui peuvent comporter des crochets $\langle \cdot \mid \cdot \rangle$.

La substitution de la variable x par la valeur v dans e, utilisée dans les règles (β) et (let), est notée $e[x \leftarrow v]$. Elle est définie de la manière habituelle, exceptée aux nœuds $\langle \cdot \mid \cdot \rangle$, où la projection appropriée de v doit être prise pour chaque branche : $\langle e_1 \mid e_2 \rangle[x \leftarrow v] = \langle e[x \leftarrow \lfloor v \rfloor_1] \mid e[x \leftarrow \lfloor v \rfloor_2] \rangle$. La fonction $\ll$ utilisée dans la règle (pop) peut toujours être définie par les égalités de la figure 5.2 (page 100). Cependant, celles-ci doivent désormais être lues en considérant que les meta-variable e, v, a et h dénotent des expressions, valeurs, résultats et clauses du langage Core ML^2. Il faut noter que $\ll$ n'est plus une fonction totale : $(\mathsf{bind}\, x = [\,]\, \mathsf{in}\, e) \ll \langle a_1 \mid a_2 \rangle$ et $([\,]\, \mathsf{handle}\, \vec{h}) \ll \langle a_1 \mid a_2 \rangle$ ne sont pas définis si a_1 et a_2 ne sont pas deux valeurs. La règle (pop) n'est naturellement applicable que si $\mathbb{E} \ll a$ est défini, on dit dans ce cas que $\mathbb{E}$ *accepte* a. De même, la sémantique des primitives doit être étendue : elle est donnée par des relations $-_f\twoheadrightarrow$ (dont les jugements sont de la forme $\vec{v} \,/\, \mu \; -_f\twoheadrightarrow e + \dot{\mu}$ où les valeurs $\vec{v}$, l'état mémoire μ et le fragment $\dot{\mu}$ ainsi que l'expression e peuvent comporter des crochets), et qui étendent les relations $-_f\rightarrow$. Je donnerai des exemples section 5.2.4 (page 108).

Les règles des deux premiers groupes sont applicables sous tout contexte. Cependant, (δ) a besoin d'un peu d'information contextuelle pour accéder à l'état mémoire : les réductions qui ont lieu sous une construction $\langle \cdot \mid \cdot \rangle$ ne doivent lire ou écrire qu'une de ses deux projections. C'est précisément le rôle de l'index i porté par les configurations : sa valeur est $\bullet$ pour les étapes de réduction en dehors des crochets, la règle (bracket) rend sa valeur égale à 1 (resp. 2) pour les réductions qui ont lieu à l'intérieur d'une branche gauche (resp. droite) d'une construction $\langle \cdot \mid \cdot \rangle$. Cet indice est ensuite utilisé dans la règle (δ) pour manipuler l'état mémoire d'une manière convenable. Tout d'abord, la primitive ne doit accéder qu'à la partie de l'état mémoire qui correspond au contexte : ainsi, le membre gauche de la prémisse de (δ) mentionne la projection $\lfloor \mu \rfloor_i$ (par convention, on pose $\lfloor \mu \rfloor_\bullet = \mu$). D'autre part, la réduction ne peut affecter (en allouant ou en modifiant des blocs) que cette même projection : pour $i \in \{\bullet, 1, 2\}$, l'opérateur $\mu \Subset_i \mu'$ est défini par les équations

$$
\begin{array}{lcll}
(\mu \Subset_i \mu')(m) & = & \mu(m) & \text{si } m \notin \mathrm{dom}(\mu') \\
(\mu \Subset_\bullet \mu')(m) & = & \mu'(m) & \left.\vphantom{\begin{array}{l}a\\a\\a\end{array}}\right\} \text{si } m \in \mathrm{dom}(\mu') \\
(\mu \Subset_1 \mu')(m) & = & \langle \mu'(m) \mid \lfloor \mu(m) \rfloor_2 \rangle & \\
(\mu \Subset_2 \mu')(m) & = & \langle \lfloor \mu(m) \rfloor_1 \mid \mu'(m) \rangle &
\end{array}
$$

Notons que l'indice i n'est pas transmis aux relations $-_f\twoheadrightarrow$. En utilisant le fait qu'elles retournent un fragment $\dot{\mu}$ précisant les modifications apportées à l'état mémoire initial (au lieu du nouvel état mémoire lui-même), il est en effet possible de déduire la sémantique des primitives dans les contextes $i = 1$ et $i = 2$ à partir de celle donnée pour le cas $i = \bullet$: c'est précisément le rôle de la règle (δ) et de l'opérateur $\Subset_i$.

Il me faut enfin énoncer quelques hypothèses sur chaque relation $-_f\twoheadrightarrow$. Les deux premières hypothèses sont techniques, elles visent à préserver la bonne formation des configurations : l'une est la contrepartie l'hypothèse 5.1 (page 100) donnée pour Core ML, l'autre garantit que si l'argument d'une primitive ne comporte pas de crochets alors il en est de même de son résultat (ce qui assure que l'on ne construit pas d'imbrications de telles constructions).

Hypothèse 5.5 *Pour tout f, **(i)** la relation $-_f\twoheadrightarrow$ est stable par renommage des adresses mémoires et **(ii)** si $f\, \vec{v} \,/\, \mu \; -_f\rightarrow e + \dot{\mu}$ alors, pour tout $i \in \{1, 2\}$ et $m \in \mathrm{fml}(\lfloor e \rfloor_i, \lfloor \dot{\mu} \rfloor_i)$, $m \in \mathrm{dom}(\dot{\mu})$ ou $\lfloor \mu(m) \rfloor_i \neq \mathsf{null}$.* □

Hypothèse 5.6 *Si $f\,\vec{v}\,/\,\mu \mathrel{-f\twoheadrightarrow} e + \dot{\mu}$ et $\vec{v}$ et μ ne comportent pas de construction $\langle\cdot \mid \cdot\rangle$ alors il en est de même de e et $\dot{\mu}$.* □

La troisième hypothèse exprime, au niveau des primitives, la correspondance entre la sémantique de Core ML^2 et celle du sous-ensemble Core ML : l'image d'une réduction par une projection doit rester une réduction valide. Cette hypothèse est utilisée dans la preuve du lemme 5.10 (page 109).

Hypothèse 5.7 *Si $f\,\vec{v}\,/\,\mu \mathrel{-f\twoheadrightarrow} e + \dot{\mu}$ alors, pour tout $i \in \{1,2\}$, $f\,\lfloor\vec{v}\rfloor_i\,/\,\lfloor\mu\rfloor_i \mathrel{-f\rightarrow} \lfloor e\rfloor_i + \lfloor\dot{\mu}\rfloor_i$.* □

Les règles du deuxième groupe empêchent les constructions $\langle\cdot \mid \cdot\rangle$ de bloquer la réduction : pour cela, elles « remontent » les crochets, départageant des sous-expressions mais permettant à la réduction de se poursuivre indépendemment dans chaque branche. Par exemple, l'expression du membre gauche de (lift-β) n'est pas un β-redex et son application duplique la sous-expression v. Le calcul étiqueté de Abadi, Lampson et Lévy [ALL96] comporte une règle quelque peu analogue, permettant de réduire les applications dont le membre gauche est une valeur étiquetée. On peut interpréter les règles *lift* en voyant le contenu de chaque $\langle\cdot \mid \cdot\rangle$ comme « secret » : en rendant de nouvelles sous-expressions secrètes lors des réductions, ces règles donnent en fait une description dynamique des flots d'information.

La règle (lift-δ) permet de départager un appel à une primitive bloqué par des crochets. Le prédicat $\vec{v}\,/\,\mu\downarrow_f$ utilisé dans sa condition d'application signifie que, sous l'état mémoire, l'appel à la primitive f avec les arguments $\vec{v}$ n'est pas bloquant, *i.e.* il existe a et $\dot{\mu}$ tels que $f\,\vec{v}/\mu \mathrel{-f\twoheadrightarrow} a+\dot{\mu}$.

La règle (lift-pop) complète la règle (pop) en réduisant les expressions de la forme $\mathbb{E}[a]$ où $\mathbb{E}$ n'accepte pas a. Le traitement de ces formes d'expressions diffère quelque peu de celui suivi pour les paires de règles (β) et (lift-β) ou (δ) et (lift-δ) : en effet, en plus de départager le contexte $\mathbb{E}$, (lift-pop) effectue immédiatement une étape de réduction dans chaque projection. Il serait probablement plus naturel de considérer la règle suivante :

$$\mathbb{E}[\langle a_1 \mid a_2\rangle] \to \langle \lfloor\mathbb{E}\rfloor_1[a_1] \mid \lfloor\mathbb{E}\rfloor_2[a_2]\rangle \quad \textit{si } \mathbb{E} \textit{ n'accepte pas } \langle a_1 \mid a_2\rangle \qquad \text{(lift-pop')}$$

Supposons un instant que règle (lift-pop) est remplacée par (lift-pop'). La réduction $\mathbb{E}[\langle a_1 \mid a_2\rangle]\,/\,\mu \to^* \langle \lfloor\mathbb{E}\rfloor_1 \ll a_1 \mid \lfloor\mathbb{E}\rfloor_2 \ll a_2\rangle\,/\,\mu$ s'effectue alors en trois étapes de réduction :

$$\begin{aligned}
\mathbb{E}[\langle a_1 \mid a_2\rangle]\,/\,\mu &\to \langle \lfloor\mathbb{E}\rfloor_1[a_1] \mid \lfloor\mathbb{E}\rfloor_2[a_2]\rangle\,/\,\mu && \text{(lift-pop')}\\
&\to \langle \lfloor\mathbb{E}\rfloor_1 \ll a_1 \mid \lfloor\mathbb{E}\rfloor_2[a_2]\rangle\,/\,\mu && \text{(pop)}\\
&\to \langle \lfloor\mathbb{E}\rfloor_1 \ll a_1 \mid \lfloor\mathbb{E}\rfloor_2 \ll a_2\rangle\,/\,\mu && \text{(pop)}
\end{aligned}$$

Cependant, cette définition de la sémantique de Core ML^2 nécessiterait une formulation plus complexe du système de type pour Core ML^2 au chapitre 6 (page 113), de manière à garantir la préservation du typage par *chacune* des étapes intermédiaires. Le langage Core ML^2 n'étant qu'un outil utilisé pour prouver le théorème de non-interférence, je préfère compliquer légèrement la définition de sa sémantique et conserver une formulation simple du système de type.

Bien entendu, la sémantique de Core ML^2 est donnée de manière à limiter autant que possible la perte de partage ; on pourrait en effet remplacer toutes les règles *lift* par la seule règle

$$e\,/\,\mu \to \langle \lfloor e\rfloor_1 \mid \lfloor e\rfloor_2\rangle\,/\,\mu$$

qui est sémantiquement correcte, mais obligerait ensuite le système de type à considérer toute expression comme « secrète ». Par ailleurs, la construction $\langle\cdot \mid \cdot\rangle$ rappelle les nœuds *fork* introduits par Field et Teitelbaum pour effectuer une réduction incrémentale des λ-termes [FT90] : (lift-β) est en fait l'une de leurs règles de réduction. Cependant, mon approche diffère sur certains points. En particulier, je manipule des expressions alors que Field et Teitelbaum considèrent des graphes, permettant à des sous-expressions d'être partagées par les deux projections.

$$\widehat{n_1} \mathbin{\widehat{+}} \widehat{n_2} \,/\, \mu \;\; -\mathbin{\widehat{+}}\twoheadrightarrow \;\; \widehat{n_1 + n_2} + \varnothing \qquad \text{(add)}$$

$$\begin{aligned}
\mathsf{proj}_j\,(v_1, v_2) \,/\, \mu \;\; &-\mathsf{proj}_j\twoheadrightarrow \;\; v_j + \varnothing && \text{(proj)}\\
(\mathsf{inj}_j\, v)\ \mathsf{case}\ v_1\ v_2 \,/\, \mu \;\; &-\mathsf{case}\twoheadrightarrow \;\; v_j\, v + \varnothing && \text{(case)}\\
\mathsf{ref}\, v \,/\, \mu \;\; &-\mathsf{ref}\twoheadrightarrow \;\; m + (m \mapsto v) \quad si\ \mu(m) = \mathsf{null} && \text{(ref)}\\
m := v' \,/\, \mu[m \mapsto v] \;\; &-:=\twoheadrightarrow \;\; () + (m \mapsto v') && \text{(assign)}\\
!\, m \,/\, \mu[m \mapsto v] \;\; &-!\twoheadrightarrow \;\; v + \varnothing && \text{(deref)}\\
\mathsf{fix}\, v_1\, v_2 \,/\, \mu \;\; &-\mathsf{fix}\twoheadrightarrow \;\; \mathsf{bind}\, x_1 = v_1\, (\lambda x_2.\mathsf{fix}\, v_1\, x_2)\ \mathsf{in}\ x_1\, v_2 + \varnothing && \text{(fix)}
\end{aligned}$$

Figure 5.4 – Règles de réduction des constantes

$$\begin{aligned}
\langle v_1 \mid v_2\rangle \mathbin{\widehat{+}} v \,/\, \mu \;&\to\; \langle v_1 \mathbin{\widehat{+}} \lfloor v\rfloor_1 \mid v_2 \mathbin{\widehat{+}} \lfloor v\rfloor_2\rangle \,/\, \mu && \text{(lift-add-1)}\\
v \mathbin{\widehat{+}} \langle v_1 \mid v_2\rangle \,/\, \mu \;&\to\; \langle \lfloor v\rfloor_1 \mathbin{\widehat{+}} v_1 \mid \lfloor v\rfloor_2 \mathbin{\widehat{+}} v_2\rangle \,/\, \mu && \text{(lift-add-2)}\\
\mathsf{proj}_j\, \langle v_1 \mid v_2\rangle \,/\, \mu \;&\to\; \langle \mathsf{proj}_j\, v_1 \mid \mathsf{proj}_j\, v_2\rangle \,/\, \mu && \text{(lift-proj)}\\
\langle v_1 \mid v_2\rangle\ \mathsf{case}\ v_1'\ v_2' \,/\, \mu \;&\to\; \langle v_1\ \mathsf{case}\ \lfloor v_1'\rfloor_1\ \lfloor v_2'\rfloor_1 \mid v_2\ \mathsf{case}\ \lfloor v_1'\rfloor_2\ \lfloor v_2'\rfloor_2\rangle \,/\, \mu && \text{(lift-case)}\\
\langle v_1 \mid v_2\rangle := v \,/\, \mu \;&\to\; \langle v_1 := \lfloor v\rfloor_1 \mid v_2 := \lfloor v\rfloor_2\rangle \,/\, \mu && \text{(lift-assign)}\\
!\langle v_1 \mid v_2\rangle \,/\, \mu \;&\to\; \langle !\, v_1 \mid !\, v_2\rangle \,/\, \mu && \text{(lift-deref)}
\end{aligned}$$

Figure 5.5 – Règles « lift »

5.2.4 Constantes

Je termine cette section en poursuivant le traitement des exemples de constructeurs et destructeurs introduits à la section 5.1.3 (page 102). Les règles de réduction de ces constantes dans Core ML² sont données figure 5.4. Une fois encore, elles sont *syntaxiquement* identiques à celles des règles de Core ML, cependant les meta-variable v et μ désignent ici des valeurs et états mémoire Core ML². Il est immédiat de vérifier la correction de ces règles exprimées par l'énoncé suivant.

Propriété 5.8 *Les relations* $-\mathbin{\widehat{+}}\twoheadrightarrow$, $-\mathsf{proj}_j\twoheadrightarrow$, $-\mathsf{case}\twoheadrightarrow$, $-\mathsf{ref}\twoheadrightarrow$, $-:=\twoheadrightarrow$, $-!\twoheadrightarrow$ *et* $-\mathsf{fix}\twoheadrightarrow$ *vérifient les hypothèses 5.5, 5.6 et 5.7.* □

Dans la présentation originale de Core ML² [PS02, PS03], chaque primitive est munie d'une règle *lift* spécifique applicable lorsque un argument de la forme $\langle v_1 \mid v_2\rangle$ bloque la réduction. Ces règles sont rappelées par la figure 5.5. Elles sont intéressantes pour elles-mêmes, car elles décrivent les flots d'information engendrés par les primitives. Dans cette thèse, ces règles sont remplacées par (lift-δ), qui départage une application de primitive quand son évaluation est bloquée par les crochets. Pour ce qui concerne nos exemples, cette règle permet de réaliser exactement les réductions indiquées figure 5.5.

5.3 Simulation

Je montre dans cette section que Core ML² est un outil approprié pour raisonner simultanément sur les exécutions de deux programmes Core ML. Cette correspondance est exprimée par le théorème 5.13 (page 110). Je l'obtiens en énonçant deux résultats intermédiaires intéressants par eux-mêmes : la *correction* (lemme 5.10), qui établit que l'image d'une réduction par une projection

est également une réduction valide, et la *complétude* (lemme 5.12, page 110) qui assure que si les deux projections d'une expression peuvent être réduites vers un résultat alors l'expression tout entière peut elle même être réduite vers un résultat.

Lemme 5.9 *Soit* $\{i_1, i_2\} = \{1, 2\}$. *Si* $e \,/_{i_1}\, \mu \to e' \,/_{i_1}\, \mu$ *alors* $e \,/\, \lfloor\mu\rfloor_{i_1} \to e' \,/\, \lfloor\mu'\rfloor_{i_1}$ *et* $\lfloor\mu\rfloor_{i_2} = \lfloor\mu'\rfloor_{i_2}$. □

⌜ *Preuve.* Par induction sur la dérivation de l'hypothèse. Tous les cas sont immédiats, sauf le suivant.

∘ *Cas* (δ). L'hypothèse est $f\,\vec{v} /_{i_1} \mu \to e' /_{i_1} \mu \oslash_{i_1} \dot{\mu}$ et la prémisse de (δ) est $f\,\vec{v} / \lfloor\mu\rfloor_{i_1} \multimap\!\!\!\!\!\rightarrow e' + \dot{\mu}$ Avec cette même prémisse, la règle (δ) permet également de dériver la réduction $f\,\vec{v} \,/\, \lfloor\mu\rfloor_{i_1} \to e' \,/\, \lfloor\mu\rfloor_{i_1} \oslash \dot{\mu}$. Il suffit donc de montrer les égalités $\lfloor\mu \oslash_{i_1} \dot{\mu}\rfloor_{i_1} = \lfloor\mu\rfloor_{i_1} \oslash \dot{\mu}$ et $\lfloor\mu \oslash_{i_1} \dot{\mu}\rfloor_{i_2} = \lfloor\mu\rfloor_{i_2}$ pour conclure. Soit m une adresse mémoire.

· *Si* $m \in \mathrm{dom}(\dot{\mu})$, alors $(\mu \oslash_{i_1} \dot{\mu})(m) = \langle \dot{\mu}(m) \mid \lfloor\mu(m)\rfloor_2 \rangle$ si $i_1 = 1$ ou bien $(\mu \oslash_{i_1} \dot{\mu})(m) = \langle \lfloor\mu(m)\rfloor_1 \mid \dot{\mu}(m) \rangle$ si $i_1 = 2$, et $(\lfloor\mu\rfloor_{i_1} \oslash \dot{\mu})(m) = \dot{\mu}(m)$. On en déduit que $\lfloor\mu \oslash_{i_1} \dot{\mu}\rfloor_{i_1}(m) = (\lfloor\mu\rfloor_{i_1} \oslash \dot{\mu})(m)$ et $\lfloor\mu \oslash_{i_1} \dot{\mu}\rfloor_{i_2}(m) = \lfloor\mu\rfloor_{i_2}(m)$.

· *Si* $m \notin \mathrm{dom}(\dot{\mu})$, alors $(\mu \oslash_{i_1} \dot{\mu})(m) = \mu(m)$ et $(\lfloor\mu\rfloor_{i_1} \oslash \dot{\mu})(m) = \lfloor\mu(m)\rfloor_{i_1}$. On en déduit que $\lfloor\mu \oslash_{i_1} \dot{\mu}\rfloor_{i_1}(m) = (\lfloor\mu\rfloor_{i_1} \oslash \dot{\mu})(m)$ et $\lfloor\mu \oslash_{i_1} \dot{\mu}\rfloor_{i_2}(m) = \lfloor\mu\rfloor_{i_2}(m)$. ⌟

Le lemme suivant exprime tout d'abord la correction de la sémantique de Core ML^2 : l'image d'une réduction par une projection est également une réduction valide. Cependant, les projections ne préservent pas le nombre d'étapes de réduction, chaque étape de réduction pouvant en effet se projeter en zéro ou une étape. C'est pourquoi je montre également que les règles de réduction de base ont effectivement un contenu calculatoire, *i.e.* correspondent à une étape de réduction pour au moins une des projections. Cette propriété sera utile pour la preuve du lemme 5.12.

Lemme 5.10 (Correction) *Si* $e/\mu \to e'/\mu'$ *alors, pour tout* $i \in \{1, 2\}$, $\lfloor e/\mu\rfloor_i \to^* \lfloor e'/\mu'\rfloor_i$. *De plus, si* $e \,/\, \mu \to e' \,/\, \mu'$ *est dérivée sans recours aux règles* lift *alors il existe* i *tel que* $\lfloor e \,/\, \mu\rfloor_i \to \lfloor e' \,/\, \mu'\rfloor_i$. □

⌜ *Preuve.* On procède par induction sur la dérivation de l'hypothèse. Tous les cas sont immédiats, exceptés les suivants.

∘ *Cas* (δ) L'hypothèse est $f\,\vec{v} \,/\, \mu \to e' \,/\, \mu \oslash \dot{\mu}$ et la prémisse de (δ) est $f\,\vec{v} \,/\, \mu \multimap\!\!\!\!\!\rightarrow e' + \dot{\mu}$ **(1)**. Par l'hypothèse 5.7 (page 107), (1) implique $f\,\lfloor\vec{v}\rfloor_i \,/\, \lfloor\mu\rfloor_i \multimap\!\!\!\!\!\rightarrow \lfloor e\rfloor_i + \lfloor\dot{\mu}\rfloor_i$. En appliquant ($\delta$), on obtient $f\,\lfloor\vec{v}\rfloor_i \,/\, \lfloor\mu\rfloor_i \to \lfloor e'\rfloor_i \,/\, \lfloor\mu\rfloor_i \oslash \lfloor\dot{\mu}\rfloor_i$ **(2)**. Puisque $\lfloor\mu \oslash \dot{\mu}\rfloor_i = \lfloor\mu\rfloor_i \oslash \lfloor\dot{\mu}\rfloor_i$, (2) est le but recherché.

∘ *Cas (bracket)* L'hypothèse est $\langle e_1 \mid e_2 \rangle \,/\, \mu \to \langle e_1' \mid e_2' \rangle \,/\, \mu'$ et les prémisses de (bracket) sont $e_{i_1} /_{i_1} \mu \to e'_{i_1} /_{i_1} \mu'$ **(1)**, $e'_{i_2} = e_{i_2}$ **(2)** et $\{i_1, i_2\} = \{1, 2\}$ **(3)**. En appliquant le lemme 5.9 à (3) et (1), on a $e_{i_1} \,/\, \lfloor\mu\rfloor_{i_1} \to e'_{i_1} \,/\, \lfloor\mu'\rfloor_{i_1}$ **(4)** et $\lfloor\mu\rfloor_{i_2} = \lfloor\mu'\rfloor_{i_2}$ **(5)**. On déduit $\lfloor e \,/\, \mu\rfloor_{i_1} \to \lfloor e' \,/\, \mu'\rfloor_{i_1}$ de (4) puis $\lfloor e \,/\, \mu\rfloor_{i_2} \to^* \lfloor e' \,/\, \mu'\rfloor_{i_2}$ de (2) et (5), ce qui permet de conclure. ⌟

Une configuration $e /_i \mu$ est *bloquée* si et seulement si c'est une forme normale pour $\to$ et e n'est pas un résultat. Le lemme suivant — qui est la propriété fondamentale utilisée pour la preuve du lemme de complétude — montre que j'ai introduit suffisamment de règles *lift* pour que les constructions $\langle \cdot \mid \cdot \rangle$ ne bloquent jamais la réduction.

Lemme 5.11 *Si* $e \,/\, \mu$ *est bloquée alors il existe* $i \in \{1, 2\}$ *tel que* $\lfloor e \,/\, \mu\rfloor_i$ *est bloquée.* □

⌜ *Preuve.* On procède par induction sur la structure de l'expression e.

∘ *Cas* e *est un résultat.* Ce cas ne peut intervenir, puisque si e est un résultat, par définition, la configuration $e \,/\, \mu$ n'est pas bloquée.

∘ *Cas* $e = v_1\, v_2$. Puisque (β) et (lift-β) ne sont pas applicables, v_1 n'est pas de la forme $\lambda x.e$ ou $\langle v_{11} \mid v_{12}\rangle$. Par conséquent, pour tout $i \in \{1,2\}$, $\lfloor v_1 \rfloor_i$ n'est pas une λ-abstraction. On en déduit que $\lfloor e \,/\, \mu \rfloor_1$ et $\lfloor e \,/\, \mu \rfloor_2$ sont bloquées.

∘ *Cas* $e = f\, \vec{v}$. Puisque (δ) n'est pas applicable, on a $\vec{v}/\mu \not\downarrow_f$. Puisque (lift-$\delta$) n'est pas applicable, on en déduit qu'il existe $i \in \{1,2\}$ tel que $\lfloor \vec{v} \rfloor_i \,/\, \lfloor \mu \rfloor_i \not\downarrow_f$. La configuration $\lfloor e \,/\, \mu \rfloor_i$ est donc bloquée.

∘ *Cas* $e = \mathsf{let}\, x = v\ \mathsf{in}\ e'$. La configuration $e \,/\, \mu$ peut être réduite par (let) et n'est donc pas bloquée.

∘ *Cas* $e = \mathbb{E}[e']$. La configuration $e' \,/\, \mu$ doit être irréductible, sinon, par (context), $\mathbb{E}[e'] \,/\, \mu$ serait réductible. Puisque $\mathbb{E}[e']$ n'est réductible ni par (pop) ni par (lift-pop), e' n'est pas un résultat, ce qui implique que $e' \,/\, \mu$ est bloquée. Par hypothèse d'induction, il existe $i \in \{1,2\}$ tel que $\lfloor e' \,/\, \mu \rfloor_i$ soit bloquée. Par inspection des règles de réduction, il en est de même de $\mathbb{E}'[\lfloor e' \rfloor_i] \,/\, \lfloor \mu \rfloor_i$ pour tout contexte d'évaluation Core ML $\mathbb{E}'$, en particulier $\lfloor \mathbb{E} \rfloor_i$. On conclut que $\lfloor e \,/\, \mu \rfloor_i$ est bloquée.

∘ *Cas* $e = \langle e_1 \mid e_2 \rangle$. Puisque (bracket) n'est pas applicable, les configurations $e_1 \,/_1\, \mu$ et $e_2 \,/_2\, \mu$ sont irréductibles. Puisque e n'est pas un résultat, il existe $i \in \{1,2\}$ tel que e_i n'est pas un résultat. Par le lemme 5.9, on en déduit que $e_i \,/\, \lfloor \mu \rfloor_i$ est bloquée. ⌟

Lemme 5.12 (Complétude) *Si, pour tout* $i \in \{1,2\}$, $\lfloor e \,/\, \mu \rfloor_i \to^* a_i \,/\, \mu_i'$ *alors il existe une configuration* $a \,/\, \mu'$ *telle que* $e \,/\, \mu \to^* a \,/\, \mu'$. □

⌜ *Preuve.* Montrons tout d'abord que la configuration $e \,/\, \mu$ n'admet pas de réduction infinie. Une séquence infinie de réductions ne peut consister exclusivement d'instances des règles *lift* : en effet, chacune de ces règles rapproche strictement une construction $\langle \cdot \mid \cdot \rangle$ de la racine de l'expression réduite. On en déduit, par le lemme 5.10, que si $e \,/\, \mu$ admet une réduction infinie alors il en est de même d'au moins une de ses projections. Or, par le lemme 5.3 (page 101), les deux projections de $e \,/\, \mu$ se réduisent en une forme normale. La sémantique du fragment Core ML étant déterministe (lemme 5.4, page 101), $\lfloor e \,/\, \mu \rfloor_1$ et $\lfloor e \,/\, \mu \rfloor_2$ n'admettent pas de dérivation infinie.

La configuration $e \,/\, \mu$ se réduit donc en une forme normale. Supposons que cette dernière soit bloquée. Alors, par le lemme 5.11 au moins une de ses projections est bloquée, ce qui implique, par le lemme 5.10 que $\lfloor e \,/\, \mu \rfloor_i$ se réduit en une configuration bloquée pour un certain $i \in \{1,2\}$. La sémantique du fragment Core ML étant déterministe (lemme 5.4, page 101), cela contredit l'hypothèse $\lfloor e \,/\, \mu \rfloor_i \to^* a_i \,/\, \mu_i'$. On en conclut que $e \,/\, \mu$ se réduit en une configuration de la forme $a \,/\, \mu'$. ⌟

Je peux maintenant énoncer le théorème de *simulation* qui est une combinaison légèrement simplifiée des lemmes de correction et de complétude. Je note $e \to^* a$ si et seulement si il existe un état mémoire μ tel que $e \,/\, \varnothing \to a \,/\, \mu$, où $\varnothing$ représente l'état mémoire vide (*i.e.*, pour toute adresse mémoire m, $\varnothing(m) = \mathsf{null}$).

Théorème 5.13 (Simulation) *Soit* e *une expression Core ML². On a* $\lfloor e \rfloor_1 \to^* a_1$ *et* $\lfloor e \rfloor_2 \to^* a_2$ *si et seulement si il existe un résultat* a *tel que* $\lfloor a \rfloor_1 = a_1$, $\lfloor a \rfloor_2 = a_2$ *et* $e \to^* a$. □

⌜ *Preuve.* Supposons tout d'abord que, pour tout $i \in \{1,2\}$, $\lfloor e \rfloor_i \to^* a_i$. Par le lemme 5.12, il existe un résultat a tel que $e \to^* a$. En utilisant les lemmes 5.10 et 5.4, on vérifie que, pour tout $i \in \{1,2\}$, $\lfloor a \rfloor_i = a_i$. Inversement, supposons que $e \to^* a$ avec $\lfloor a \rfloor_i = a_i$ pour un certain $i \in \{1,2\}$. Par le lemme 5.10, la première hypothèse donne $\lfloor e \rfloor_i \to^* \lfloor a \rfloor_i$, qui devient, grâce à la deuxième hypothèse, $\lfloor e \rfloor_i \to^* a_i$. ⌟

Le résultat de complétude nécessite que les deux projections convergent ; il n'est pas applicable dans le cas où l'une d'entre elles diverge. En effet, posons $e = \mathsf{bind}\, x = \langle \Omega \mid \widehat{0} \rangle\ \mathsf{in}\ \widehat{0}$, où Ω est une expression qui ne termine pas. La projection droite de e est $\mathsf{bind}\, x = \widehat{0}\ \mathsf{in}\ \widehat{0}$ et se réduit vers $\widehat{0}$. Cependant e ne peut être réduit vers aucun terme dont la projection droite est $\widehat{0}$. Une telle formulation de la complétude va naturellement m'amener à établir un résultat de non-interférence

faible dans lequel deux programmes peuvent être assurés de produire le même résultat seulement si ils terminent tous les deux. Je ne cherche de toutes façons pas à obtenir une propriété de non-interférence *forte* : cela aurait peu de sens de s'intéresser au flots d'information liés à la terminaison sans prendre en compte le problème plus général des *timing leaks*. De plus, un tel résultat devrait nécessiter un système de type beaucoup plus restrictif.

Notons également que le résultat de simulation nécessite que le nombre d'adresses mémoire disponible soit infini : cette propriété est nécessaire pour la validité de la propriété 5.8 (page 108) pour ref. Cependant, les implémentations de ML sont naturellement limitées par la mémoire disponible sur la machine d'exécution. Elles déclenchent généralement une erreur si une nouvelle allocation est demandée alors que la mémoire est saturée. Cela signifie donc que le résultat de non-interférence que je vais établir ne considère pas cet autre canal de communication binaire, comme la terminaison.

En résumé, le lemme de complétude montre que j'ai donné assez de règles *lift* pour réduire toutes les expressions Core ML^2. Dans le prochain chapitre, à chacune de ces règles va correspondre un cas de la preuve de préservation du typage, obligeant ainsi à vérifier que le système de type prend en compte toutes les formes possibles de flots d'information.

CHAPITRE SIX

Typage et non-interférence

Je décris maintenant un système de type pour Core ML, dénommé MLIF($\mathcal{T}$). Ce système permet d'établir des propriétés de sécurité ou d'intégrité pour les programmes Core ML, qui sont des instances du théorème de non-interférence établi au terme de ce chapitre. Ce résultat, qui exprime la correction du système, est obtenu en le ramenant à la préservation du typage par réduction pour une extension du système de type au langage Core ML^2.

Le système MLIF($\mathcal{T}$) est basé sur les types *bruts* (section 1.2, page 21) : il n'a pas de notion de variables de type et traite le polymorphisme d'une manière extensionelle. Cette approche *semi-syntaxique* permet une preuve simple et abstraite de la correction. Elle a été introduite avec le système B($\mathcal{T}$), utilisé par Pottier [Pot01a] comme intermédiaire pour démontrer la sûreté de HM($\mathcal{X}$) [OSW99]. Cependant, cette formulation élémentaire ne donne pas d'algorithme de synthèse ni même de vérification de type : j'étudierai cette question au chapitre 7 (page 135).

Dans ce chapitre, chaque occurrence du symbole $\star$ doit être lue comme un type brut quelconque de sorte appropriée pour le contexte.

6.1 Types

Les types que je considère dans ce chapitre sont les *types bruts* définis section 1.2 (page 21). Dans cette section liminaire, j'avais introduit, de manière abstraite, trois ensembles d'atomes, de constructeurs de type et d'étiquettes. Je précise maintenant leur définition ainsi que leur rôle, pour le système MLIF($\mathcal{T}$) et cette partie de la thèse.

La définition de l'ensemble des atomes $(\mathcal{L}, \leq_{\mathcal{L}})$ est encore laissée abstraite : il peut en effet toujours s'agir d'un treillis arbitraire. Cependant, les atomes ont désormais un rôle bien précis : je les vois comme des *niveaux d'information* permettant de décrire la quotité d'information attachée à chaque expression. Ils peuvent représenter des concepts divers utilisés pour spécifier des notions de sécurité ou d'intégrité relatives aux données et aux programmes, comme par exemple des droits d'accès ou des hiérarchies de principaux. Dans le cas le plus simple, le treillis est réduit à deux éléments : $\bot$, associé aux données « publiques », et $\top$, pour les données « secrètes ». La relation d'ordre $\leq_{\mathcal{L}}$ décrit quant à elle l'ensemble des flots d'information autorisés : un transfert d'information est possible entre une source de niveau ℓ_1 et un receveur de niveau ℓ_2 si et seulement si $\ell_1 \leq_{\mathcal{L}} \ell_2$. Je note dans ce chapitre les niveaux d'information ℓ ou pc : suivant l'usage introduit par

$\to$	$:: \mathsf{Type}^- \cdot \mathsf{Atom}^- \cdot \mathsf{Row}_{\mathcal{E}}\,\mathsf{Atom}^+ \cdot \mathsf{Atom}^+ \cdot \mathsf{Type}^+$
ref	$:: \mathsf{Type}^{\pm} \cdot \mathsf{Atom}^+$
unit	$:: \varnothing$
int	$:: \mathsf{Atom}^+$
$\times$	$:: \mathsf{Type}^+ \cdot \mathsf{Type}^+$
$+$	$:: \mathsf{Type}^+ \cdot \mathsf{Type}^+ \cdot \mathsf{Atom}^+$

Figure 6.1 – Signatures et variances des constructeurs de type

Denning [DD77, Den82], cette seconde meta-variable est généralement employée pour les niveaux décrivant l'information associée au « compteur de programme » (*program counter* en anglais), c'est-à-dire au contexte d'évaluation.

Les étiquettes sont identifiées avec les noms d'exceptions, ce qui justifie l'identification des notations introduites pour ces deux notions aux chapitres 1 et 5. Les rangées brutes sont ainsi des fonctions associant à chaque nom d'exception un niveau d'information. Elle permettent de décrire l'information transmise par une expression *via* les exceptions qu'elle est susceptible de lever.

Je suppose enfin l'existence de deux constructeurs de type particuliers — $\to$ et ref — qui sont nécessaires à la formulation des règles du système MLIF($\mathcal{T}$). Leurs signatures et variances sont données dans la première moitié de la figure 6.1. L'application du constructeur $\to$ est notée de manière infixe avec les trois annotations au dessus la flèche : on écrit $t' \xrightarrow{pc\,[r]\,\ell} t$ pour $\to t'\,pc\,r\,\ell\,t$. Pour le typage des constantes introduites à la section 5.1.3 (page 102), je supposerai l'existence de quatre constructeurs de type supplémentaires — unit, int, $\times$ et $+$ — dont les signatures sont également données figure 6.1. Les applications des constructeurs $\times$ et $+$ sont notées de manière infixe : on écrit $t_1 \times t_2$ et $(t_1 + t_2)^\ell$ pour $\times\, t_1\, t_2$ et $+\, t_1\, t_2\, \ell$, respectivement. Ainsi, l'ensemble des types bruts considérés dans cette partie inclut le langage suivant :

$$t ::= t \xrightarrow{pc\,[r]\,\ell} t \mid \mathsf{ref}\,t\,\ell \mid \mathsf{unit} \mid \mathsf{int}\,\ell \mid t_1 \times t_2 \mid (t_1 + t_2)^\ell \mid \ldots \qquad \text{(type brut)}$$

Il s'agit des types habituels de ML, étendus avec des annotations de sécurité.

Les types de fonctions portent plusieurs annotations de sécurité. Tout d'abord, le niveau ℓ décrit l'information attachée à l'*identité* de la fonction. Quand la fonction est appliquée, une partie de cette information peut être reproduite dans son résultat ou dans ses effets : leurs niveaux de sécurité devront donc être supérieurs ou égaux à ℓ. L'annotation pc représente l'information portée par le contexte dans lequel la fonction est appelée. Pour ne pas laisser cette dernière s'échapper, la fonction ne pourra modifier des adresses mémoire, ou lever des exceptions, seulement à un niveau pc ou supérieur. En d'autres termes, le niveau pc représente une borne inférieure sur les effets de la fonction. Les annotations ℓ et pc sont standards et peuvent être trouvées sous différents noms dans la littérature, comme par exemple dans les travaux de Heintze et Riecke [HR98]. Une petite inadvertance de leur part est cependant corrigée en remarquant que l'annotation pc peut être rendue contravariante au lieu d'invariante.

En plus de ces deux niveaux de sécurité, les types de fonctions portent un effet $[r]$. Pour chaque nom d'exception ξ, le niveau $r(\xi)$ décrit la quotité d'information acquise en observant que la fonction a levé une exception de nom ξ. Suivant les travaux de Myers [Mye99a, Mye99b], j'associe un niveau de sécurité distinct pour chaque nom d'exception, de manière à obtenir une précision suffisante dans l'analyse de l'information qu'elles véhiculent. Les rangées brutes sont en fait très proches des ensembles de *path labels* de Myers ; je reviendrai sur ce point à la section 9.2 (page 160). Il faut noter que les rangées ne mentionnent pas le type des arguments des exceptions : en effet, comme en ML, le typage des exceptions est rendu monomorphe en supposant donnée une fonction

$\to$ :: $\mathsf{Type}^- \cdot \mathsf{Atom}^- \cdot \mathsf{Row}_{\mathcal{E}}\,\mathsf{Atom}^+ \cdot \mathsf{Atom}^{+\lhd} \cdot \mathsf{Type}^+$ ref :: $\mathsf{Type}^{\pm} \cdot \mathsf{Atom}^{+\lhd}$	$\ell_0 \lhd t' \xrightarrow{pc\,[r]\,\ell} t \;\Leftrightarrow\; \ell_0 \leq \ell$ $\ell_0 \lhd \mathsf{ref}\,t\,\ell \;\Leftrightarrow\; \ell_0 \leq \ell$
unit :: $\varnothing$ int :: $\mathsf{Atom}^{+\lhd}$ $\times$ :: $\mathsf{Type}^{+\lhd} \cdot \mathsf{Type}^{+\lhd}$ $+$ :: $\mathsf{Type}^{+} \cdot \mathsf{Type}^{+} \cdot \mathsf{Atom}^{+\lhd}$	$\ell_0 \lhd \mathsf{unit} \;\Leftrightarrow\; \text{vrai}$ $\ell_0 \lhd \mathsf{int}\,\ell \;\Leftrightarrow\; \ell_0 \leq \ell$ $\ell_0 \lhd t_1 \times t_2 \;\Leftrightarrow\; \ell_0 \lhd t_1 \text{ et } \ell_0 \lhd t_2$ $\ell_0 \lhd (t_1 + t_2)^\ell \;\Leftrightarrow\; \ell_0 \leq \ell$

Figure 6.2 – Paramètres gardés des constructeurs de type

type(·) des noms d'exceptions vers les types bruts. Ce choix permet de donner aux fonctions des types beaucoup plus concis.

Les types des adresses mémoire (ou références) portent une annotation, ℓ, qui représente la quotité d'information attachée à l'identité de la référence. Son contenu est lui-même décrit par le paramètre t.

Le type $\mathsf{int}\,\ell$ décrit les expressions entières dont la valeur peut refléter des informations de niveau ℓ. Puisqu'il n'y a qu'une seule valeur de type unit, (), la valeur d'une expression de type unit ne porte aucune information. Par conséquent, il serait superflu d'annoter le constructeur de type unit par un niveau de sécurité. Similairement, les types produits ne portent pas d'annotation de sécurité, puisque, en l'absence d'un opérateur d'égalité physique, comme == en Caml [Ler, LDG+b], toute l'information portée par une paire est en fait portée par ses composantes. Enfin, les types sommes portent une annotation de sécurité, ℓ, qui reflète la quotité d'information obtenue en déterminant si la valeur a été construite en utilisant l'injection gauche ou droite.

6.2 Gardes

Je dois maintenant expliciter comment les niveaux portés par les types permettent de préciser, dans le système MLIF($\mathcal{T}$), la quotité d'information associée à chaque valeur ou expression Core ML, et ainsi de décrire les flots d'information.

Dans la plupart des systèmes de type pour l'analyse de flots d'information [HR98, PC00, ZM02], une solution très simple est adoptée : dans un type, chaque nœud porte *exactement* un niveau d'information, qui est covariant. Dans le cadre de cette thèse, ce choix revient à se restreindre à des constructeurs de type ayant une signature de la forme $[\vec{\kappa} \cdot \mathsf{Atom}]$: sous cette hypothèse, le type associé à chaque expression est nécessairement de la forme $c\,\vec{t}\,\ell$. Cette propriété syntaxique peut alors être utilisée dans la formulation des règles de typage pour refléter les flots d'information au niveau des types. En effet, si le résultat produit par une expression est susceptible de porter des informations d'un certain niveau ℓ', il suffit de s'assurer que le niveau ℓ porté par le type associé à cette dernière, $c\,\vec{t}\,\ell$, vérifie l'inégalité $\ell' \leq \ell$.

Dans cette thèse, j'adopte une solution plus générale, en ne faisant pas d'hypothèse *a priori* sur la forme des types. Cette solution permet une formulation plus abstraite des règles de typage du système, et offre également une plus grande souplesse. Elle autorise en particulier l'introduction de constructeurs de type ne portant *aucun* niveau de sécurité, comme unit et $\times$, ce qui permet d'obtenir des types plus concis dans certains cas. La signification des niveaux d'information portés par les types est donnée par un prédicat binaire $\lhd$ de signature $\mathsf{Type} \cdot \mathsf{Atom}$. En quelques mots, l'assertion $\ell \lhd t$ (lire : ℓ *garde* t) contraint le type t à avoir un niveau de sécurité supérieur ou égal à ℓ ; elle est d'une certaine manière similaire à la notion de *type t protégé par un niveau ℓ* d'Abadi, Banarjee, Heintze et Riecke [ABHR99]. Ce prédicat est défini par la donnée, pour chaque constructeur c de signature $[\kappa_1 \cdots \kappa_n]$, d'un sous-ensemble de $[1, n]$ noté guarded(c). Chaque élément j de cette partie

doit correspondre à une position covariante de sorte Type ou Atom du constructeur c (*i.e.* $c.j = +$ et $\kappa_j \in \{\mathsf{Type}, \mathsf{Atom}\}$). La relation $\lhd$ est alors définie par la règle suivante :

$$\frac{c :: \kappa_1 \cdots \kappa_n \qquad \forall j \in \text{guarded}(c) \begin{cases} \kappa_j = \mathsf{Atom} \Rightarrow \ell \leq t_j \\ \kappa_j = \mathsf{Type} \Rightarrow \ell \lhd t_j \end{cases}}{\ell \lhd c\, t_1 \cdots t_n}$$

La définition de l'ensemble guarded(c) pour les constructeurs introduits à la section précédente est donnée dans la colonne de gauche de la figure 6.2, où les positions gardées sont signalées par l'exposant $\lhd$ dans les signatures. La colonne de droite de cette même figure donne la traduction de cette spécification sur la définition de la relation $\lhd$, sous forme d'équivalences logiques.

La propriété suivante montre que la définition du prédicat $\lhd$ est compatible avec le sous-typage. Elle est assurée par le fait que chaque position gardée d'un constructeur de type est au covariante.

Propriété 6.1 *Soient ℓ et ℓ' deux niveaux, t et t' deux types de sorte* Type. *Si $\ell' \leq \ell$, $\ell \lhd t$ et $t \leq t'$ alors $\ell' \lhd t'$.* □

⌜ *Preuve.* On procède par induction sur la dérivation de $\ell \lhd t$. Soient ℓ et ℓ' deux niveaux, t et t' deux types de sorte Type tels que $\ell' \leq \ell$ **(1)**, $\ell \lhd t$ **(2)** et $t \leq t'$ **(3)**. Le type t étant de sorte Type, il existe c et $t_1, \cdots, t_n$ tels que $t = c\, t_1 \cdots t_n$ **(4)**. Par (3), il existe $t'_1, \ldots, t'_n$, tels que $t' = c\, t'_1 \cdots t'_n$ **(5)** et, pour tout $j \in [1, n]$, $t_j \leq^{c.j} t'_j$ **(6)**. Soit $j \in$ guarded(c) **(7)**. On a $c.j = +$ donc, par (6), $t_j \leq t'_j$ **(8)**. On distingue deux cas :

· *Si $\kappa_j =$* Atom, alors, grâce à l'égalité (4), (2) et (7) impliquent $\ell \leq t_j$. Par (1) et (8), on en déduit $\ell' \leq t'_j$.

· *Si $\kappa_j =$* Type, alors, grâce à l'égalité (4), (2) et (7) impliquent $\ell \lhd t_j$ **(9)**. En appliquant l'hypothèse d'induction à (1), (9) et (8), on obtient $\ell' \lhd t'_j$.

On en conclut que $\ell' \lhd c\, t'_1 \cdots t'_n$, c'est-à-dire, grâce à l'égalité (5), $\ell' \lhd t'$. ⌟

6.3 Le système MLIF($\mathcal{T}$)

6.3.1 Jugements de typage

Un ***environnement mémoire*** M est une fonction partielle des adresses mémoire vers les types bruts de sorte Type.

La définition du système de type MLIF($\mathcal{T}$) fait intervenir cinq formes de jugements de typage ; chacune d'entre elles est destinée à une classe syntaxique particulière du langage : valeurs, expressions, clauses, états mémoire et configurations. L'ensemble des jugements valides est défini par les règles de dérivation des figures 6.3, 6.4 et 6.5, que je commenterai à la section 6.3.2. Je donne pour l'instant quelques explications sur les formes respectives des différents jugements.

Puisque les valeurs sont des formes normales du langage, leur évaluation ne produit pas d'effets de bord et ne lève pas d'exceptions. Ainsi, les jugements de la première sorte sont très simples :

$$X, M \vdash v : t$$

Cette forme de jugement est en fait identique à celle manipulée dans le système B($\mathcal{T}$) [Pot01a] (où elle est applicable à toutes les expressions et non aux seules valeurs) : t est le type (de sorte Type) obtenu pour la valeur v, X et M donnent les types des variables et adresses mémoire qui apparaissent dans v. Comme dans B($\mathcal{T}$), on étend cette forme de jugements aux schémas bruts en écrivant $X, M \vdash v : s$ si et seulement si $\forall t \in s \;\; X, M \vdash v : t$. Par ailleurs, puisque les expressions peuvent avoir des effets de bord et lever des expressions, la deuxième forme de jugements est plus élaborée :

$$pc, X, M \vdash e : t \;\; [r]$$

$$\text{V-VAR}\ \frac{t \in X(x)}{X, M \vdash x : t} \qquad \text{V-LOC}\ X, M \vdash m : M(m) \qquad \text{V-ABS}\ \frac{pc, X[x \mapsto t'], M \vdash e : t\ \ [r]}{X, M \vdash \lambda x.e : t' \xrightarrow{pc\,[r]\,\star} t}$$

$$\text{V-CONSTRUCTOR}\ \frac{\vdash k : t_1 \cdots t_n \to t \qquad \forall j\ \ X, M \vdash v_j : t_j}{X, M \vdash k\, v_1 \cdots v_n : t} \qquad \text{V-SUB}\ \frac{X, M \vdash v : t' \qquad t' \leq t}{X, M \vdash v : t}$$

Figure 6.3 – Règles de typage des valeurs

Le niveau pc indique la quotité d'information associée au contexte dans lequel l'expression e est placée, c'est-à-dire à la connaissance que l'expression e est évaluée. Ce niveau est une borne inférieure sur les effets de l'expression. Son introduction est standard, dans les analyses typées de flots d'information pour des langages munis d'effets de bord [VS97b, HR98]. La rangée r indique la quotité d'information obtenue en observant les exceptions qui s'échappent de e. Ces deux annotations sont naturellement étroitement reliées à celles qui décorent les types flèches : la correspondance est rendue explicite par les règles de typage des λ-abstractions (V-ABS) et des applications (E-APP).

Les jugements de typage portant sur les clauses sont utilisés comme prémisses de la règle E-HANDLE, laquelle permet de typer les contextes handle. La forme de ces jugements est comparable à celle utilisée pour les expressions :

$$pc, X, M \vdash h : r' \Rightarrow t\ \ [r]$$

Le type t est remplacé par une paire formée d'une rangée r' est d'un type t de sorte Type, notée $r' \Rightarrow t$. La rangée r' est celle produite par l'expression placée dans le trou du contexte handle auquel la clause h appartient : il s'agit donc des exceptions potentiellement rattrapables par h. Le type t et la rangée r décrivent quant à eux le résultat produit par la clause dans le cas où elle rattrape effectivement une exception.

Les deux dernières sortes de jugements employées permettent de raisonner à propos des états mémoire : $M \vdash \mu$ et des configurations : $X \vdash e \,/\, \mu : t\ \ [r]$. Elles sont analogues à celles utilisées dans B($\mathcal{T}$) [Pot01a].

6.3.2 Règles de typage

Je commente tout d'abord les règles de la figure 6.3 qui permettent de dériver des jugements portant sur des valeurs. Les règles V-VAR et V-LOC donnent des types aux variables et aux adresses mémoires à partir de l'environnement approprié. Remarquons que $X(x)$ est un schéma brut, dont V-VAR prend une instance arbitraire. Comme il est usuel dans les systèmes d'effets, V-ABS enregistre, dans les arguments du constructeur de type $\to$, les annotations relatives aux effets produits par le corps de la fonction. Chaque constructeur k est typé en supposant donné un ensemble d'axiomes de la forme $\vdash k : t_1 \cdots t_n \to t$ (où n est l'arité de k), qui sont utilisés comme prémisse de V-CONSTRUCTOR. La notation $t_1 \cdots t_n \to t$ peut être lue comme un type de fonction n-aire, qui n'est pas annoté puisque les constructeurs produisent des valeurs.

Les règles de la figure 6.4 produisent des jugements sur des expressions ou des clauses. La première d'entre elles, E-VALUE, permet de voir une valeur comme une expression et reflète le fait que les valeurs n'ont pas d'effets. La prémisse de E-RAISE vérifie que l'argument v de l'exception a un type approprié, déterminé à partir du nom d'exception ξ par la fonction type$(\cdot)$. Sa conclusion assure que l'effet associé à l'expression raise $\xi\, v$ est une rangée qui associe au moins le niveau pc au nom d'exception ξ. Cela garantit, en conjonction avec les règles E-BIND, E-HANDLE et E-FINALLY, que tout fragment de programme qui observe cette exception est typé à un niveau supérieur ou égal à pc.

Expressions

E-VALUE
$$\frac{X, M \vdash v : t}{\star, X, M \vdash v : t \ [\star]}$$

E-RAISE
$$\frac{X, M \vdash v : \mathrm{type}(\xi)}{pc, X, M \vdash \mathsf{raise}\,\xi\, v : \star \ [\xi : pc; \star]}$$

E-APP
$$\frac{X, M \vdash v_1 : t' \xrightarrow{pc \sqcup \ell\,[r]\,\ell} t \qquad X, M \vdash v_2 : t' \qquad \ell \lhd t}{pc, X, M \vdash v_1\, v_2 : t \ [r]}$$

E-DESTRUCTOR
$$\frac{\vdash f : t_1 \cdots t_n \xrightarrow{pc'\,[r]} t \qquad pc \leq pc' \qquad \forall j \ \ X, M \vdash v_j : t_j}{pc, X, M \vdash f\, v_1 \cdots v_n : t \ [r]}$$

E-LET
$$\frac{X, M \vdash v : s \qquad s \neq \varnothing \qquad pc, X[x \mapsto s], M \vdash e : t \ [r]}{pc, X, M \vdash \mathsf{let}\, x = v \,\mathsf{in}\, e : t \ [r]}$$

E-BIND
$$\frac{pc, X, M \vdash e_1 : t_1 \ [r_1] \qquad pc \sqcup \Uparrow r_1, X[x \mapsto t_1], M \vdash e_2 : t_2 \ [r_2]}{pc, X, M \vdash \mathsf{bind}\, x = e_1 \,\mathsf{in}\, e_2 : t_2 \ [r_1 \sqcup r_2]}$$

E-HANDLE
$$\frac{pc, X, M \vdash e : t \ [r] \qquad pc, X, M \vdash \vec{h} : r \Rightarrow t \ [r']}{pc, X, M \vdash e \,\mathsf{handle}\, \vec{h} : t \ [r_{|\mathrm{escape}(\vec{h})} \sqcup r']}$$

E-FINALLY
$$\frac{pc, X, M \vdash e_1 : t \ [r] \qquad pc, X, M \vdash e_2 : \star \ [\partial\bot]}{pc, X, M \vdash e_1 \,\mathsf{finally}\, e_2 : t \ [r]}$$

E-SUB
$$\frac{pc, X, M \vdash e : t' \ [r'] \qquad t' \leq t \qquad r' \leq r}{pc, X, M \vdash e : t \ [r]}$$

Clauses

H-VARDONE
$$\frac{\forall \xi \in \Xi \ \ \mathrm{type}(\xi) \leq t' \qquad pc \sqcup \Uparrow r_{|\Xi}, X[\![x \mapsto t']\!], M \vdash e : t \ [r'] \qquad \Uparrow r_{|\Xi} \lhd t}{pc, X, M \vdash (\Xi\, x \to e) : r \Rightarrow t \ [r']}$$

H-VARPROP
$$\frac{\forall \xi \in \Xi \ \ \mathrm{type}(\xi) \leq t' \qquad pc \sqcup \Uparrow r_{|\Xi}, X[\![x \mapsto t']\!], M \vdash e : \star \ [r']}{pc, X, M \vdash (\Xi\, x \to e \,\mathsf{pgt}) : r \Rightarrow \star \ [r' \sqcup (r \sqcup \partial \Uparrow r')_{|\Xi}]}$$

H-WILDDONE
$$\frac{pc \sqcup \Uparrow r_{|\Xi}, X, M \vdash e : t \ [r'] \qquad \Uparrow r_{|\Xi} \lhd t}{pc, X, M \vdash (\Xi\, \_ \to e) : r \Rightarrow t \ [r']}$$

H-WILDPROP
$$\frac{pc \sqcup \Uparrow r_{|\Xi}, X, M \vdash e : \star \ [r']}{pc, X, M \vdash (\Xi\, \_ \to e \,\mathsf{pgt}) : r \Rightarrow \star \ [r' \sqcup (r \sqcup \partial \Uparrow r')_{|\Xi}]}$$

Figure 6.4 – Règles de typage des expressions et des clauses

La règle E-APP considère les applications de fonctions. Le niveau de sécurité pc, qui est une hypothèse de la conclusion, apparaît sur le constructeur $\to$ de la prémisse. Il représente la quotité d'information qui transite, du fait de l'application, depuis le contexte *appelant* vers le contexte *appelé*. De plus, puisque les effets d'une fonction peuvent également révéler des informations sur son identité, les niveaux des premiers doivent être supérieurs ou égaux à celui attaché à la fonction, c'est-à-dire ℓ. En conséquence de ces deux remarques, le corps de la fonction doit *in fine* être typé au niveau $pc \sqcup \ell$. Enfin, le résultat lui-même de l'application est susceptible de porter des informations sur l'identité de la fonction : c'est la raison pour laquelle la troisième prémisse de E-APP contraint le type t à être gardé par ℓ.

Les applications de destructeurs sont typées en supposant donnés un ensemble d'axiomes de la forme $\vdash f : t_1 \cdots t_n \xrightarrow{pc\,[r]} t$ (où n est l'arité de f), qui sont utilisés comme prémisse pour E-DESTRUCTOR. La forme de ces jugements est similaire à celle portant sur les constructeurs. Cependant, ils mentionnent deux annotations, comme les types flèches, permettant de décrire les effets potentiellement produits par le destructeur.

La construction let ne pouvant lier que des valeurs, la formulation de la règle E-LET est presque aussi simple qu'en ML. Notons que la valeur v peut recevoir un schéma brut s, ce qui permet à la valeur x d'être utilisée avec plusieurs types différents dans e. La deuxième prémisse exige que le schéma brut s soit non vide : on souhaite généralement que la valeur v soit bien typée même si elle n'est pas utilisée dans e. Cette prémisse n'est toutefois pas nécessaire pour montrer la correction du système.

Dans la construction $\mathsf{bind}\, x = e_1 \,\mathsf{in}\, e_2$, l'expression e_2 observe, si elle est évaluée, que e_1 n'a pas levé d'exception. Pour prendre en compte ce canal d'information potentiel, la règle E-BIND type l'expression e_2 à un niveau augmenté de $\Uparrow r_1$, la combinaison des niveaux associés à chacune des exceptions que e_1 peut lever. Il s'agit d'une approximation, qui donne de bons résultats en pratique — en particulier dans le cas courant où on sait statiquement que e_1 ne lève pas d'exceptions. Je reviendrai sur ce point à la section 9.2 (page 160).

La rangée $r_{|\mathrm{escape}(\vec{h})} \sqcup r'$ qui apparaît dans la conclusion de la règle E-HANDLE décrit les exceptions qui peuvent s'échapper de la construction. La composante $r_{|\mathrm{escape}(\vec{h})}$ correspond aux exceptions levées par e et rattrapées par aucune des clauses $\vec{h}$, tandis que r' reprend les exceptions levées ou propagées par les clauses. La prémisse $pc, X, M \vdash \vec{h} : r \Rightarrow t\ \ [r']$ est une abréviation pour les jugements $pc, X, M \vdash h_1 : r \Rightarrow t\ \ [r]; \ldots ; pc, X, M \vdash h_n : r \Rightarrow t\ \ [r']$, où $\vec{h} = h_1 \cdots h_n$. Chacun de ces jugements est obtenu, suivant la forme de h_i, par l'une des règles H-VARDONE, H-VARPROP, H-WILDDONE ou H-WILDDONE. Chacune de ces quatre règles inclut une prémisse portant sur le corps de la clause. Comme pour la construction bind, cette expression observe, si elle est évaluée, que celle précédant handle a levé une exception appartenant à Ξ : pour prendre cette potentialité de flot d'information en compte, les quatre règles typent l'expression e à un niveau augmenté de $\Uparrow r_{|\Xi}$. Les clauses typées par les règles H-VARDONE et H-WILDDONE retournent le résultat produit par l'expression e : ainsi le type t et l'effet $[r']$ donnés par la prémisse sont simplement reportés sur la conclusion. La prémisse $\Uparrow r_{|\Xi} \lhd t$ reflète quant à elle le fait que la valeur produite permet de déterminer, au niveau de la construction handle, si la clause particulière a été exécutée. D'autre part, dans les règles H-VARPROP et H-WILDPROP, le type produit par l'expression e n'a pas d'importance, puisque les clauses typées par ces règles produisent toujours une exception. Celle-ci est décrite par l'effet mentionné dans la conclusion de la règle, formé de deux composantes : la rangée r' décrit les exceptions levées par e, tandis que la rangée $(r \sqcup \partial\Uparrow r')_{|\Xi}$ correspond à la propagation de l'exception rattrapée. La rangée uniforme $\partial\Uparrow r'$ reflète la prédominance des exceptions levées par e vis-à-vis de l'exception rattrapée qui n'est propagée que dans le cas où e produit une valeur.

Notons enfin que, pour les clauses de la forme forme $\Xi\, x \to e$ ou $\Xi\, x \to e\,\mathsf{pgt}$, les types des exceptions de l'ensemble Ξ doivent, selon la première prémisse des règles H-VARDONE et H-VARPROP, avoir un super-type t'. Cela permet de préserver la linéarité des dérivations de typage principales, par rapport à la taille des programmes : l'expression e est ainsi typée dans un environnement

$$\textsc{store}\ \frac{\forall m \notin \mathrm{dom}(M)\ \ \mu(m) = \mathsf{null} \qquad \forall m \in \mathrm{dom}(M)\ \ \varnothing, M \vdash \mu(m) : M(m)}{M \vdash \mu} \qquad \textsc{conf}\ \frac{\star, X, M \vdash e : t\ \ [r] \qquad M \vdash \mu}{X \vdash e \,/\, \mu : t\ \ [r]}$$

Figure 6.5 – Règles de typage des états mémoire et des configurations

unique, où la variable x reçoit le type t'. Notons que cette restriction correspond à celle présente dans les langages SML et Caml où les *patterns* apparaissant dans les constructions handle (pour SML) et try . . . with (pour Caml) lient également les variables de manière monomorphe.

Suivant les travaux de Myers, la règle E-FINALLY type les deux sous-expressions e_1 et e_2 au même niveau pc : si j'avais considéré e_1 finally e_2 comme du sucre syntaxique pour $(\mathsf{bind}\, x = e_1\ \mathsf{in}\ e_2; x)$ handle $\Xi\_ \to e_2$ pgt où $x \notin \mathrm{ftv}(e_2)$, un typage moins précis aurait été obtenu, e_2 étant considérée à un niveau augmenté de $\Uparrow r$. L'introduction d'une construction spécifique pour cet idiome permet au contraire de prendre en compte le fait que e_2 est *toujours* exécutée, quel que soit le résultat de e_1. Zdancewic et Myers [ZM01a, ZM02] ont montré comment des continuations linéaires ordonnées permettaient de donner une solution générale à ce problème. La règle donnée par Myers dans sa thèse [Mye99a, Mye99b] pour la construction finally n'est cependant pas correcte : elle ne prend pas encompte le fait que, en observant une exception levée par e_1, on peut déduire que e_2 a termine normalement. Pour corriger cela, je contrains l'expression e_2 à ne pas divulguer d'information en levant une exception : son effet doit être la rangée constante $\partial\bot$. Ce choix peut paraître restrictif ; je crois cependant qu'il s'agit d'un bon compromis entre l'expressivité et la simplicité du système. Il sera discuté à la section 9.2 (page 160). La règle de Myers comporte une deuxième erreur : sont effet global devrait être $X_1 \oplus X_2$, et non $X_1[\underline{\mathsf{n}} := \underline{\emptyset}] \oplus X_2$, puisque la terminaison normale de l'expression complète implique celle de e_1. Cela est pris en compte dans la règle E-FINALLY, même si je n'associe pas explicitement un niveau à la terminaison normale, voir la section 9.2 (page 160). Ces deux erreurs dans le système de Myers ont été découvertes par mon approche formelle [Mye01].

La règle STORE donne la condition de bon typage d'un état mémoire μ par un environnement mémoire M. J'effectue un léger abus de notation en étendant les jugements sur les valeurs aux blocs mémoire. Remarquons que, aucun jugement sur le bloc mémoire null n'étant dérivable, la seconde prémisse implique $\forall m \in \mathrm{dom}(M)\ \ M(m) \neq \mathsf{null}$. Enfin, la règle CONF est identique à celle de $\mathrm{B}(\mathcal{T})$.

J'ai introduit, au chapitre 5 (page 97), la séquence $e_1; e_2$ comme du sucre syntaxique pour l'expression $\mathsf{bind}\, x = e_1\ \mathsf{in}\ e_2$ où $x \notin \mathrm{fpv}(e_2)$. Il est cependant agréable de disposer par la suite d'une version simplifiée de la règle E-BIND pour cet idiome, dans laquelle les deux expressions sont typées sous le même environnement X :

$$\frac{pc, X, M \vdash e_1 : \star\ \ [r_1] \qquad pc \sqcup \Uparrow r_1, X, M \vdash e_2 : t\ \ [r_2]}{pc, X, M \vdash (e_1; e_2) : t\ \ [r_1 \sqcup r_2]}$$

Le lemme suivant énonce la validité de cette règle.

Lemme 6.2 (Séquence) *Si* $pc, X, M \vdash e_1 : \star\ \ [r_1]$ *et* $pc \sqcup \Uparrow r_1, X, M \vdash e_2 : t\ \ [r_2]$ *alors* $pc, X, M \vdash (e_1; e_2) : t\ \ [r_1 \sqcup r_2]$. □

⌜ *Preuve.* Supposons $pc, X, M \vdash e_1 : t_1\ \ [r_1]$ **(H₁)** et $pc \sqcup \Uparrow r_1, X, M \vdash e_2 : t\ \ [r_2]$ **(H₂)**. Soit x une variable fraîche pour $\mathrm{fpv}(e_2)$ et $\mathrm{dom}(X)$. Par le lemme 6.10 (page 124), (H_2) donne $pc \sqcup \Uparrow r_1, X[\![x \mapsto t_1]\!], M \vdash e_2 : t\ \ [r_2]$ **(1)**. En appliquant E-BIND aux prémisses (1) et (H_2), on obtient le but recherché : $pc, X, M \vdash \mathsf{bind}\, x = e_1\ \mathsf{in}\ e_2 : t\ \ [r_1 \sqcup r_2]$. ⌟

$$\textsc{v-bracket}\quad \frac{X, M \vdash_H v_1 : t \qquad X, M \vdash_H v_2 : t \qquad \ell \in H \qquad \ell \lhd t}{X, M \vdash_H \langle v_1 \mid v_2 \rangle : t}$$

$$\textsc{e-bracket}\quad \frac{\begin{array}{c} pc \sqcup pc', X, M \vdash_H e_1 : t \ [r] \qquad pc \sqcup pc', X, M \vdash_H e_2 : t \ [r] \\ pc' \in H \qquad pc' \lhd t \text{ ou } e_1 \nearrow \text{ ou } e_2 \nearrow \end{array}}{pc, X, M \vdash_H \langle e_1 \mid e_2 \rangle : t \ [r]}$$

$$\textsc{w-bracket}\quad \frac{X, M \vdash_H v : t \qquad \ell \in H \qquad \ell \lhd t}{X, M \vdash_H \langle v \mid \mathsf{null} \rangle : t \qquad X, M \vdash_H \langle \mathsf{null} \mid v \rangle : t}$$

Figure 6.6 – Règles de typage spécifiques pour Core ML²

6.4 Une extension de MLIF(𝒯) à Core ML²

J'introduis maintenant une extension du système de type MLIF(𝒯) à Core ML², dénommée MLIF²(𝒯). Je montre que les jugements de typage de ce nouveau système sont préservés par réduction. Ce résultat constitue le point central de la preuve du théorème de non-interférence donnée à la section 6.6 (page 133). Le système MLIF²(𝒯) constitue en fait une expression simple de l'invariant relatif à deux configurations préservé par MLIF(𝒯) et utilisé dans la preuve de non-interférence.

6.4.1 Jugements et règles de typage

Les jugements de typage de MLIF²(𝒯) ont la même forme que ceux de MLIF(𝒯), mais portent un paramètre supplémentaire, H, qui est un sous-ensemble clos supérieurement de $\mathcal{L}$:

$$\begin{array}{c} X, M \vdash_H v : t \\ pc, X, M \vdash_H v : t \ [r] \\ pc, X, M \vdash_H h : r' \Rightarrow t \ [r] \\ M \vdash_H \mu \\ X \vdash_H e \,/\, \mu : t \ [r] \end{array}$$

Il permet d'introduire temporairement une dichotomie entre des niveaux d'information « hauts » (ceux dans H) et « bas » (dans $\mathcal{L}\backslash H$). Puisque la non-interférence considère deux expressions qui ne diffèrent que par des sous-termes de niveau « haut », le système de type pour Core ML² contraindra les expressions de la forme $\langle e_1 \mid e_2 \rangle$, que nous utilisons pour représenter les différences entre deux expressions Core ML, à avoir un résultat et des effets de niveau « haut ».

Les règles de typage pour Core ML² incluent celles données pour Core ML dans les figures 6.3, 6.4 et 6.5 : chaque occurrence du symbole $\vdash$ dans un jugement doit y être remplacée par $\vdash_H$ pour obtenir un jugement Core ML². Ces règles se contentent de propager l'ensemble H dans les dérivations, sans l'utiliser directement. Ce paramètre intervient naturellement dans les trois règles complémentaires, qui sont données figure 6.6, pour typer les constructions $\langle \cdot \mid \cdot \rangle$. V-BRACKET concerne les crochets qui apparaissent dans les valeurs : elle contraint les deux membres d'une construction $\langle \cdot \mid \cdot \rangle$ à avoir le même type, qui doit avoir un (des) niveau(x) de sécurité « haut », c'est-à-dire être gardé par un élément de H (les deux dernières prémisses peuvent être lues comme « *il existe* $\ell \in H$ *tel que* $\ell \lhd t$ »). La règle E-BRACKET permet de typer les constructions $\langle \cdot \mid \cdot \rangle$ des expressions. Les effets et résultats de deux sous-expressions apparaissant à l'intérieur des crochets sont contraints à avoir un niveau « haut ». Cependant, la quatrième prémisse de la règle est légèrement plus générale que celle de V-BRACKET. Par définition, le prédicat auxiliaire $e \nearrow$ est valide si et seulement si l'expression Core ML e est de la forme $\mathsf{raise}\,\xi\,v$ ou $e'; \mathsf{raise}\,\xi\,v$. Ce critère

syntaxique, qui est préservé par substitution et par réduction, assure que e ne peut produire une valeur, c'est-à-dire que e doit diverger ou se réduire en une exception. Il n'y a pas de moyen, dans la syntaxe des jugements de typage, d'exprimer le fait que une expression ne peut produire une valeur. Pourtant, cette possibilité est requise dans quelques situations, pour obtenir le résultat de préservation du typage par réduction. Ainsi, l'utilisation du prédicat $\nearrow$ dans la dernière prémisse de E-BRACKET peut être vue comme un moyen simple d'obtenir localement cette expressivité supplémentaire. Pour résumer, E-BRACKET contraint le type t à être gardé par un niveau « haut », sauf s'il est connu qu'une des deux sous-expressions ne produira jamais de valeur. Enfin, la règle W-BRACKET, étend les jugements portant sur les valeurs aux blocs mémoire de la forme $\langle \mathsf{null} \mid v \rangle$ ou $\langle v \mid \mathsf{null} \rangle$: elle est similaire à la règle V-BRACKET.

Une dérivation du système MLIF($\mathcal{T}$) peut être vue comme une dérivation MLIF^2($\mathcal{T}$), en décorant chacun de ses jugements par un ensemble H arbitraire.

Lemme 6.3 *Si* $X, M \vdash v : t$ *alors* $X, M \vdash_H v : t$. *Si* $pc, X, M \vdash e : t \ [r]$ *alors* $X, M \vdash_H v : t$. □

Notons que la réciproque de cette propriété est également vraie, mais ne sera pas utilisée dans la suite.

6.4.2 Préservation du typage par réduction

Je montre maintenant que la réduction des configurations Core ML^2 préserve le typage dans ce système étendu. Je commence par une série de lemmes techniques permettant d'analyser et de construire des dérivations de typage. Le premier d'entre eux complète la règle E-SUB : il montre que la paramètre pc des jugements est contravariant.

Lemme 6.4 (Affaiblissement) *Si* $pc' \leq pc$ *et* $pc, X, M \vdash_H e : t \ [r]$ *alors* $pc', X, M \vdash_H e : t \ [r]$. □

⌜ *Preuve.* On procède par induction sur la dérivation de $pc, X, M \vdash_H e : t \ [r]$ **(H$_1$)**.

∘ *Cas* E-VALUE. L'expression e est une valeur v. La prémisse de E-VALUE est $X, M \vdash_H v : t$. En appliquant E-VALUE avec cette même prémisse, on peut également dériver le jugement $pc', X, M \vdash_H v : t \ [r]$.

∘ *Cas* E-RAISE. Le jugement (H$_1$) est de la forme $pc, X, M \vdash_H \mathsf{raise}\,\xi\, v : t \ [\xi : pc; r']$. Cette dérivation se termine par une instance de E-RAISE de prémisse $X, M \vdash_H v : \mathrm{type}(\xi)$. Par une instance de E-RAISE avec cette même prémisse, on peut également dériver le jugement $pc', X, M \vdash_H \mathsf{raise}\,\xi\, v : t \ [\xi : pc'; r']$. Par l'hypothèse $pc' \leq pc$ et E-SUB, on obtient le but : $pc', X, M \vdash_H \mathsf{raise}\,\xi\, v : t \ [\xi : pc; r']$.

∘ *Cas* E-APP. L'expression e est de la forme $v_1\, v_2$. Les prémisses de E-APP sont $X, M \vdash_H v_1 : t' \xrightarrow{pc \sqcup \ell\,[r]\,\ell} t$ **(1)**, $X, M \vdash_H v_2 : t'$ **(2)** et $\ell \lhd t$ **(3)**. L'hypothèse $pc' \leq pc$ implique $pc' \sqcup \ell \leq pc \sqcup \ell$. Par la contravariance du constructeur de type $\to$ vis-à-vis de son deuxième argument, on en déduit que $t' \xrightarrow{pc' \sqcup \ell\,[r]\,\ell} t \leq t' \xrightarrow{pc \sqcup \ell\,[r]\,\ell} t$ **(4)**. Par V-SUB, (1) et (4), on a $X, M \vdash_H v_1 : t' \xrightarrow{pc' \sqcup \ell\,[r]\,\ell} t$ **(5)**. En appliquant E-APP aux prémisses (5), (2) et (3), on obtient le but : $pc', X, M \vdash_H v_1\, v_2 : t \ [r]$.

∘ *Cas* E-DESTRUCTOR. L'expression e est $f\, v_1 \cdots v_n$. Les prémisses de E-DESTRUCTOR sont $\vdash f : t_1 \cdots t_n \xrightarrow{pc''\,[r]} t$ **(1)** et $pc \leq pc''$ **(2)** et $\forall j \ X, M \vdash v_j : t_j$ **(3)**. Par (2) et l'hypothèse $pc' \leq pc$, on a $pc' \leq pc''$ **(4)**. Par une instance de E-DESTRUCTOR de prémisses (1), (4) et (3), on obtient le but : $pc', X, M \vdash_H f\, v_1 \cdots v_n : t \ [r]$.

∘ *Les cas* E-LET, E-BIND, E-HANDLE, E-FINALLY *et* E-SUB s'obtiennent de manière directe à partir l'hypothèse d'induction.

∘ *Cas* E-BRACKET. L'expression e est de la forme $\langle e_1 \mid e_2 \rangle$. Les prémisses sont, pour tout $i \in \{1, 2\}$, $pc \sqcup pc'', X, M \vdash_H e_i : t \ [r]$ **(1)**, $pc'' \in H$ **(2)** et $pc'' \lhd t$ **(3)**. Par l'hypothèse $pc' \leq pc$, on a $pc' \sqcup pc'' \leq pc \sqcup pc''$. En appliquant l'hypothèse d'induction à (1), il vient, pour tout $i \in \{1, 2\}$, $pc' \sqcup pc'', X, M \vdash_H e_i : t \ [r]$ **(4)**. Par une instance de E-BRACKET avec les prémisses (4), (2) et (3), on obtient le but recherché : $pc', X, M \vdash_H \langle e_1 \mid e_2 \rangle : t \ [r]$. ⌟

Le deuxième lemme montre que les jugements de typage sont préservés par projection des expressions. Il s'agit d'un résultat relativement grossier : en effet, il n'utilise pas les deux dernières prémisses des règles V-BRACKET et E-BRACKET.

Lemme 6.5 (Projection) *Soit $i \in \{1,2\}$. Si $X, M \vdash_H v : t$ alors $X, M \vdash_H \lfloor v \rfloor_i : t$. Si $pc, X, M \vdash_H e : t\ [r]$ alors $pc, X, M \vdash_H \lfloor e \rfloor_i : t\ [r]$.* □

⌜ *Preuve.* On procède par induction sur la dérivation donnée en hypothèse.

∘ *Cas* E-BRACKET. L'expression e est de la forme $\langle e_1 \mid e_2 \rangle$ et $\lfloor e \rfloor_i = e_i$. Parmi les prémisses de E-BRACKET, on a $pc \sqcup pc', X, M \vdash_H e_i : t\ [r]$. Par le lemme 6.4, on obtient le but recherché : $pc, X, M \vdash_H e_i : t\ [r]$.

Le cas V-BRACKET est similaire. Tous les autres cas s'obtiennent de manière immédiate à partir de l'hypothèse d'induction. ⌟

Je donne maintenant quatre lemmes qui permettent d'analyser les dérivations qui portent sur des résultats. Le premier est une sorte de réciproque de la règle E-VALUE. Le second permet de manipuler les jugements portant sur une exception. Enfin, les lemmes 6.8 et 6.9 considèrent les résultats de la forme $\langle a_1 \mid a_2 \rangle$.

Lemme 6.6 (Valeur) *$pc, X, M \vdash_H v : t\ [r]$ implique $X, M \vdash_H v : t$.* □

⌜ *Preuve.* Par induction sur la dérivation de $pc, X, M \vdash_H v : t\ [r]$. Puisque v est une valeur, seuls les cas suivants sont à envisager.

∘ *Cas* E-VALUE. La prémisse de la règle est le but recherché.

∘ *Cas* E-SUB. Le but résulte de l'hypothèse d'induction et de V-SUB.

∘ *Cas* E-BRACKET. Le but résulte de l'hypothèse d'induction et de V-BRACKET.

Lemme 6.7 (Exception) *Supposons $\xi \in \Xi$. Si $pc, X, M \vdash_H \mathsf{raise}\,\xi\, v : t\ [r]$ alors $X, M \vdash_H v : \mathrm{type}(\xi)$, $pc \leq r(\xi)$ et, pour tout $\ell \in \mathcal{L}$, on a $pc \sqcup \ell, X, M \vdash_H \mathsf{raise}\,\xi\, v : \star\ [(r \sqcup \partial\ell)_{|\Xi}]$.* □

⌜ *Preuve.* La dérivation de $pc, X, M \vdash_H \mathsf{raise}\,\xi\, v : t\ [r]$ peut se terminer par une instance de E-RAISE ou E-SUB. On se ramène au premier cas par une induction immédiate. La prémisse de E-RAISE est alors notre premier but : $X, M \vdash_H v : \mathrm{type}(\xi)$ **(1)**. On a de plus $r(\xi) = pc$, ce qui donne le deuxième but. Enfin, puisque $\xi \in \Xi$, on en déduit $(r \sqcup \partial\ell)_{|\Xi}(\xi) = pc \sqcup \ell$ **(2)**. Par une instance de E-RAISE, on dérive de (1) et (2) le jugement $pc \sqcup \ell, X, M \vdash_H \mathsf{raise}\,\xi\, v : t'\ [(r \sqcup \partial\ell)_{|\Xi}]$, qui est notre troisième but. ⌟

Lemme 6.8 (Valeur gardée) *Si $X, M \vdash_H \langle v_1 \mid v_2 \rangle : t$ alors il existe $pc \in H$ tel que $pc \lhd t$.* □

⌜ *Preuve.* On procède par induction sur la dérivation du jugement donné en hypothèse. De par la forme de la valeur portée par ce jugement, seuls les deux cas suivants sont à envisager.

∘ *Cas* V-BRACKET. Parmi les prémisses de V-BRACKET, on a $pc \in H$ et $pc \lhd t$.

∘ *Cas* V-SUB. Les prémisses de V-SUB sont $X, M \vdash_H \langle v_1 \mid v_2 \rangle : t'$ **(1)** et $t' \leq t$ **(2)**. Par l'hypothèse d'induction appliquée à (1), il existe $pc \in H$ tel que $pc \lhd t'$. Par la propriété 6.1 (page 116) et (2), on en déduit $pc \lhd t$. ⌟

Lemme 6.9 (Résultat gardé) *Si $pc, X, M \vdash_H \langle a_1 \mid a_2 \rangle : t\ [r]$ alors il existe $pc' \in H$ tel que, pour tout $i \in \{1,2\}$, $pc \sqcup pc', X, M \vdash_H a_i : t\ [r]$ et, si a_1 et a_2 sont des valeurs, $pc' \lhd t$.* □

⌜ *Preuve.* On procède par induction sur la dérivation du jugement $pc, X, M \vdash_H \langle a_1 \mid a_2 \rangle : t\ [r]$ **($\mathbf{H_1}$)**. De par la forme de l'expression portée par ce jugement, seuls les cas suivants sont à envisager.

∘ *Cas* E-VALUE. a_1 et a_2 sont des valeurs. La prémisse de E-VALUE est $X, M \vdash_H \langle a_1 \mid a_2 \rangle : t$ **(1)**. Par le lemme 6.8, il existe $pc' \in H$ tel que $pc' \lhd t$. Par une instance de E-VALUE de prémisse (1), on a

$pc \sqcup pc', X, M \vdash_H \langle a_1 \mid a_2 \rangle : t\ [r]$. En appliquant le lemme 6.5, on obtient $pc \sqcup pc', X, M \vdash_H a_i : t\ [r]$, pour tout $i \in \{1, 2\}$.

∘ *Cas* E-BRACKET. Les prémisses de E-BRACKET sont $pc \sqcup pc', X, M \vdash_H a_i : t\ [r]$ **(1)**, pour tout $i \in \{1, 2\}$, $pc' \in H$ **(2)** et $pc' \lhd t$ ou $a_1 \not\uparrow$ ou $a_2 \not\uparrow$ **(3)**. (1) et (2) donnent le premier but. Si a_1 et a_2 sont des valeurs, alors $a_1 \not\uparrow$ et $a_2 \not\uparrow$ sont faux. On déduit de (3) le deuxième but : $pc' \lhd t$.

∘ *Cas* E-SUB. Les prémisses de E-SUB sont $pc, X, M \vdash_H \langle a_1 \mid a_2 \rangle : t'\ [r']$ **(1)**, $t' \leq t$ **(2)** et $r' \leq r$ **(3)**. Par l'hypothèse d'induction appliquée à (1), il existe $pc' \in H$ tel que $pc \sqcup pc', X, M \vdash_H a_i : t'\ [r']$ **(4)**, pour tout $i \in \{1, 2\}$, et si a_1 et a_2 sont des valeurs, $pc' \lhd t'$ **(5)**. Une instance de E-SUB de prémisses (4), (2) et (3) donne $pc \sqcup pc', X, M \vdash_H a_i : t\ [r]$. Par (5) et la propriété 6.1 (page 116), si a_1 et a_2 sont des valeurs, alors $pc' \lhd t$. ⌟

Les deux lemmes suivants donnent deux propriétés habituelles des systèmes de type. Le premier montre que les jugements sont préservés par extension de l'environnement X ou de l'environnement mémoire M'. Le second est un résultat de substitution.

Lemme 6.10 (Affaiblissement) *Supposons que X' et M' étendent respectivement X et M. Si $pc, X, M \vdash_H e : t\ [r]$ alors $pc, X', M' \vdash_H e : t\ [r]$. Si $X, M \vdash_H v : t$ alors $X', M' \vdash_H v : t$.* □

⌜ *Preuve.* Par induction sur la dérivation donnée en hypothèse. ⌟

Lemme 6.11 (Substitution) *Supposons $X, M \vdash_H v : s$. Soit X' un environnement étendant X. Si $X'[\![x \mapsto s]\!], M \vdash_H v' : t$ alors $X', M \vdash_H v'[x \leftarrow v] : t$. Si $pc, X'[\![x \mapsto s]\!], M \vdash_H e : t\ [r]$ alors $pc, X', M \vdash_H e[x \leftarrow v] : t\ [r]$.* □

⌜ *Preuve.* Supposons $X, M \vdash_H v : s$ **($\mathbf{H_1}$)**. On démontre la propriété par induction sur la structure de la dérivation du jugement donné en hypothèse : $X'[\![x \mapsto s]\!], M \vdash_H v' : t$ ou $pc, X'[\![x \mapsto s]\!], M \vdash_H e : t\ [r]$ **($\mathbf{H_2}$)**.

∘ *Cas* V-VAR. La valeur v' est un identificateur x et la prémisse de V-VAR est $t \in X'[\![x \mapsto s]\!](x)$ **(1)**. Si x est la variable x, alors (1) donne $t \in s$. Ainsi, l'hypothèse (H_1) implique $X, M \vdash_H v : t$ et, par le lemme 6.10, $X', M \vdash_H v : t$. Il s'agit de notre but, puisque $v'[x \leftarrow v] = v$. Sinon, si v' n'est pas x, $X'[\![x \mapsto s]\!](x) = X'(x)$ et, par (1), on obtient $t \in X'(x)$. Par une instance de V-VAR, on en déduit notre but : $X', M \vdash_H x : t$.

∘ *Cas* V-ABS. Le jugement (H_2) étant obtenu par une instance de V-ABS, e est de la forme $\lambda y.e'$ et t de la forme $t'' \xrightarrow{pc\,[r]\,\ell} t'$. La prémisse de V-ABS est $pc, X'[\![x \mapsto s]\!][\![y \mapsto t'']\!], M \vdash_H e : t'\ [r]$ **(1)** avec $x \neq y$ **(2)**. Par (2), on a $X'[\![x \mapsto s]\!][\![y \mapsto t'']\!] = X'[\![y \mapsto t'']\!][\![x \mapsto s]\!]$, ainsi, l'hypothèse d'induction appliquée à (H_1) et (1) donne $pc, X'[\![y \mapsto t'']\!], M \vdash_H e[x \leftarrow v] : t\ [r]$. Par V-ABS, on en déduit $X', M \vdash_H \lambda y.(e[x \leftarrow v]) : t'' \xrightarrow{pc\,[r]\,\ell} t'$ **(3)**. Par (2), $\lambda y.(e[x \leftarrow v]) = (\lambda y.e)[x \leftarrow v]$ donc (3) est le but recherché.

∘ *Cas* V-BRACKET. Le jugement (H_2) étant obtenu par une instance de V-BRACKET, la valeur v est de la forme $\langle v_1 \mid v_2 \rangle$. Les prémisses de V-ABS sont $X'[\![x \mapsto s]\!], M \vdash_H v_i : t$ **(1)**, pour tout $i \in \{1, 2\}$, $pc \in H$ **(2)** et $pc \lhd t$ **(3)**. Soit $i \in \{1, 2\}$. Grâce au lemme 6.5, l'hypothèse (H_1) implique $X, M \vdash_H \lfloor v \rfloor_i : s$ **(4)**. En appliquant l'hypothèse d'induction à (4) et (1), on obtient $X', M \vdash_H v_i[x \leftarrow \lfloor v \rfloor_i] : t$ **(5)**. Par une instance de V-BRACKET avec les prémisses (5), (2) et (3), on dérive le jugement $X', M \vdash_H \langle v_1[x \leftarrow \lfloor v \rfloor_1] \mid v_2[x \leftarrow \lfloor v \rfloor_2] \rangle : t$. Puisque $\langle v_1 \mid v_2 \rangle[x \leftarrow v] = \langle v_1[x \leftarrow \lfloor v \rfloor_1] \mid v_2[x \leftarrow \lfloor v \rfloor_2] \rangle$, il s'agit de notre but.

Les autres cas sont immédiats ou analogues aux précédents. ⌟

Je termine cette série de lemmes par deux énoncés relatifs à la réduction des contextes bind et handle. Il s'agit en fait de fragments de la preuve du théorème 6.16 (page 126). Cependant, afin de partager leur démonstration pour les deux cas où ils interviennent, (pop) et (lift-pop), je les isole sous la forme de deux lemmes indépendants.

Lemme 6.12 (Contexte bind) *Supposons que le contexte* $\mathsf{bind}\, x = [\,]\ \mathsf{in}\ e$ *accepte* a. *Si* $pc, X, M \vdash_H a : t_1\ [r_1]$ *et* $pc, X[\![x \mapsto t_1]\!], M \vdash_H e : t\ [r_2]$ *alors* $pc, X, M \vdash_H (\mathsf{bind}\, x = [\,]\ \mathsf{in}\ e) \ll a : t\ [r_1 \sqcup r_2]$. □

⌜ *Preuve.* Supposons $pc, X, M \vdash_H a : t_1\ [r_1]$ **($\mathbf{H_1}$)** et $pc, X[\![x \mapsto t_1]\!], M \vdash_H e : t\ [r_2]$ **($\mathbf{H_2}$)**. On raisonne par cas suivant la forme du résultat a ; puisque $\mathsf{bind}\,[\,] = x\ \mathsf{in}\ e$ accepte a, seules les deux possibilités suivantes sont à envisager.

∘ *Cas* a *est une valeur* v. Par le lemme 6.6 (page 123), le jugement (H_1) donne $X, M \vdash_H v : t_1$ **(1)**. En appliquant le lemme de substitution (lemme 6.11) aux jugements (1) et (H_2), on obtient : $pc, X, M \vdash_H e[x \leftarrow v] : t\ [r_2]$. E-SUB permet d'en déduire le but : $pc, X, M \vdash_H e[x \leftarrow v] : t\ [r_1 \sqcup r_2]$.

∘ *Cas* a *est une exception* $\mathsf{raise}\,\xi\, v$. Par le lemme 6.7 (page 123), (H_1) donne le jugement $pc, X, M \vdash_H \mathsf{raise}\,\xi\, v : t\ [r_1]$. En appliquant une instance de E-SUB, on obtient le but : $pc, X, M \vdash_H \mathsf{raise}\,\xi\, v : t\ [r_1 \sqcup r_2]$. ⌟

Lemme 6.13 (Contexte handle) *Supposons que le contexte* $[\,]\ \mathsf{handle}\ h_1 \cdots h_n$ *accepte* a. *Si* $pc \sqcup pc', X, M \vdash_H a : t\ [r]$ *et, pour tout* $j \in [1, n]$, $pc, X, M \vdash_H h_j : r \Rightarrow t\ [r']$ *alors* $pc \sqcup pc', X, M \vdash_H ([\,]\ \mathsf{handle}\ h_1 \cdots h_n) \ll a : t\ [r_{|\mathrm{escape}(h_1 \cdots h_n)} \sqcup r']$. *De plus, si* a *est une exception alors* $pc' \lhd t$ *ou* $(([\,]\ \mathsf{handle}\ h_1 \cdots h_n) \ll a)\!\uparrow$. □

⌜ *Preuve.* Supposons $pc \sqcup pc', X, M \vdash_H a : t\ [r]$ **($\mathbf{H_1}$)** et, pour tout $j \in [1, n]$, $pc, X, M \vdash_H h_j : r \Rightarrow t\ [r']$ **($\mathbf{H_2}$)**. On raisonne par cas suivant la forme du résultat a. Puisque ce résultat est accepté par $[\,]\ \mathsf{handle}\ h_1 \cdots h_n$, seules les possibilités suivantes sont à envisager.

∘ *Cas* a *est une valeur* v. Par le lemme 6.6 (page 123), (H_1) donne $X, M \vdash_H v : t$. Par une instance de E-VALUE, ce jugement permet de dériver le but : $pc \sqcup pc', X, M \vdash_H v : t\ [r_{|\mathrm{escape}(h_1 \cdots h_n)} \sqcup r']$.

∘ *Cas* a *est une exception* $\mathsf{raise}\,\xi\, v$ *avec* $\xi \in \mathrm{handled}(h_j)$. Soit $\Xi = \mathrm{handled}(h_j)$. Par le lemme 6.7 (page 123), (H_1) implique $pc' \leq \Uparrow r_{|\Xi}$ **(1)** et $X, M \vdash_H v : \mathrm{type}(\xi)$ **(2)**. On distingue quatre sous-cas, suivant la forme de la clause h_j.

· *Si* $h_j = \Xi\_ \to e$. La dérivation de (H_2) se termine par une instance de H-WILDDONE dont les prémisses sont $pc \sqcup \Uparrow r_{|\Xi}, X, M \vdash_H e : t\ [r']$ **(3)** et $\Uparrow r_{|\Xi} \lhd t$ **(4)**. En appliquant le lemme 6.4 (page 122) au jugement (3), et grâce à (1), on obtient $pc \sqcup pc', X, M \vdash_H e : t\ [r']$. Une instance de E-SUB donne le premier but : $pc \sqcup pc', X, M \vdash_H e : t\ [r_{|\mathrm{escape}(h_1 \cdots h_n)} \sqcup r']$. Par (1), (4) et la propriété 6.1 (page 116), on obtient le deuxième but : $pc' \lhd t$.

· *Si* $h_j = \Xi x \to e$. La dérivation de (H_2) se termine par une instance de H-VARDONE, parmi les prémisses de laquelle on trouve $\mathrm{type}(\xi) \leq t'$ **(5)** et $pc \sqcup \Uparrow r_{|\Xi}, X[\![x \mapsto t']\!], M \vdash_H e : t\ [r']$ **(6)**. Par V-SUB, (2) et (5) donnent $X, M \vdash_H v : t'$ **(7)**. Le lemme 6.11, appliqué aux jugements (7) et (6), donne $pc \sqcup \Uparrow r_{|\Xi}, X, M \vdash_H e[x \leftarrow v] : t\ [r']$. On peut alors conclure de la même manière que pour le sous-cas précédent.

· *Si* $h_j = \Xi\_ \to e\,\mathsf{pgt}$. La dérivation de ($\mathrm{H}_2$) se termine par une instance de H-WILDPROP dont la prémisse est $pc \sqcup \Uparrow r_{|\Xi}, X, M \vdash_H e : t_2\ [r_2]$ **(8)** avec $r' = r_2 \sqcup (r \sqcup \partial\Uparrow r_2)_{|\Xi}$ **(9)**. Par le lemme 6.4 (page 122) et grâce à (1), le jugement (8) peut être affaibli en $pc \sqcup pc', X, M \vdash_H e : t_2\ [r_2]$ **(10)**. Par le lemme 6.7 (page 123), (H_1) donne $pc \sqcup pc' \sqcup \Uparrow r_2, X, M \vdash_H \mathsf{raise}\,\xi\, v : t\ [(r \sqcup \partial\Uparrow r_2)_{|\Xi}]$ **(11)**. En appliquant le lemme 6.2 (page 120) à (10) et (11), on obtient $pc \sqcup pc', X, M \vdash_H (e; \mathsf{raise}\,\xi\, v) : t\ [r_2 \sqcup (r \sqcup \partial\Uparrow r_2)_{|\Xi}]$, qui, grâce à (9), est le premier but. Par définition, on a $(e; \mathsf{raise}\,\xi\, v)\!\uparrow$, ce qui donne le second but.

· *Si* $h_j = \Xi x \to e\,\mathsf{pgt}$. La dérivation de ($\mathrm{H}_2$) se termine par une instance de H-VARPROP, parmi les prémisses de laquelle on trouve $\mathrm{type}(\xi) = t'$ **(12)** et $pc \sqcup \Uparrow r_{|\Xi}, X[\![x \mapsto t']\!], M \vdash_H e : t_2\ [r_2]$ **(13)** avec $r' = r_2 \sqcup (r \sqcup \partial\Uparrow r_2)_{|\Xi}$. Le lemme 6.11, appliqué aux jugements (2) et (13), grâce à (12), donne $pc \sqcup \Uparrow r_{|\Xi}, X, M \vdash_H e[x \leftarrow v] : t\ [r_2]$. On peut alors conclure de la même manière que pour le sous-cas précédent.

∘ *Cas a est une exception* $\mathsf{raise}\,\xi\,v$ *avec* $\xi \in \mathrm{escape}(h_1 \cdots h_n)$. Par le lemme 6.7 (page 123), (H_1) donne $pc, X, M \vdash_H \mathsf{raise}\,\xi\,v : t\ [r_{1|\mathrm{escape}(h_1\cdots h_n)}]$. En appliquant E-SUB, on obtient le but : $pc, X, M \vdash_H \mathsf{raise}\,\xi\,v : t\ [r_{|\mathrm{escape}(h_1\cdots h_n)} \sqcup r']$. ⌟

La préservation du typage par les règles (δ) et (lift-δ), nécessite naturellement de relier la sémantique des primitives aux types donnés aux constructeurs et destructeurs. Je formule pour cela les deux hypothèses suivantes.

Hypothèse 6.14 *Supposons* ***(i)*** $\vdash f : t_1 \cdots t_n \xrightarrow{pc\,[r]} t$, ***(ii)*** $\forall j \in [1,n]\ \varnothing, M \vdash_H v_j : t_j$ *et* ***(iii)*** $M \vdash_H \mu$. *Si* $f\,v_1 \cdots v_n\,/\,\mu -\!\!\!\!\!\!\rightarrow_f e + \dot{\mu}$ *alors il existe* M' *étendant* M *sur* $\mathrm{dom}(\dot{\mu})$ *tel que* $pc, \varnothing, M' \vdash e : t\ [r]$, *et, pour tout* $m \in \mathrm{dom}(\dot{\mu})$, $M' \vdash \dot{\mu}(m)$ *et* $pc \lhd M'(m)$. □

Hypothèse 6.15 *Supposons* ***(i)*** $\vdash f : t_1 \cdots t_n \xrightarrow{pc\,[r]} t$, ***(ii)*** $\forall j \in [1,n]\ \varnothing, M \vdash_H v_j : t_j$ *et* ***(iii)*** $M \vdash_H \mu$. *Si* $\bar{v}/\mu \not\downarrow_f$ *et* $\forall i \in \{1,2\}\ \lfloor \bar{v} \rfloor_i / \lfloor \mu \rfloor_i \downarrow_f$ *alors il existe* $pc' \in H$ *tel que* $\vdash f : t_1 \cdots t_n \xrightarrow{pc \sqcup pc'\,[r]} t$ *et* $pc' \lhd t$. □

Dans ces deux énoncés, les suppositions (i), (ii) et (iii) assurent que la configuration $f\,v_1 \cdots v_n\,/\,\mu$ a le type t dans l'environnement mémoire μ. La première hypothèse correspond à la préservation du typage par (δ), la seconde par (lift-δ).

Je peux maintenant donner le résultat principal de cette section, le théorème de stabilité. Je donne en fait deux énoncés de cette propriété. Le premier (théorème 6.16) permet une preuve par induction simple. Le deuxième (théorème 6.17, page 128) est plus abstrait et ne mentionne pas d'environnement mémoire.

Théorème 6.16 (Stabilité) *Supposons* $e\,/_i\,\mu \to e'\,/_i\,\mu'$ *et* $pc, \varnothing, M \vdash_H e : t\ [r]$ *et* $M \vdash_H \mu$. *Si* $i \in \{1,2\}$, *supposons* $pc \in H$. *Alors il existe un environnement mémoire* μ', *qui étend* μ, *tel que* $pc, \varnothing, M' \vdash_H e' : t\ [r]$ *et* $M' \vdash_H \mu'$. □

⌜ *Preuve.* On procède par induction sur la dérivation de $e\,/_i\,\mu \to e'\,/_i\,\mu'$. On peut supposer, sans perte de généralité, que la dérivation de $pc, \varnothing, M \vdash_H e : t\ [r]$ **$(\mathbf{H_1})$** ne se termine pas par une instance de E-SUB. Par conséquent, elle doit se terminer par une instance de la règle dirigée par la syntaxe qui correspond à la structure de l'expression e.

Je considère tout d'abord les règles qui n'affectent pas l'état mémoire, c'est-à-dire telles que $\mu' = \mu$. Pour ces règles, on peut choisir $M' = M$, de telle sorte que le second but, $M' \vdash_H \mu'$ est identique à la troisième hypothèse : $M \vdash_H \mu$ **$(\mathbf{H_2})$**. Il reste à prouver $pc, \varnothing, M \vdash_H e' : t\ [r]$ **$(\mathbf{C_1})$**.

∘ *Cas (β).* Les expressions e et e' sont respectivement $(\lambda x.e_0)\,v$ et $e_0[x \leftarrow v]$. La dérivation de (H_1) se termine par une instance de E-APP dont les prémisses comprennent les jugements $\varnothing, M \vdash_H \lambda x.e_0 : t' \xrightarrow{pc \sqcup \ell\,[r]\,\ell} t$ **(1)** et $\varnothing, M \vdash_H v : t'$ **(2)**. La dérivation de (1) se termine par une instance de V-ABS, éventuellement suivie par une ou plusieurs instances de V-SUB. Puisque $\to$ est covariant (resp. contravariant) en ses premier et deuxième (resp. troisième et quatrième) paramètres, en appliquant le lemme 6.4 (page 122) et E-SUB à la prémisse de V-ABS, on obtient $pc, \{x \mapsto t''\}, M \vdash_H e_0 : t\ [r]$ **(3)** pour un certain type t'' tel que $t' \leq t''$ **(4)**. Par une instance de V-SUB, on peut dériver le jugement $\varnothing, M \vdash_H v : t''$ **(5)** de (2) et (4). Ainsi, en appliquant le lemme 6.11 (page 124) avec les hypothèses (5) et (3), on obtient le but $(\mathrm{C}_1) : pc, \varnothing, M \vdash_H e_0[x \leftarrow v] : t\ [r]$.

∘ *Cas (let).* Les expressions e et e' sont respectivement $\mathsf{let}\,x = v\ \mathsf{in}\ e_0$ et $e_0[x \leftarrow v]$. La dérivation du jugement (H_1) se termine par une instance de E-LET dont les prémisses sont $\varnothing, M \vdash_H v : s$ et $pc, \{x \mapsto s\}, M \vdash_H e_0 : t\ [r]$. En appliquant le lemme 6.11 (page 124) à ces deux jugements, on obtient le but $(\mathrm{C}_1) : pc, \varnothing, M \vdash_H e_0[x \leftarrow v] : t\ [r]$.

∘ *Cas (pop).* Les expressions e et e' sont respectivement $\mathbb{E}[a]$ et $\mathbb{E} \ll a$. On distingue trois sous-cas suivant la forme du contexte $\mathbb{E}$.

· *Si* $\mathbb{E}$ *est* $\mathsf{bind}\,x = [\,]\ \mathsf{in}\ e_0$. La dérivation du jugement (H_1) se termine par une instance de E-BIND dont les prémisses sont $pc, \varnothing, M \vdash_H a : t_1\ [r_1]$ **(1)** et $pc \sqcup \Uparrow r_1, \{x \mapsto t_1\}, M \vdash_H e_0 : t\ [r_2]$

(2) avec $r = r_1 \sqcup r_2$. Par le lemme 6.4 (page 122), (2) peut être affaibli en $pc, \{x \mapsto t_1\}, M \vdash_H e_0 : t\ [r_2]$ **(3)**. En appliquant le lemme 6.12 aux jugements (1) et (3), on obtient le but (C_1) : $pc, \varnothing, M \vdash_H \mathbb{E} \ll a : t\ [r]$.

· *Si* $\mathbb{E}$ *est* $[\,]\ \mathsf{handle}\ (h_1 \cdots h_n)$. La dérivation de (H_1) se termine par une instance de E-HANDLE dont les prémisses sont $pc, \varnothing, M \vdash_H a : t\ [r_1]$ et, pour tout $i \in [1, n]$, $pc, \varnothing, M \vdash_H h_i : r_1 \Rightarrow t\ [r_2]$ avec $r = r_{1|\mathrm{escape}(h_1 \cdots h_n)} \sqcup r_2$. En appliquant le lemme 6.13 (page 125) à ces jugements, on obtient le but (C_1) : $pc, \varnothing, M \vdash_H \mathbb{E} \ll a : t\ [r]$.

· *Si* $\mathbb{E}$ *est* $[\,]\ \mathsf{finally}\ e_0$. La dérivation de (H_1) se termine par une instance de E-FINALLY dont les prémisses sont $pc, \varnothing, M \vdash_H a : t\ [r]$ et $pc, \varnothing, M \vdash_H e : t_2\ [\partial\bot]$. Soit x une variable fraîche pour $\mathrm{fpv}(e)$. Puisque $pc \sqcup \Uparrow(\partial\bot) = pc$, le lemme 6.2 (page 120) peut être appliqué à ces jugements. On obtient ainsi le but (C_1) : $pc, \varnothing, M \vdash_H (e; a) : t\ [r]$.

∘ *Cas (lift-β).* Les expressions e et e' sont respectivement $\langle v_1 \mid v_2 \rangle\, v$ et $\langle v_1 \lfloor v \rfloor_1 \mid v_2 \lfloor v \rfloor_2 \rangle$. La dérivation de (H_1) se termine par une instance de E-APP dont les prémisses sont $\varnothing, M \vdash_H \langle v_1 \mid v_2 \rangle : t' \xrightarrow{pc \sqcup \ell\,[r]\,\ell} t$ **(1)**, $\varnothing, M \vdash_H v : t'$ **(2)** et $\ell \lhd t$ **(3)**. Par le lemme 6.8 (page 123) et (1), il existe $pc' \in H$ **(4)** tel que $pc' \leq \ell$ **(5)**. Par (3) et la propriété 6.1 (page 116), on obtient $pc' \lhd t$ **(6)**. Soit $i \in \{1, 2\}$. En appliquant le lemme 6.5 (page 123) à (1) et (2), on obtient respectivement $\varnothing, M \vdash_H v_i : t' \xrightarrow{pc \sqcup \ell\,[r]\,\ell} t$ et $\varnothing, M \vdash_H \lfloor v \rfloor_i : t'$. Une instance de E-APP permet de dériver de ces deux prémisses le jugement $pc \sqcup \ell, \varnothing, M \vdash_H v_i \lfloor v \rfloor_i : t\ [r]$, lequel peut être affaibli, par (5) et le lemme 6.4 (page 122), en $pc \sqcup pc', \varnothing, M \vdash_H v_i \lfloor v \rfloor_i : t\ [r]$ **(7)**. Par une instance de E-BRACKET de prémisses (7), (4) et (6), on obtient le but (C_1) : $pc, \varnothing, M \vdash_H \langle v_1 \lfloor v \rfloor_1 \mid v_2 \lfloor v \rfloor_2 \rangle : t\ [r]$.

∘ *Cas (lift-δ).* Les deux expressions e et e' sont respectivement $f\, v_1 \cdots v_n$ et $\langle f \lfloor v_1 \cdots v_n \rfloor_1 \mid f \lfloor v_1 \cdots v_n \rfloor_2 \rangle$, avec $v_1 \cdots v_n / \mu \not\downarrow_f$ **(1)** et, pour tout $j \in \{1, 2\}$, $\lfloor v_1 \cdots v_n \rfloor_j / \lfloor \mu \rfloor_j \downarrow_f$ **(2)**. La dérivation de (H_1) se termine par une instance de E-DESTRUCTOR de prémisses $\vdash f : t_1 \cdots t_n \xrightarrow{pc'\,[r]} t$ **(3)**, $pc \leq pc'$ **(4)** et, pour tout $j \in [1, n]$, $\varnothing, M \vdash_H v_j : t_j$ **(5)**. En appliquant l'hypothèse 6.15 à (3), (5) et (H_2), il existe $pc'' \in H$ **(6)** tel $\vdash f : t_1 \cdots t_n \xrightarrow{pc' \sqcup pc''\,[r]} t$ **(7)**, et $pc'' \lhd t$ **(8)**. Soit $i' \in \{1, 2\}$. Par (5) et le lemme 6.5 (page 123), on a $\varnothing, M \vdash_H \lfloor v_j \rfloor_{i'} : t_j$ **(9)**, pour tout $j \in [1, n]$. Par une instance de E-DESTRUCTOR de prémisses (7), (4) et (9), on obtient $pc \sqcup pc'', \varnothing, M \vdash_H f \lfloor v_1 \cdots v_n \rfloor_{i'} : t\ [r]$ **(10)**. Par une instance de E-BRACKET de prémisses (10), (6) et (8), on en déduit le but (C_1) : $pc, \varnothing, M \vdash_H \langle f \lfloor v_1 \cdots v_n \rfloor_1 \mid f \lfloor v_1 \cdots v_n \rfloor_2 \rangle : t\ [r]$.

∘ *Cas (lift-pop).* Les expressions e et e' sont respectivement $\mathbb{E}[\langle a_1 \mid a_2 \rangle]$ et $\langle \lfloor \mathbb{E} \rfloor_1 \ll a_1 \mid \lfloor \mathbb{E} \rfloor_2 \ll a_2 \rangle$, et $\mathbb{E}$ n'accepte pas $\langle a_1 \mid a_2 \rangle$ **(1)**. On raisonne par cas suivant la forme du contexte $\mathbb{E}$; grâce à (1), seules les deux possibilités suivantes sont à envisager.

· *Si* $\mathbb{E} = \mathsf{bind}\, x = [\,]\ \mathsf{in}\ e_2$. La dérivation de (H_1) se termine par une instance de E-BIND dont les prémisses sont $pc, \varnothing, M \vdash_H \langle a_1 \mid a_2 \rangle : t_1\ [r_1]$ **(2)** et $pc \sqcup \Uparrow r_1, \{x \mapsto t_1\}, M \vdash_H e_2 : t\ [r_2]$ **(3)** avec $r = r_1 \sqcup r_2$ **(4)**. Par le lemme 6.9 (page 123) et (2), il existe $pc' \in H$ **(5)** tel que, pour tout $i' \in \{1, 2\}$, $pc \sqcup pc', \varnothing, M \vdash_H a_{i'} : t_1\ [r_1]$ **(6)**. Par (1), il existe $i' \in \{1, 2\}$ tel que $a_{i'}$ soit une exception **(7)**. Par le lemme 6.7 (page 123) et (6), on en déduit que $pc' \leq \Uparrow r_1$. Ainsi, par le lemme 6.4 (page 122), le jugement (3) peut être affaibli en $pc \sqcup pc', \{x \mapsto t_1\}, M \vdash_H e_2 : t\ [r_2]$ **(8)**. Soit $i' \in \{1, 2\}$. En appliquant le lemme 6.5 (page 123) à (8), on obtient $pc \sqcup pc', \{x \mapsto t_1\}, M \vdash_H \lfloor e_2 \rfloor_{i'} : t\ [r_2]$. Par (6) et le lemme 6.12 (page 125), puis en utilisant (4), $pc \sqcup pc', \varnothing, M \vdash_H (\mathsf{bind}\, x = [\,]\ \mathsf{in}\ \lfloor e_2 \rfloor_{i'}) \ll a_{i'} : t\ [r]$ **(9)** s'ensuit. Par (1), a_1 et a_2 ne sont pas deux valeurs donc il existe $i' \in \{1, 2\}$ tel que $(\mathsf{bind}\, x = [\,]\ \mathsf{in}\ \lfloor e_2 \rfloor_{i'}) \nearrow$ **(10)**. En appliquant E-BRACKET aux prémisses (9), (5) et (10), on obtient le but (C_1) : $pc, \varnothing, M \vdash_H \langle (\mathsf{bind}\, x = [\,]\ \mathsf{in}\ \lfloor e_2 \rfloor_1) \ll a_1 \mid (\mathsf{bind}\, x = [\,]\ \mathsf{in}\ \lfloor e_2 \rfloor_2) \ll a_2 \rangle : t\ [r]$.

· *Si* $\mathbb{E} = [\,]\ \mathsf{handle}\ (h_1 \cdots h_n)$. La dérivation de (H_1) se termine par une instance de E-HANDLE dont les prémisses sont $pc, \varnothing, M \vdash_H \langle a_1 \mid a_2 \rangle : t\ [r_1]$ **(11)** et, pour tout $j \in [1, n]$, $pc, \varnothing, M \vdash_H h_j : r_1 \Rightarrow t\ [r_2]$ **(12)**, avec $r = r_{1|\mathrm{escape}(h_1 \cdots h_n)} \sqcup r_2$ **(13)**. Par le lemme 6.9 (page 123) et (11), il existe $pc' \in H$ **(14)** tel que, pour tout $i' \in \{1, 2\}$, $pc \sqcup pc', \varnothing, M \vdash_H a_{i'} : t\ [r_1]$ **(15)**, et, si a_1 et a_2 sont deux valeurs, $pc' \lhd t$ **(16)**. Soit $i' \in \{1, 2\}$. Par le lemme 6.5 (page 123) et (12), on obtient, pour tout

$j \in [1,n]$, $pc, \varnothing, M \vdash_H \lfloor h_j \rfloor_{i'} : r_1 \Rightarrow t\ [r_2]$ **(17)**. En appliquant le lemme 6.13 (page 125) aux jugements (15) et (17), grâce à (13), on obtient $pc \sqcup pc', \varnothing, M \vdash_H ([\,] \mathsf{\ handle\ } (\lfloor h_1 \rfloor_{i'} \cdots \lfloor h_n \rfloor_{i'})) \ll a_{i'} : t\ [r]$ **(18)** et, si $a_{i'}$ est une exception, $pc' \lhd t$ ou $([\,] \mathsf{\ handle\ } (\lfloor h_1 \rfloor_{i'} \cdots \lfloor h_n \rfloor_{i'})) \ll a_{i'} \nearrow$ **(19)**. Par (16) et (19), on a $pc' \lhd t$ ou $([\,] \mathsf{\ handle\ } (\lfloor h_1 \rfloor_1 \cdots \lfloor h_n \rfloor_1)) \ll a_1 \nearrow$ ou $([\,] \mathsf{\ handle\ } (\lfloor h_1 \rfloor_2 \cdots \lfloor h_n \rfloor_2)) \ll a_2 \nearrow$ **(20)**. Ainsi, en appliquant E-BRACKET aux prémisses (18), (14) et (20), on obtient le but (C_1) : $pc, \varnothing, M \vdash_H \langle ([\,] \mathsf{\ handle\ } (\lfloor h_1 \rfloor_1 \cdots \lfloor h_n \rfloor_1)) \ll a_1 \mid ([\,] \mathsf{\ handle\ } (\lfloor h_1 \rfloor_2 \cdots \lfloor h_n \rfloor_2)) \ll a_2 \rangle : t\ [r]$.

Je considère maintenant les trois règles susceptibles de modifier l'état mémoire.

◦ *Cas (δ).* L'expression e et l'état mémoire μ' sont respectivement $f\, v_1 \cdots v_n$ et $\mu \oslash_i \dot{\mu}$ où $f\, v_1 \cdots v_n \,/\, \mu \multimap\!\!\!\!\!\rightarrow e' + \dot{\mu}$ **(1)**. La dérivation de (H_1) se termine par une instance de E-DESTRUCTOR dont les prémisses sont $\vdash f : t_1 \cdots t_n \xrightarrow{pc'\,[r]} t$ **(2)**, $pc \leq pc'$ **(3)** et, pour tout $j \in [1,n]$, $\varnothing, M \vdash_H v_j : t_j$ **(4)**. Par l'hypothèse 6.14 (page 126) appliquée à (2), (4), (H_2) et (1), il existe M' étendant M sur $\mathrm{dom}(\dot{\mu})$ **(5)** tel que $pc', \varnothing, M' \vdash_H e' : t\ [r]$ **(6)** et, pour tout $m \in \mathrm{dom}(\dot{\mu})$, $\varnothing, M' \vdash_H \dot{\mu}(m) : M'(m)$ **(7)** et $pc' \lhd M'(m)$ **(8)**. Par le lemme 6.4 (page 122), (3) et (6) donnent le premier but : $pc, \varnothing, M' \vdash_H e' : t\ [r]$.

La dérivation de (H_2) se termine par une instance de STORE dont les prémisses sont $\forall m \notin \mathrm{dom}(M)\ \ \mu(m) = \mathsf{null}$ **(9)** et $\forall m \in \mathrm{dom}(M)\ \varnothing, M \vdash_H \mu(m) : M(m)$ **(10)**. Soit $m \notin \mathrm{dom}(M')$. Par (5), on a $m \notin \mathrm{dom}(M)$ et $m \notin \mathrm{dom}(\dot{\mu})$. En utilisant (9), et par $\mu' = \mu \oslash_i \dot{\mu}$, on obtient $\mu'(m) = \mathsf{null}$. On en déduit finalement : $\forall m \notin \mathrm{dom}(M')\ \ \mu'(m) = \mathsf{null}$ **(11)**. Soit $m \in \mathrm{dom}(M')$, on veut montrer $\varnothing, M' \vdash_H \mu'(m) : M'(m)$ **(12)**. Si $m \notin \mathrm{dom}(\dot{\mu})$ alors, par (5), $m \in \mathrm{dom}(M)$ et $\mu'(m) = \mu(m)$. On déduit de (10), par le lemme 6.10 (page 124), que $\varnothing, M' \vdash_H \mu(m) : M(m)$. Supposons maintenant que $m \in \mathrm{dom}(\dot{\mu})$, on distingue alors les sous-cas suivants :

· *Si* $i = \bullet$, alors $\mu'(m) = \dot{\mu}(m)$ et (7) donne immédiatement (12).

· *Si* $i = 1$, alors, par hypothèse, $pc \in H$ **(13)**. Par la propriété 6.1 (page 116), (3) et (8) donnent $pc \lhd \mu'(m)$ **(14)**. Si $\mu(m) = \mathsf{null}$, alors $\mu'(m) = \langle \dot{\mu}(m) \mid \mathsf{null} \rangle$; en appliquant W-BRACKET aux prémisses (7), (13) et (14), on obtient le but (12) : $\varnothing, M' \vdash_H \langle \dot{\mu}(m) \mid \mathsf{null} \rangle : M'(m)$. Si $\mu(m) \neq \mathsf{null}$, alors $\mu'(m) = \langle \dot{\mu}(m) \mid \lfloor \mu(m) \rfloor_2 \rangle$. En appliquant les lemmes 6.5 et 6.10 à (10), on a $\varnothing, M' \vdash_H \lfloor \mu(m) \rfloor_2 : M'(m)$ **(15)**. En appliquant V-BRACKET aux prémisses (7). (15), (13) et (14), on obtient également le but (12) : $\varnothing, M' \vdash_H \langle \dot{\mu}(m) \mid \lfloor \mu(m) \rfloor_2 \rangle : M'(m)$.

· *Le sous-cas où* $i = 2$ est similaire au précédent.

On en déduit : $\forall m \in \mathrm{dom}(M')\ \varnothing, M' \vdash_H \mu'(m) : M'(m)$ **(16)**. En appliquant une instance de STORE aux prémisses (11) et (16), on obtient le second but : $M' \vdash_H \mu'$.

◦ *Cas (context).* Les expressions e et e' sont respectivement $\mathbb{E}[e_0]$ et $\mathbb{E}[e'_0]$ où $e_0 \,/_i\, \mu \to e'_0 \,/_i\, \mu'$. En appliquant l'hypothèse d'induction à la première prémisse de E-BIND, E-HANDLE ou E-FINALLY, on obtient une nouvelle version de celle-ci où M et e_0 sont respectivement remplacés par M' et e'_0, M' étant un environnement mémoire étendant M tel que $M' \vdash_H \mu'$. Puisque M' étend M, par le lemme 6.10 (page 124), la (les) autre(s) prémisse(s) reste(nt) valide(s) en remplaçant M par M'. Ainsi, une nouvelle instance de E-BIND, E-HANDLE ou E-FINALLY permet de conclure.

◦ *Cas (bracket).* Les expressions e et e' sont respectivement $\langle e_1 \mid e_2 \rangle$ et $\langle e'_1 \mid e'_2 \rangle$. On a $e_{i_1} \,/_{i_1}\, \mu \to e'_{i_1} \,/_{i_1}\, \mu'$ **(1)** et $e_{i_2} = e'_{i_2}$, avec $\{i_1, i_2\} = \{1,2\}$. Puisque $\langle e_1 \mid e_2 \rangle$ n'est pas une valeur, sa dérivation doit se terminer par une instance de E-BRACKET dont les prémisses sont $pc \sqcup pc', \varnothing, M \vdash_H e_{i_1} : t\ [r]$ **(2)**, $pc \sqcup pc', \varnothing, M \vdash_H e_{i_2} : t\ [r]$ **(3)**, $pc' \in H$ **(4)** et $pc' \lhd t$ ou $e_{i_1} \nearrow$ ou $e_{i_2} \nearrow$ **(5)**. Grâce à (4), l'hypothèse d'induction peut être appliquée à (1), (2) et (H_2). Il existe donc M' étendant M **(6)** tel que $pc \sqcup pc', \varnothing, M' \vdash_H e'_{i_1} : t\ [r]$ **(7)** et $M' \vdash_H \mu'$ **(8)**. Par le lemme 6.10 (page 124) et (6), (3) implique $pc \sqcup pc', \varnothing, M' \vdash_H e'_{i_2} : t\ [r]$ **(9)**. Puisque le prédicat $\cdot\nearrow$ est préservé par réduction, (5) implique $pc' \lhd t$ ou $e'_{i_1} \nearrow$ ou $e'_{i_2} \nearrow$ **(10)**. Par une instance de E-BRACKET de prémisses (7), (9), (4) et (10), on obtient le premier but : $pc, \varnothing, M \vdash_H \langle e'_1 \mid e'_2 \rangle : t\ [r]$. (8) est le deuxième but. ⌟

Corollaire 6.17 (Stabilité) *Si* $\varnothing \vdash_H e \,/\, \mu : t\ [r]$ *et* $e \,/\, \mu \to e' \,/\, \mu'$ *alors* $\varnothing \vdash_H e' \,/\, \mu' : t\ [r]$. □

Entiers naturels

$$\text{V-INT}\quad \vdash \widehat{n} : \varnothing \to \mathsf{int}\,\bot \qquad\qquad \text{E-ADD}\quad \vdash \widehat{+} : \mathsf{int}\,\ell \cdot \mathsf{int}\,\ell \xrightarrow{\top\,[\partial\bot]} \mathsf{int}\,\ell$$

Références

$$\text{V-UNIT}\quad \vdash () : \varnothing \to \mathsf{unit} \qquad \text{E-REF}\quad \frac{pc \lhd t}{\vdash \mathsf{ref} : t \xrightarrow{pc\,[\partial\bot]} \mathsf{ref}\,t\,\bot} \qquad \text{E-ASSIGN}\quad \frac{pc \sqcup \ell \lhd t}{\vdash := : t \cdot \mathsf{ref}\,t\,\ell \xrightarrow{pc\,[\partial\bot]} \mathsf{unit}} \qquad \text{E-DEREF}\quad \frac{t' \leq t \qquad \ell \lhd t}{\vdash ! : \mathsf{ref}\,t'\,\ell \xrightarrow{\top\,[\partial\bot]} t}$$

Paires

$$\text{V-PAIR}\quad \vdash (\cdot,\cdot) : t_1 \cdot t_2 \to t_1 \times t_2 \qquad\qquad \text{E-PROJ}\quad \vdash \mathsf{proj}_j : t_1 \times t_2 \xrightarrow{\bot\,[\partial\top]} t_j$$

Sommes binaires

$$\text{V-INJ}\quad \vdash \mathsf{inj}_j : t \to (t +^j \star)^{\bot} \qquad \text{E-CASE}\quad \frac{\ell \lhd t}{\vdash \mathsf{case} : (t_1 + t_2)^{\ell} \cdot (t_1 \xrightarrow{pc\sqcup\ell\,[r]\,\ell} t) \cdot (t_2 \xrightarrow{pc\sqcup\ell\,[r]\,\ell} t) \xrightarrow{pc\,[r]} t}$$

Point fixe

$$\text{E-FIX}\quad \frac{\ell \leq pc \qquad \ell \lhd t}{\vdash \mathsf{fix} : (t' \xrightarrow{pc\,[r]\,\ell} t) \xrightarrow{\top\,[\partial\bot]\,\bot} (t' \xrightarrow{pc\,[r]\,\ell} t) \cdot t' \xrightarrow{pc\,[r]} t}$$

Figure 6.7 – Axiomes pour les constructeurs et destructeurs

⌜ *Preuve.* Le jugement $\varnothing \vdash_H e \,/\, \mu : t\ \ [r]$ est obtenu par une instance de CONF dont les prémisses sont $pc, \varnothing, M \vdash_H e : t\ \ [r]$ et $M \vdash_H \mu$. Par $e \,/\, \mu \to e' \,/\, \mu'$ et par le théorème 6.16 (page 126), on en déduit qu'il existe M' tel que $pc, \varnothing, M' \vdash_H e' : t\ \ [r]$ et $M' \vdash_H \mu'$. En appliquant CONF à ces deux jugements, on obtient le but : $\varnothing \vdash_H e' \,/\, \mu' : t\ \ [r]$. ⌟

Je ne donne pas d'énoncé de *progression* pour Core ML et MLIF($\mathcal{T}$), exprimant qu'une configuration bien typée ne peut pas être bloquée : il s'agit d'une propriété indépendante du problème qui m'intéresse ici, qui n'est pas nécessaire pour établir la non-interférence. Elle peut cependant être obtenue par une analyse de cas immédiate sur les expressions du langage, après avoir formulé les hypothèses habituelles sur les constantes. On peut également obtenir ce théorème en montrant que toute configuration bien typée dans MLIF($\mathcal{T}$) est également bien typée dans une variante de B($\mathcal{T}$), pour laquelle la progression est connue.

6.5 Constantes

La figure 6.7 donne les axiomes pour le typage des constantes introduites à la section 5.1.3 (page 102). Je donne également, dans la figure 6.8, des règles de typage dérivées pour les applications de constructeurs et de destructeurs, dont la lecture est probablement plus compréhensible. Elles sont obtenues en combinant les axiomes précédents avec les règles V-CONSTRUCTOR et E-DESTRUCTOR ainsi que V-SUB et E-SUB.

Par V-INT, les constantes entières $\widehat{n}$ ont le type $\mathsf{int}\,\bot$ (et, par covariance de int, $\widehat{\ell}$ pour tout niveau ℓ). E-ADD reflète le fait que le résultat d'une addition donne potentiellement des informations sur ses deux arguments. La formulation de la règle est simplifiée en supposant que ces derniers ont le même niveau ℓ : puisque la dérivation de chaque prémisse peut se terminer par une instance de

Entiers naturels

$$\text{V-INT}\quad X, M \vdash \widehat{n} : \mathsf{int}\star \qquad\qquad \text{E-ADD}\quad \frac{X, M \vdash v_1 : \mathsf{int}\,\ell \qquad X, M \vdash v_2 : \mathsf{int}\,\ell}{\star, X, M \vdash v_1 \mathbin{\widehat{+}} v_2 : \mathsf{int}\,\ell \;\; [\star]}$$

Références

$$\text{V-UNIT}\quad \Gamma, M \vdash () : \mathsf{unit} \qquad\qquad \text{E-REF}\quad \frac{X, M \vdash v : t \qquad pc \lhd t}{pc, X, M \vdash \mathsf{ref}\, v : \mathsf{ref}\, t \star \;\; [\star]}$$

$$\text{E-ASSIGN}\quad \frac{X, M \vdash v_1 : \mathsf{ref}\, t\,\ell \qquad X, M \vdash v_2 : t \qquad pc \sqcup \ell \lhd t}{pc, X, M \vdash v_1 := v_2 : \mathsf{unit} \;\; [\star]} \qquad \text{E-DEREF}\quad \frac{X, M \vdash v : \mathsf{ref}\, t'\,\ell \qquad t' \leq t \qquad \ell \lhd t}{pc, X, M \vdash !\, v : t \;\; [\star]}$$

Paires

$$\text{V-PAIR}\quad \frac{X, M \vdash v_1 : t_1 \qquad X, M \vdash v_2 : t_2}{X, M \vdash (v_1, v_2) : t_1 \times t_2} \qquad\qquad \text{E-PROJ}\quad \frac{X, M \vdash v : t_1 \times t_2}{\star, X, M \vdash \mathsf{proj}_j\, v : t_j \;\; [\star]}$$

Sommes binaires

$$\text{V-INJ}\quad \frac{X, M \vdash v : t}{X, M \vdash \mathsf{inj}_j\, v : (t +^j \star)^\star} \qquad\qquad \text{E-CASE}\quad \frac{X, M \vdash v : (t_1 + t_2)^\ell \qquad \ell \lhd t \qquad X, M \vdash v_1 : t_1 \xrightarrow{pc \sqcup \ell\,[r]\,\ell} t \qquad X, M \vdash v_2 : t_2 \xrightarrow{pc \sqcup \ell\,[r]\,\ell} t}{pc, X, M \vdash v\ \mathsf{case}\ v_1\ v_2 : t \;\; [r]}$$

Point fixe

$$\text{E-FIX}\quad \frac{X, M \vdash v_1 : (t' \xrightarrow{pc\,[r]\,\ell} t) \xrightarrow{\top\,[\partial\bot]\,\bot} (t' \xrightarrow{pc\,[r]\,\ell} t) \qquad X, M \vdash v_2 : t' \qquad \ell \leq pc \qquad \ell \lhd t}{pc, X, M \vdash \mathsf{fix}\, v_1\, v_2 : t \;\; [r]}$$

Figure 6.8 – Règles dérivées pour les constructeurs et destructeurs

V-SUB, on peut ré-écrire E-ADD de manière équivalente en :

$$\text{E-ADD}\quad \frac{X, M \vdash v_1 : \mathsf{int}\,\ell_1 \qquad X, M \vdash v_2 : \mathsf{int}\,\ell_2}{\star, X, M \vdash v_1 \mathbin{\widehat{+}} v_2 : \mathsf{int}(\ell_1 \sqcup \ell_2)\ \ [\star]}$$

Les règles E-REF et E-ASSIGN requièrent $pc \lhd t$ pour assurer que pc est une borne inférieure sur le niveau du bloc mémoire dont le contenu est modifié. La prémisse $\ell \lhd t$ de E-ASSIGN et E-DEREF reflète le fait que l'écriture ou la lecture d'une adresse mémoire peut, indirectement, révéler quelque information sur son identité. Notons enfin que, t' apparaissant en position invariante dans la première prémisse de E-DEREF, il faut en considérer un super-type t de manière à ce que la contrainte $\ell \lhd t$ ne porte que sur le type du *résultat* du déréférencement, et non sur celui du bloc mémoire lui-même.

La règle V-PAIR est standard. Dans E-PROJ, les annotations pc et r ne sont pas contraintes, puisque les projections n'ont pas d'effets. J'écris $(t_1 +^1 t_2)^\ell$ pour $(t_1 + t_2)^\ell$ et $(t_2 +^2 t_1)^\ell$ pour $(t_2 + t_1)^\ell$ dans la règle V-INJ. Dans E-CASE, chaque branche v_j obtient, si elle est exécutée, de l'information sur le tag de la somme. Par conséquent, elle doit être typée sous un niveau $pc \sqcup \ell$, et le type de son résultat doit être gardé par ℓ. En utilisant le codage expliqué section 5.1.3 (page 102), on en déduit des règles de typage pour les Booléens. Par V-INJ, les constantes true et false ont le type $(\mathsf{unit} + \mathsf{unit})^\ell$, pour tout ℓ, lequel peut être noté $\mathsf{bool}\,\ell$. On peut dériver de E-CASE et V-ABS la règle suivante pour les conditionnelles :

$$\frac{\Gamma, M \vdash v : \mathsf{bool}\,\ell \qquad pc \sqcup \ell, \Gamma, M \vdash e_1 : t\ \ [r] \qquad pc \sqcup \ell, \Gamma, M \vdash e_2 : t\ \ [r] \qquad \ell \lhd t}{pc, \Gamma, M \vdash \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : t\ \ [r]}$$

La règle de typage de l'opérateur de point fixe, E-FIX, est identique à celle de ML, aux annotations portées par les types flèches près. Le premier argument de fix, v_1, doit être une fonction. Pour simplifier la formulation de la règle, je suppose qu'elle n'est pas le résultat d'un calcul et ne produit pas d'effets lors de son application, en annotant sa flèche par les constantes $\top$, $\partial\bot$ et $\bot$: cela est suffisant pour le codage de la construction $\mathsf{let\ rec}$ qui nous intéresse. L'argument de cette fonction est la valeur construite par le point fixe, qui doit elle-même être une fonction, de type $t' \xrightarrow{pc\,[r]\,\ell} t$. Les prémisses $\ell \leq pc$ et $\ell \lhd t$ assurent que cette fonction peut être appliquée, ce qui est nécessaire pour dérouler le point fixe. Enfin, le deuxième argument de fix, v_2, est la valeur passée en argument à la fonction définie récursivement. Elle doit avoir le type t', de telle sorte que l'application $\mathsf{fix}\,v_1\,v_2$ produit finalement un résultat de type t avec l'effet r. On peut obtenir, à partir des règles E-LET, V-ABS, E-APP et E-FIX, la règle suivante pour la construction $\mathsf{let\ rec}$ proposée section 5.1.3 (page 102) :

$$\frac{pc', X[\![f \mapsto t'' \xrightarrow{pc'\,[r']\,\bot} t']\!][\![x \mapsto t'']\!], M \vdash e_1 : t'\ \ [r'] \qquad pc, X[\![f \mapsto t'' \xrightarrow{pc'\,[r']\,\bot} t']\!], M \vdash e_2 : t\ \ [r]}{pc, X, M \vdash \mathsf{let\ rec}\ f x = e_1\ \mathsf{in}\ e_2 : t\ \ [r]}$$

Je montre maintenant que ces règles de typage sont correctes, c'est-à-dire vérifient, avec les réductions données section 5.2.4 (page 108), les hypothèses 6.14 et 6.15.

Lemme 6.18 *L'hypothèse 6.14 (page 126) est vérifiée par les primitives* $\widehat{+}$, ref, $:=$, $!$, proj_j, case *et* fix. □

⌜ *Preuve.* Soit f l'une des primitives $\widehat{+}$, $\widehat{\div}$, ref, $:=$, $!$, proj_j, case et fix. Je suppose $\vdash f : t_1 \cdots t_n \xrightarrow{pc\,[r]} t$ **(H₁)**, $\forall j \in [1,n]\ \ \varnothing, M \vdash_H v_j : t_j$ **(H₂)**, $M \vdash_H \mu$ **(H₃)** et $f\,v_1 \cdots v_n \,/\, \mu \multimap_f \!\!\!\!\rightarrow e + \dot\mu$ **(H₄)**. Il faut montrer qu'il existe M' étendant M sur $\mathrm{dom}(\dot\mu)$ tel que $pc, \varnothing, M' \vdash e : t\ \ [r]$ **(C₁)**, $\forall m \in \mathrm{dom}(\dot\mu)\ \ M' \vdash \dot\mu(m)$ **(C₂)** et $pc \lhd M'(m)$ **(C₃)**.

Je raisonne par cas suivant la règle utilisée pour obtenir (H_4). Je considère tout d'abord les règles qui ne modifient pas l'état mémoire, c'est-à-dire telles que $\dot\mu = \varnothing$. Pour celles-ci, il faut choisir $M' = M$, et les buts (C_2) et (C_3) sont immédiats. Il reste à vérifier le but (C_1).

∘ *Cas (add).* Par (H_4) et (H_1), e et t sont respectivement de la forme $\widehat{n}$ et $\mathsf{int}\,\ell$. Par V-INT, V-CONSTRUCTOR et E-VALUE, on obtient le but (C_1) : $pc, \varnothing, M \vdash \widehat{n} : \mathsf{int}\,\ell$ $[r]$.

∘ *Cas (proj).* L'hypothèse (H_4) s'écrit $\mathsf{proj}_j\,(v'_1, v'_2)\,/\,\mu \multimap_{\mathsf{proj}_j} v_j + \varnothing$ (H_4). Par (H_1) et (H_2), on a $\varnothing, M \vdash (v'_1, v'_2) : t'_1 \times t'_2$ **(1)** avec $t = t_j$ **(2)**. La dérivation de (1) se termine par une instance de V-CONSTRUCTOR de prémisse V-PAIR, possiblement suivie d'une ou plusieurs instances de V-SUB. On en déduit que $\varnothing, M \vdash v_j : t_j$, puis par E-VALUE et (2), on obtient le but (C_1) : $pc, \varnothing, M \vdash v_j : t_j$ $[r]$.

∘ *Cas (case).* L'hypothèse (H_4) s'écrit $(\mathsf{inj}_j\, v)\ \mathsf{case}\ v'_1\ v'_2\ /\ \mu \multimap_{\mathsf{case}} v'_j\, v + \varnothing$. Par ($H_1$) et ($H_2$), on a $\varnothing, M \vdash \mathsf{inj}_j\, v : (t'_1 + t'_2)^\ell$ **(1)**, $\varnothing, M \vdash v'_j : t'_j \xrightarrow{pc \sqcup \ell\,[r]\,\ell} t$ **(2)** et $\ell \lhd t$ **(3)**. La dérivation de (H_2) se termine par une instance de V-CONSTRUCTOR possiblement suivie d'une ou plusieurs instances de V-SUB. On en déduit que $\varnothing, M \vdash v : t_j$ **(4)**. Par une instance de E-APP de prémisses (2), (4) et (3), on en déduit le but (C_1) : $pc, \varnothing, M \vdash v'_j\, v : t$ $[r]$.

∘ *Cas (deref).* L'hypothèse (H_4) s'écrit $!\,m/\mu \multimap_{!} \mu(m) + \varnothing$. Par ($H_1$) et ($H_2$), on a $\varnothing, M \vdash m : \mathsf{ref}\,t'\,\ell$ **(1)** et $t' \leq t$ **(2)**. La dérivation de (1) se termine par une instance de V-LOC, possiblement suivie d'une ou plusieurs instances de V-SUB. Par l'invariance de ref pour son premier argument, on en déduit que $M(m) = t'$, et, par (H_3), $\varnothing, M \vdash \mu(m) : t'$. Par V-SUB et E-VALUE, grâce à (2), on obtient le but (C_1) : $pc, \varnothing, M \vdash \mu(m) : t$ $[r]$.

∘ *Cas (fix).* L'hypothèse (H_4) s'écrit $\mathsf{fix}\, v_1\, v_2\ /\ \mu \multimap_f \mathsf{bind}\, x_1 = v_1\, (\lambda x_2.\mathsf{fix}\, v_1\, x_2)\ \mathsf{in}\ x_1\, v_2 + \varnothing$. Par ($H_1$) et ($H_2$), on a $\varnothing, M \vdash v_1 : (t' \xrightarrow{pc\,[r]\,\ell} t) \xrightarrow{\top\,[\partial\bot]\,\bot} (t' \xrightarrow{pc\,[r]\,\ell} t)$ **(1)**, $\varnothing, M \vdash v_2 : t'$ **(2)**, $\ell \leq pc$ **(3)** et $\ell \lhd t$ **(4)**. Par E-FIX, (1) donne $pc, \{x_2 \mapsto t'\}, M \vdash \mathsf{fix}\, v_1\, x_2 : t$ $[r]$. Par V-ABS, on en déduit $\varnothing, M \vdash \lambda x_2.\mathsf{fix}\, v_1\, x_2 : t' \xrightarrow{pc\,[r]\,\ell} t$ **(5)**. Par une instance de E-APP de prémisses (1), (5) et $\bot \lhd t$, on dérive $\top, \varnothing, M \vdash v_1\, (\lambda x_2.\mathsf{fix}\, v_1\, x_2) : t' \xrightarrow{pc\,[r]\,\ell} t$ $[\partial\bot]$ **(6)**. Par E-APP, (2), (3) et (4), on a $pc, \{x_1 \mapsto t' \xrightarrow{pc\,[r]\,\ell} t\}, M \vdash x_1\, v_2 : t$ $[r]$ **(7)**. Par une instance de E-BIND de prémisses (6) et (7), on obtient le but (C_1) : $pc, \varnothing, M \vdash \mathsf{bind}\, x_1 = v_1\, (\lambda x_2.\mathsf{fix}\, v_1\, x_2)\ \mathsf{in}\ x_1\, v_2 : t$ $[r]$.

Je considère enfin les deux règles qui modifient l'état mémoire.

∘ *Cas (ref).* L'hypothèse (H_4) s'écrit $\mathsf{ref}\, v\ /\ \mu \multimap_{\mathsf{ref}} m + \{m \mapsto v\}$ avec $\mu(m) = \mathsf{null}$ **(1)**. Par (H_1) et (H_2), on a $\varnothing, M \vdash v : t'$ **(2)** et $t = \mathsf{ref}\, t'\, \ell$ **(3)**. Par (H_3) et (1), $m \notin \mathrm{dom}(M)$. Soit $M' = M[\![m \mapsto t']\!]$. En appliquant le lemme 6.10 (page 124) à (2), on a $\varnothing, M' \vdash \dot{\mu}'(m) : M'(m)$, ce qui donne le but ($C_2$). V-LOC et E-VALUE permettent de dériver (C_1) : $pc, \varnothing, M' \vdash m : M'(m)$ $[r]$.

∘ *Cas (assign).* L'hypothèse (H_4) s'écrit $m := v\ /\ \mu \multimap_{:=} () + \{m \mapsto v\}$. Par ($H_1$) et ($H_2$), on a $\varnothing, M \vdash m : \mathsf{ref}\, t'\, \ell$ **(1)**, $\varnothing, M \vdash v : t'$ **(2)** et $t = \mathsf{unit}$ **(3)**. Par V-UNIT, V-CONSTRUCTOR et E-VALUE permettent de dériver le jugement $pc, \varnothing, M \vdash () : \mathsf{unit}$ $[r]$ qui est, grâce à (3), le but (C_1). La dérivation de (1) se termine par une instance de V-LOC, possiblement suivie d'une ou plusieurs instances de V-SUB. Par l'invariance de ref pour son premier argument, on en déduit que $M(m) = t'$. Ainsi, (2) est le but (C_2) : $\varnothing, M \vdash \dot{\mu}(m) : M(m)$. ⌟

Lemme 6.19 *L'hypothèse 6.15 (page 126) est vérifiée par les primitives* $\widehat{+}$, ref, :=, !, proj_j, case *et* fix. □

⌜ *Preuve.* Je suppose $\vdash f : t_1 \cdots t_n \xrightarrow{pc\,[r]} t$ **(H_1)**, $\forall j \in [1,n]$ $\varnothing, M \vdash_H v_j : t_j$ **(H_2)** $M \vdash_H \mu$ **(H_3)**, $\bar{v}\,/\,\mu \not\downarrow_f$ **(H_4)** et $\forall i \in \{1,2\}$ $\lfloor \bar{v} \rfloor_i\,/\,\lfloor \mu \rfloor_i \downarrow_f$ **(H_5)**. Je cherche à montrer l'existence de $pc' \in H$ tel que $\vdash f : t_1 \cdots t_n \xrightarrow{pc \sqcup pc'\,[r]} t$ **(C_1)** et $pc' \lhd t$ **(C_2)**. Je considère successivement les différentes primitives.

∘ *Cas* $f = \widehat{+}$. L'hypothèse (H_1) s'écrit $\vdash \widehat{+} : \mathsf{int}\,\ell \cdot \mathsf{int}\,\ell \xrightarrow{\top\,[\partial\bot]} \mathsf{int}\,\ell$ et, par (H_2), pour tout $j \in [1,2]$, $\varnothing, M \vdash v_j : \mathsf{int}\,\ell$ **(1)**. Par (H_4) et (H_5), l'une des valeurs v_1 ou v_2 est de la forme $\langle \cdot \mid \cdot \rangle$. Par le lemme 6.8 (page 123) et (1), on en déduit qu'il existe $pc' \in H$ tel que $pc' \lhd \mathsf{int}\,\ell$, *i.e.* ($C_2$). Puisque $\top \sqcup pc' = \top$, (H_1) est le but (C_1).

∘ *Cas* $f = \mathsf{ref}$. Les hypothèses (H_4) et (H_5) ne pouvant être simultanément satisfaites, ce cas ne peut intervenir.

∘ *Cas* $f = :=$. L'hypothèse (H_1) s'écrit $\vdash := : \mathsf{ref}\, t'\, \ell \cdot t' \xrightarrow{pc\,[\partial\bot]} \mathsf{unit}$ avec $pc \sqcup \ell \lhd t$ **(1)** et, par (H_2), on a $\varnothing, M \vdash v_1 : \mathsf{ref}\, t'\, \ell$ **(2)**. Par (H_4) et (H_5), la valeur v_1 est de la forme $\langle \cdot \mid \cdot \rangle$. Par le lemme 6.8 (page 123), la définition de $\lhd$ pour ref et (2), on en déduit qu'il existe $pc' \in H$ tel que $pc' \leq \ell$. (1) peut alors s'écrire $(pc \sqcup pc') \sqcup \ell \lhd t$. Par E-ASSIGN, on obtient $(\mathrm{C}_1) : \vdash := : \mathsf{ref}\, t'\, \ell \cdot t' \xrightarrow{pc \sqcup pc'\,[\partial\bot]} \mathsf{unit}$ et, par la propriété 6.1 (page 116), $(\mathrm{C}_2) : pc' \lhd t$.

∘ *Cas* $f = !$. L'hypothèse (H_1) s'écrit $\vdash ! : \mathsf{ref}\, t'\, \ell \xrightarrow{\top\,[\partial\bot]} t$ avec $t' \leq t$ **(1)** et $\ell \lhd t$ **(2)** et, par (H_2), on a $\varnothing, M \vdash v_1 : \mathsf{ref}\, t'\, \ell$ **(3)**. Par (H_4) et (H_5), la valeur v_1 est de la forme $\langle \cdot \mid \cdot \rangle$. Par le lemme 6.8 (page 123), la définition de $\lhd$ pour ref et (3), on en déduit qu'il existe $pc' \in H$ tel que $pc' \leq \ell$. Par (2) et la propriété 6.1 (page 116), cela implique le but $(\mathrm{C}_2) : pc' \lhd t$. Puisque $\top \sqcup pc' = \top$, (H_1) est le but (C_1).

∘ *Cas* $f = \mathsf{proj}_j$. L'hypothèse (H_1) s'écrit $\vdash \mathsf{proj}_j : t_1 \times t_2 \xrightarrow{\top\,[\partial\bot]} t_j$ avec $t = t_j$ **(1)** et, par (H_2), on a $\varnothing, M \vdash v_1 : t_1 \times t_2$ **(2)**. Par (H_4) et (H_5), la valeur v_1 est de la forme $\langle \cdot \mid \cdot \rangle$. Par le lemme 6.8 (page 123), la définition de $\lhd$ pour $\times$ et (2), on en déduit qu'il existe $pc' \in H$ tel que $pc' \lhd t_j$. Grâce à (1), il s'agit du but (C_2). Puisque $\top \sqcup pc' = \top$, (H_1) est le but (C_1).

∘ *Cas* $f = \mathsf{case}$. L'hypothèse (H_1) s'écrit $\vdash \mathsf{case} : (t_1 + t_2)^\ell \cdot (t_1 \xrightarrow{pc \sqcup \ell\,[r]\,\ell} t) \cdot (t_2 \xrightarrow{pc \sqcup \ell\,[r]\,\ell} t) \xrightarrow{pc\,[r]} t$ avec $\ell \lhd t$ **(1)** et, par (H_2), on a $\varnothing, M \vdash v_1 : (t_1 + t_2)^\ell$ **(2)**. Par (H_4) et (H_5), la valeur v_1 est de la forme $\langle \cdot \mid \cdot \rangle$. Par le lemme 6.8 (page 123), la définition de $\lhd$ pour $+$ et (2), on en déduit qu'il existe $pc' \in H$ tel que $pc' \leq \ell$ **(3)**. Par (1) et la propriété 6.1 (page 116), (C_2) s'ensuit : $pc' \lhd t$. Par (3), $(pc \sqcup pc') \sqcup \ell = pc \sqcup \ell$, ainsi, par (1) et E-CASE, on obtient (C_1) : $\vdash \mathsf{case} : (t_1 + t_2)^\ell \cdot (t_1 \xrightarrow{pc \sqcup \ell\,[r]\,\ell} t) \cdot (t_2 \xrightarrow{pc \sqcup \ell\,[r]\,\ell} t) \xrightarrow{pc \sqcup pc'\,[r]} t$.

∘ *Cas* $f = \mathsf{fix}$. Les hypothèses (H_4) et (H_5) ne pouvant être simultanément satisfaites, ce cas ne peut intervenir. ⌟

6.6 Non-interférence

Je montre enfin le résultat principal de cette partie : le théorème de non-interférence. Je suppose pour cela que le langage Core ML est au moins muni des constantes entières. Celles-ci permettent en effet de comparer simplement les résultats produits par deux programmes, une propriété intéressante pour formuler la non-interférence. Soit K un ensemble (fini ou infini) de constructeurs et c un constructeur de types. On dit que c est ***propre*** à K si et seulement si, pour tout constructeur k, si il existe $\vec{t}$ et $\vec{t'}$ tels que $\vdash k : \vec{t} \to c\,\vec{t'}$ alors $k \in K$. Je suppose dans la suite que le constructeur de type int n'est utilisé que pour typer les constantes entières, *i.e.* est propre à l'ensemble $\{\, \widehat{n} \mid n \in \mathbb{Z} \,\}$.

Le système de type $\mathrm{MLIF}^2(\mathcal{T})$ donne des niveaux de sécurité « hauts » (c'est-à-dire dans H) aux valeurs de la forme $\langle v_1 \mid v_2 \rangle$. Par stabilité du typage par réduction, toute expression qui produit une telle valeur doit être annotée par un niveau « haut ». Inversement, aucune expression avec une annotation « basse » ne peut produire une telle valeur, comme exprimé — dans le cas particulier des entiers — par le lemme suivant.

Lemme 6.20 *Soient H un sous-ensemble de $\mathcal{L}$ clos supérieurement. Soit $\ell \notin H$. Si $\star, \varnothing, \varnothing \vdash_H e : \mathsf{int}\,\ell\ [\star]$ et $e \to^* v$ alors $\lfloor v \rfloor_1 = \lfloor v \rfloor_2$.* □

⌜ *Preuve.* Par le corollaire 6.17 (page 128) et CONF, il existe un environnement mémoire M tel que $\varnothing, M \vdash_H v : \mathsf{int}\,\ell$. Puisque int est propre aux constantes entières, une valeur close de type $\mathsf{int}\star$ doit être de la forme $\widehat{n}$ ou $\langle \widehat{n}_1 \mid \widehat{n}_2 \rangle$. Dans le deuxième cas, par le lemme 6.8 (page 123), il existe $pc' \in H$ tel que $pc' \leq \ell$. H étant clos supérieurement, cela implique $\ell \in H$, une contradiction. On a ainsi $v = \widehat{n} = \lfloor v \rfloor_1 = \lfloor v \rfloor_2$. ⌟

J'utilise maintenant la simulation entre Core ML et Core ML^2 établie au chapitre 5 (page 97) pour reformuler ce résultat dans le cadre de Core ML.

Théorème 6.21 (Non-interférence) *Soit ℓ et t_H deux éléments de $\mathcal{L}$ tels que $t_H \not\leq \ell$. Supposons $t_H \lhd t$ et $\star, \{x \mapsto t\}, \varnothing \vdash e : \mathsf{int}\,\ell\ [\star]$. Si, pour tout $i \in \{1,2\}$, $\varnothing, \varnothing \vdash v_i : t$ et $e[x \leftarrow v_i] \to^* v_i'$ alors $v_1' = v_2'$.* □

⌜ *Preuve.* On suppose $t_H \not\leq \ell$ **(1)**, $t_H \lhd t$ **(2)**, $pc, \{x \mapsto t\}, \varnothing \vdash e : \mathsf{int}\,\ell\ [r]$ **(3)** et, pour tout $i \in \{1,2\}$, $\varnothing, \varnothing \vdash v_i : t$ **(4)** et $e[x \leftarrow v_i] \to^* v_i'$ **(5)**. Soit H la clôture supérieure de $\{t_H\}$ (c'est-à-dire $\{t'_H \in \mathcal{L} \mid t_H \leq t'_H\}$. Par le lemme 6.3 (page 122), les jugements (3) et (4) donnent respectivement $pc, \{x \mapsto t\}, \varnothing \vdash_H e : \mathsf{int}\,\ell\ [r]$ **(6)** et $\varnothing, \varnothing \vdash_H v_i : t$ **(7)**, pour tout $i \in \{1,2\}$. Soit $v = \langle v_1 \mid v_2 \rangle$; par un instance de V-BRACKET de prémisses (7) et (2), on a $\varnothing, \varnothing \vdash_H v : t$ **(8)**. En appliquant le lemme de substitution (lemme 6.11, page 124) à (6) et (8), on obtient $pc, \varnothing, \varnothing \vdash_H e[x \leftarrow v] : \mathsf{int}\,\ell\ [r]$ **(9)**. Puisque, pour tout $i \in \{1,2\}$ $\lfloor e[x \leftarrow v] \rfloor_i = e[x \leftarrow v_i]$, par (5) et par le théorème 5.13 (page 110), il existe un résultat a tel que $e[x \leftarrow v] \to^* a$ **(10)** avec $\lfloor a \rfloor_1 = v_1'$ et $\lfloor a \rfloor_2 = v_2'$ **(11)**. Puisque H est clos supérieurement, (1) implique $\ell \notin H$ **(12)**. En appliquant le lemme 6.20 avec les hypothèses (12), (9) et (10), puisque (11) assure que a est une valeur, on obtient $\lfloor a \rfloor_1 = \lfloor a \rfloor_2$. Grâce à (11), on en conclut $v_1' = v_2'$. ⌟

En d'autres termes, t_H et ℓ sont des niveaux de sécurité tels qu'un flot d'information de t_H vers ℓ est prohibé par le treillis $\mathcal{L}$. Supposons alors que le trou x dans l'expression e a un type de niveau « haut » t et e admet le type de niveau « bas » $\mathsf{int}\,\ell$. Alors, quelle que soit la valeur (de type t) placée dans le trou, l'expression e va produire la même valeur (si elle en produit effectivement une). Puisque la terminaison des deux programmes $e[x \leftarrow v_1]$ et $e[x \leftarrow v_2]$ est assurée par les hypothèses, il s'agit d'un énoncé de non-interférence *faible*, *cf.* la section 5.3 (page 108). Pour plus de simplicité dans la formulation du résultat, j'ai restreint mon attention au cas de résultats entiers, qui peuvent être comparés en utilisant l'égalité. Il serait naturellement possible de donner un énoncé plus général, utilisant une notion d'équivalence observationnelle, comme corollaire du théorème 6.21. On pourrait également, dans un autre corollaire, autoriser plusieurs trous — à la place du seul x. Dès lors que le langage est muni de paires, cela revient essentiellement à spécialiser le théorème 6.21 au cas où t est un type produit.

CHAPITRE SEPT

7

Synthèse de types

Le système présenté au chapitre précédent, MLIF($\mathcal{T}$), a permis de donner une preuve de non-interférence relativement simple, où les questions liées à la formation et à l'instanciation des schémas de type n'interviennent pas. Cependant, du point de vue algorithmique, ce système n'est pas satisfaisant : l'obtention de jugements de type principaux nécessite la considération d'ensembles de types bruts infinis, qui n'ont *a priori* pas de représentation finie. De ce fait, MLIF($\mathcal{T}$) ne dispose pas d'algorithme de synthèse — ni même de vérification — des types. Pour pallier à cette limitation, je dérive dans ce chapitre, à partir de MLIF($\mathcal{T}$), un nouveau système de type dans le style de HM($\mathcal{X}$) [OSW99]. Ce système suit la vision habituelle, *intentionnelle*, du polymorphisme. Il diffère de MLIF($\mathcal{T}$) par l'utilisation de variables et de schémas de type : il est en effet basé sur les types et contraintes de la logique $\mathcal{X}$ introduits aux sections 1.3 et 1.4. Comme HM($\mathcal{X}$), ce système a des types principaux et un algorithme d'inférence.

Dans la première section de ce chapitre, je donne la définition des jugements de MLIF($\mathcal{X}$) ainsi que ses règles de typage. Je montre ensuite que ce système vérifie une propriété de non-interférence (théorème 7.5, page 141) similaire à celle donnée pour MLIF($\mathcal{T}$) (théorème 6.21). Elle est obtenue en codant les jugements de MLIF($\mathcal{X}$) comme ensembles de jugements MLIF($\mathcal{T}$). Dans la seconde section du chapitre, je décris un algorithme qui traduit chaque problème de typage pour MLIF($\mathcal{X}$) en une contrainte logique. En d'autres termes, je ramène l'inférence de type pour MLIF($\mathcal{X}$) à la résolution de contraintes pour la logique $\mathcal{X}$.

Notons enfin que je ne considère plus, dans ce chapitre, que les expressions *source* du langage Core ML, c'est-à-dire celles qui ne contiennent pas d'adresses mémoire.

Dans ce chapitre, chaque occurrence du symbole $\star$ doit être lu comme un type quelconque de sorte appropriée pour le contexte.

7.1 Un système de type à base de contraintes

7.1.1 Jugements et règles de typage

Le système MLIF($\mathcal{X}$) est basé sur les types introduits à la section 1.3 (page 24). Dans ce chapitre, j'utilise la meta-variable ρ pour dénoter les rangées d'atomes, c'est-à-dire les types de sorte $\mathsf{Row}_\Xi\,\mathsf{Atom}$ pour un certain Ξ, ainsi que les meta-variable λ et π pour les types de sorte Atom. La meta-variable τ est quant à elle réservée aux types « normaux », c'est-à-dire de sorte

Type. En plus des prédicats de sous-typage ($\leq$) et de garde ($\lhd$), les contraintes utilisées pour formuler le système MLIF($\mathcal{X}$) et générées par l'algorithme d'inférence défini à la section 7.2.1 (page 141) font intervenir, pour chaque partie finie ou co-finie Ξ de $\mathcal{E}$, un prédicat binaire $\leq_\Xi$ de signature $\mathsf{Row}_\Xi\,\mathsf{Atom} \cdot \mathsf{Row}_\Xi\,\mathsf{Atom}$, dont l'interprétation est définie comme suit :

$$\varphi \vdash \rho_1 \leq_\Xi \rho_2 \;\Leftrightarrow\; \forall \xi \in \Xi\ \ \varphi(\rho_1)(\xi) \leq \varphi(\rho_2)(\xi)$$

Les contraintes générées par le système MLIF($\mathcal{X}$) font également intervenir, pour chaque partie ξ finie ou co-finie de $\mathcal{E}$, le prédicat binaire $\leq_\Xi$ de signature $\mathsf{Row}_\Xi\,\mathsf{Atom} \cdot \mathsf{Row}_\Xi\,\mathsf{Atom}$, dont l'interprétation est définie comme suit :

$$\varphi \vdash \rho_1 \leq_\Xi \rho_2 \;\Leftrightarrow\; \forall \xi \in \Xi\ \ \varphi(\rho_1)(\xi) \leq \varphi(\rho_2)(\xi)$$

Ces contraintes peuvent être exprimées en utilisant l'inégalité habituelle et des termes de rangées [Pot03] :

- Si Ξ est en ensemble fini $\{\xi_1, \ldots, \xi_n\}$ (les ξ_i étant distincts) : la contrainte $\rho_1 \leq_\Xi \rho_2$ est alors équivalente à $\exists \alpha_1\beta_1 \cdots \alpha_n\beta_n\alpha\beta.(\rho_1 = (\xi_1 : \alpha_1; \cdots; \xi_n : \alpha_n; \alpha) \wedge \rho_2 = (\xi_1 : \beta_1; \cdots; \xi_n : \beta_n; \beta) \wedge (\bigwedge_{i\in[1,n]} \alpha_i \leq \beta_i))$ pour des variables $\alpha_1, \beta_1, \ldots, \alpha_n, \beta_n, \alpha, \beta$ distinctes et fraîches pour ρ_1 et ρ_2.
- Si Ξ est en ensemble co-fini $\mathcal{E}\backslash\{\xi_1, \ldots, \xi_n\}$ (les ξ_i étant distincts) : la contrainte $\rho_1 \leq_\Xi \rho_2$ est alors équivalente à $\exists \alpha_1\beta_1 \cdots \alpha_n\beta_n\alpha\beta.(\rho_1 = (\xi_1 : \alpha_1; \cdots; \xi_n : \alpha_n; \alpha) \wedge \rho_2 = (\xi_1 : \beta_1; \cdots; \xi_n : \beta_n; \beta) \wedge \alpha \leq \beta)$ pour des variables $\alpha_1, \beta_1, \ldots, \alpha_n, \beta_n, \alpha, \beta$ distinctes et fraîches pour ρ_1 et ρ_2.

J'introduis également quelques sucres syntaxiques pour désigner certaines formes de contraintes composites qui interviennent dans la formulation des différentes règles du système MLIF($\mathcal{X}$) :

$$\begin{array}{rlll}
\Uparrow\rho \leq \lambda & \text{représente} & \rho \leq \partial\lambda & \\
\lambda \leq \Downarrow\rho & \text{représente} & \partial\lambda \leq \rho & \\
\Uparrow_\Xi\rho \leq \lambda & \text{représente} & \exists\alpha.(\rho \leq_\Xi \alpha \wedge \Uparrow\alpha \leq \lambda) & \text{où } \alpha \# \mathrm{ftv}(\rho, \lambda) \\
\lambda \leq \Downarrow_\Xi\rho & \text{représente} & \exists\alpha.(\lambda \leq \Downarrow\alpha \wedge \alpha \leq_\Xi \rho) & \text{où } \alpha \# \mathrm{ftv}(\rho, \lambda) \\
\Uparrow_\Xi\rho \lhd \tau & \text{représente} & \exists\alpha.(\Uparrow_\Xi\rho \leq \alpha \wedge \alpha \lhd \tau) & \text{où } \alpha \# \mathrm{ftv}(\rho, \tau) \\
\Uparrow_{\Xi_1}\rho_1 \leq \Downarrow_{\Xi_2}\rho_2 & \text{représente} & \exists\alpha.(\Uparrow_{\Xi_1}\rho_1 \leq \alpha \wedge \alpha \leq \Downarrow_{\Xi_2}\rho_2) & \text{où } \alpha \# \mathrm{ftv}(\rho_1, \rho_2)
\end{array}$$

Notons que l'interprétation des symboles $\Uparrow$ et $\Downarrow$ dans les contraintes, qui découle de la définition de ces sucres, correspond aux opérateurs de borne supérieure $\Uparrow$ et inférieure $\Downarrow$ sur les rangées atomiques introduits au chapitre 1 (page 19). Il est en effet aisé de vérifier les équivalences suivantes :

$$\varphi \vdash \Uparrow\rho \leq \lambda \;\Leftrightarrow\; \Uparrow\varphi(\rho) \leq \varphi(\lambda) \;\Leftrightarrow\; \forall \xi \in \mathcal{E}\ \ \varphi(\rho(\xi)) \leq \varphi(\lambda)$$
$$\varphi \vdash \lambda \leq \Downarrow\rho \;\Leftrightarrow\; \varphi(\lambda) \leq \Downarrow\varphi(\rho) \;\Leftrightarrow\; \forall \xi \in \mathcal{E}\ \ \varphi(\lambda) \leq \varphi(\rho(\xi))$$

Un ***environnement*** Γ est une liste d'association dont les clefs sont des variables de programme et les entrées des schémas sans variables de programme libres. L'ensemble des variables de programme définies par un environnement Γ, noté $\mathrm{dpv}(\Gamma)$, est défini par $\mathrm{dpv}(\varnothing) = \varnothing$ et $\mathrm{dpv}(\Gamma; x : \sigma) = \mathrm{dpv}(\Gamma) \cup \{x\}$. Étant donnée une affectation des variables de type φ, l'interprétation d'un environnement Γ, notée $[\![\Gamma]\!]_\varphi$, est un environnement brut défini par les égalités suivantes :

$$\begin{array}{rcl}
[\![\varnothing]\!]_\varphi & = & \varnothing \\
[\![\Gamma; x : \sigma]\!]_\varphi & = & [\![\Gamma]\!]_\varphi[x \mapsto [\![\sigma]\!]_\varphi]
\end{array}$$

Étant donnée une contrainte C, on définit la contrainte $\mathsf{let}\,\Gamma\,\mathsf{in}\,C$ par :

$$\begin{array}{rcl}
\mathsf{let}\,\varnothing\,\mathsf{in}\,C & = & C \\
\mathsf{let}\,\Gamma; x : \sigma\,\mathsf{in}\,C & = & \mathsf{let}\,\Gamma\,\mathsf{in}\,\mathsf{let}\,x\ :\ \sigma\,\mathsf{in}\,C
\end{array}$$

Le système MLIF($\mathcal{X}$) fait intervenir trois formes de jugements de typage, portant respectivement sur les valeurs, expressions et clauses :

$$\begin{array}{c}
C, \Gamma \vdash v : \sigma \\
C, \pi, \Gamma \vdash e : \tau\ \ [\rho] \\
C, \pi, \Gamma \vdash h : \rho' \Rightarrow \tau\ \ [\rho]
\end{array}$$

Règles dirigées par la syntaxe

$$\textsc{v'-var}\ \frac{C \Vdash \exists \Gamma(x)}{C, \Gamma \vdash x : \Gamma(x)} \qquad \textsc{v'-abs}\ \frac{C, \pi, (\Gamma; x : \tau') \vdash e : \tau \ \ [\rho]}{C, \Gamma \vdash \lambda x.e : \tau' \xrightarrow{\pi\,[\rho]\,\star} \tau}$$

$$\textsc{v'-constructor}\ \frac{\vdash k : \forall \bar\alpha[D].\tau_1 \cdots \tau_n \to \tau \qquad \forall j \ \ C, \Gamma \vdash v_j : \tau_j \qquad C \Vdash D}{C, \Gamma \vdash k\, v_1 \cdots v_n : \tau}$$

Règles non dirigées par la syntaxe

$$\textsc{v'-gen}\ \frac{C \wedge D, \Gamma \vdash v : \tau \qquad \bar\alpha \# \operatorname{ftv}(C, \Gamma)}{C \wedge \exists \bar\alpha.D, \Gamma \vdash v : \forall \bar\alpha[D].\tau} \qquad \textsc{v'-inst}\ \frac{C, \Gamma \vdash v : \forall \bar\alpha[D].\tau \qquad C \Vdash D}{C, \Gamma \vdash v : \tau}$$

$$\textsc{v'-sub}\ \frac{C, \Gamma \vdash v : \tau' \qquad C \Vdash \tau' \leq \tau}{C, \Gamma \vdash v : \tau} \qquad \textsc{v'-hide}\ \frac{C, \Gamma \vdash v : \sigma \qquad \bar\alpha \# \operatorname{ftv}(\Gamma, \sigma)}{\exists \bar\alpha.C, \Gamma \vdash v : \sigma}$$

Figure 7.1 – Valeurs

Ces formes de jugements sont similaires à celles utilisées pour le système MLIF($\mathcal{T}$). Les environnements bruts, types bruts et schémas bruts sont respectivement remplacés par des environnements, types et schémas. Puisque je ne considère plus d'expressions avec des adresses mémoires, les jugements ne portent plus d'environnement mémoire. Enfin, chaque jugement débute par une contrainte C, supposée sans variable de programme libre, qui représente une hypothèse sur les variables de type libres du jugement : pour que ce dernier soit valide, la contrainte C doit être satisfiable. Les jugements de typage sont considérés identiques *modulo* équivalence logique des contraintes. Ainsi, la dérivabilité d'un jugement de typage dépend de la *sémantique* de la contrainte qu'il porte, et non de sa *syntaxe*.

Le système MLIF($\mathcal{X}$) est défini par les règles des figures 7.1 et 7.2. Elles sont très similaires à celles données pour le système MLIF($\mathcal{T}$) ; je commente ici les principales différences. Les schémas de type sont formés par V'-GEN et instanciés par V'-INST, ces deux règles étant identiques à celles données dans HM($\mathcal{X}$) [Pot01a]. La règle V'-VAR se contente de reproduire le schéma trouvé dans l'environnement, sans en prendre d'instance, ce qui peut être fait ensuite par V'-INST : l'unique prémisse de V-VAR assure en effet que le schéma est instanciable sous l'hypothèse courante. Les règles V'-HIDE et E'-HIDE permettent de rendre une variable de type locale à une sous-dérivation, ce qui facilite la gestion des noms, sans toutefois permettre de typer plus de programmes. Les types des constantes sont donnés par des jugements de la forme $\vdash k : \forall \bar\alpha[C].\tau_1 \cdots \tau_n \to \tau$ (pour un constructeur k) ou $\vdash f : \forall \bar\alpha[C].\tau_1 \cdots \tau_n \xrightarrow{\pi\,[\rho]} \tau$ (pour un destructeur f). Ces jugements apparaissent comme prémisse des règles V'-CONSTRUCTOR et E'-DESTRUCTOR. Ils associent à chaque constante un « schéma étendu », qui est supposé clos (*i.e.* tel que $\operatorname{ftv}(C, \tau_1, \ldots, \tau_n, \tau) \subseteq \bar\alpha$ ou $\operatorname{ftv}(C, \tau_1, \ldots, \tau_n, \pi, \rho, \tau) \subseteq \bar\alpha$, respectivement), et unique modulo α-renommage des variables de type $\bar\alpha$ (*i.e.* les variables de type $\bar\alpha$ sont supposées liées dans $C, \tau_1, \ldots, \tau_n, \tau$ ou $C, \tau_1, \ldots, \tau_n, \pi, \rho, \tau$, respectivement). Ces jugements sont reliés à ceux utilisés dans le système MLIF($\mathcal{T}$) par l'hypothèse suivante. Elle est utilisée dans la preuve de correction du système MLIF($\mathcal{X}$) (théorème 7.4, page 140).

Hypothèse 7.1 *Si* $\vdash k : \forall \bar\alpha[C].\tau_1 \cdots \tau_n \to \tau$ *(resp.* $\vdash f : \forall \bar\alpha[C].\tau_1 \cdots \tau_n \xrightarrow{\pi\,[\rho]} \tau$*) alors, pour toute solution* φ *de* C, $\vdash k : \varphi(\tau_1) \cdots \varphi(\tau_n) \to \varphi(\tau)$ *(resp.* $\vdash f : \varphi(\tau_1) \cdots \varphi(\tau_n) \xrightarrow{\varphi(\pi)\,[\varphi(\rho)]} \varphi(\tau)$*).* □

Les règles relatives aux constantes introduites section 6.5 (page 129) sont données comme exemples

Règles dirigées par la syntaxe

E'-VALUE
$$\frac{C, \Gamma \vdash v : \tau}{C, \star, \Gamma \vdash v : \tau \ [\star]}$$

E'-RAISE
$$\frac{C, \Gamma \vdash v : \mathrm{type}(\xi)}{C, \pi, \Gamma \vdash \mathsf{raise}\, \xi\, v : \star \ [\xi : \pi; \star]}$$

E'-APP
$$\frac{C, \Gamma \vdash v_1 : \tau' \xrightarrow{\pi' \, [\rho] \, \lambda} \tau \qquad C, \Gamma \vdash v_2 : \tau' \qquad C \Vdash \pi \leq \pi' \qquad C \Vdash \lambda \leq \pi' \qquad C \Vdash \lambda \lhd \tau}{C, \pi, \Gamma \vdash v_1\, v_2 : \tau \ [\rho]}$$

E'-DESTRUCTOR
$$\frac{\vdash f : \forall \bar{\alpha}[D].\tau_1 \cdots \tau_n \xrightarrow{\pi' \, [\rho]} \tau \qquad \forall j \ \ C, \Gamma \vdash v_j : \tau_j \qquad C \Vdash D \qquad C \Vdash \pi \leq \pi'}{\pi, C, \Gamma \vdash f\, v_1 \cdots v_n : \tau \ [\rho]}$$

E'-LET
$$\frac{C, \Gamma \vdash v : \sigma \qquad C, \pi, (\Gamma; x : \sigma) \vdash e : \tau \ [\rho]}{C, \pi, \Gamma \vdash \mathsf{let}\, x = v\, \mathsf{in}\, e : \tau \ [\rho]}$$

E'-BIND
$$\frac{C, \pi, \Gamma \vdash e_1 : \tau_1 \ [\rho_1] \qquad C, \pi_2, (\Gamma; x : \tau_1) \vdash e_2 : \tau \ [\rho] \qquad C \Vdash \pi \leq \pi_2 \qquad C \Vdash \Uparrow \rho_1 \leq \pi_2 \qquad C \Vdash \rho_1 \leq \rho}{C, \pi, \Gamma \vdash \mathsf{bind}\, x = e_1\, \mathsf{in}\, e_2 : \tau \ [\rho]}$$

E'-HANDLE
$$\frac{C, \pi, \Gamma \vdash e : \tau \ [\rho_1] \qquad C, \pi, \Gamma \vdash \bar{h} : \rho_1 \Rightarrow \tau \ [\rho] \qquad C \Vdash \rho_1 \leq_{\mathrm{escape}(\bar{h})} \rho}{C, \pi, \Gamma \vdash e\, \mathsf{handle}\, \bar{h} : \tau \ [\rho]}$$

E'-FINALLY
$$\frac{C, \pi, \Gamma \vdash e_1 : \tau \ [\rho] \qquad C, \pi, \Gamma \vdash e_2 : \star \ [\rho_2] \qquad C \Vdash \Uparrow \rho_2 \leq \bot}{C, \pi, \Gamma \vdash e_1\, \mathsf{finally}\, e_2 : \tau \ [\rho]}$$

Règles dirigées par la syntaxe (clauses)

H'-VARDONE
$$\frac{C, \pi', (\Gamma; x : \mathrm{types}(\bar{\xi})) \vdash e : \tau \ [\rho'] \qquad C \Vdash \exists\, \mathrm{types}(\bar{\xi}) \qquad C \Vdash \pi \leq \pi' \qquad C \Vdash \Uparrow_{\bar{\xi}} \rho \leq \pi' \qquad C \Vdash \Uparrow_{\bar{\xi}} \rho \lhd \tau}{C, \pi, \Gamma \vdash \bar{\xi}\, x \to e : \rho \Rightarrow \tau \ [\rho']}$$

H'-VARPROP
$$\frac{C, \pi', (\Gamma; x : \mathrm{types}(\bar{\xi})) \vdash e : \star \ [\rho_2] \qquad C \Vdash \exists\, \mathrm{types}(\bar{\xi}) \qquad C \Vdash \pi \leq \pi' \qquad C \Vdash \Uparrow_{\bar{\xi}} \rho \leq \pi' \qquad C \Vdash \rho_2 \leq \rho' \qquad C \Vdash \rho \leq_{\bar{\xi}} \rho' \qquad C \Vdash \Uparrow_{\bar{\xi}} \rho_2 \leq \Downarrow_{\bar{\xi}} \rho'}{C, \pi, \Gamma \vdash \bar{\xi}\, x \to e\, \mathsf{pgt} : \rho \Rightarrow \star \ [\rho']}$$

H'-WILDDONE
$$\frac{C, \pi', \Gamma \vdash e : \tau \ [\rho'] \qquad C \Vdash \pi \leq \pi' \qquad C \Vdash \Uparrow_{\Xi} \rho \leq \pi' \qquad C \Vdash \Uparrow_{\Xi} \rho \lhd \tau}{C, \pi, \Gamma \vdash \Xi\_ \to e : \rho \Rightarrow \tau \ [\rho']}$$

H'-WILDPROP
$$\frac{C, \pi', \Gamma \vdash e : \star \ [\rho_2] \qquad C \Vdash \pi \leq \pi' \qquad C \Vdash \Uparrow_{\Xi} \rho \leq \pi' \qquad C \Vdash \rho_2 \leq \rho' \qquad C \Vdash \rho \leq_{\Xi} \rho' \qquad C \Vdash \Uparrow_{\Xi} \rho_2 \leq \Downarrow_{\Xi} \rho'}{C, \pi, \Gamma \vdash \Xi\_ \to e\, \mathsf{pgt} : \rho \Rightarrow \star \ [\rho']}$$

Règles non dirigées par la syntaxe

E'-SUB
$$\frac{C, \pi, \Gamma \vdash e : \tau' \ [\rho'] \qquad C \Vdash \tau' \leq \tau \qquad C \Vdash \rho' \leq \rho}{C, \pi, \Gamma \vdash e : \tau \ [\rho]}$$

E'-HIDE
$$\frac{C, \pi, \Gamma \vdash e : \tau \ [\rho] \qquad \bar{\alpha} \# \mathrm{ftv}(\pi, \Gamma, \tau, \rho)}{\exists \bar{\alpha}.C, \pi, \Gamma \vdash e : \tau \ [\rho]}$$

Figure 7.2 – Expressions

Entiers naturels

V'-INT
$\vdash \widehat{n} : \forall\varnothing[\mathsf{true}].\varnothing \to \mathsf{int}\,\bot$

E'-ADD
$\vdash \widehat{+} : \forall\alpha[\mathsf{true}].\,\mathsf{int}\,\alpha \cdot \mathsf{int}\,\alpha \xrightarrow{\top\,[\partial\bot]} \mathsf{int}\,\alpha$

Références

V'-UNIT
$\vdash () : \forall\varnothing[\mathsf{true}].\varnothing \to \mathsf{unit}$

E'-REF
$\vdash \mathsf{ref} : \forall\alpha\beta[\beta \lhd \alpha].\alpha \xrightarrow{\beta\,[\partial\bot]} \mathsf{ref}\,\alpha\,\bot$

E'-ASSIGN
$\vdash := : \forall\alpha\beta\gamma[\beta \lhd \alpha \wedge \gamma \lhd \alpha].\alpha \cdot \mathsf{ref}\,\alpha\,\gamma \xrightarrow{\beta\,[\partial\bot]} \mathsf{unit}$

E'-DEREF
$\vdash ! : \forall\alpha\beta\gamma[\alpha \leq \gamma \wedge \beta \lhd \gamma].\,\mathsf{ref}\,\alpha\,\beta \xrightarrow{\top\,[\partial\bot]} \gamma$

Paires

V'-PAIR
$\vdash (\cdot,\cdot) : \forall\alpha_1\alpha_2[\mathsf{true}].\alpha_1 \cdot \alpha_2 \to \alpha_1 \times \alpha_2$

E'-PROJ
$\vdash \mathsf{proj}_j : \forall\alpha_1\alpha_2[\mathsf{true}].\alpha_1 \times \alpha_2 \xrightarrow{\bot\,[\partial\top]} \alpha_j$

Sommes binaires

V'-INJ
$\vdash \mathsf{inj}_j : \forall\alpha\beta[\mathsf{true}].\alpha \to (\alpha +^j \beta)^\bot$

E'-CASE
$\vdash \mathsf{case} : \forall\alpha_1\alpha_2\alpha\beta\gamma\gamma'\delta[\beta \lhd \alpha \wedge \beta \leq \gamma' \wedge \gamma \leq \gamma'].(\alpha_1 + \alpha_2)^\beta \cdot (\alpha_1 \xrightarrow{\gamma'\,[\delta]\,\beta} \alpha) \cdot (\alpha_2 \xrightarrow{\gamma'\,[\delta]\,\beta} \alpha) \xrightarrow{\gamma\,[\delta]} t$

Point fixe

E'-FIX
$\vdash \mathsf{fix} : \forall\alpha\alpha'\beta\gamma\delta[\delta \leq \beta].(\alpha' \xrightarrow{\beta\,[\gamma]\,\delta} \alpha) \xrightarrow{\top\,[\partial\bot]\,\bot} (\alpha' \xrightarrow{\beta\,[\gamma]\,\delta} \alpha) \cdot \alpha' \xrightarrow{\beta\,[\gamma]} \alpha$

Figure 7.3 – Règles de typage des constantes

sur la figure 7.3 (page 139). Il est immédiat de montrer qu'elles correspondent à celles données pour MLIF($\mathcal{T}$) (section 6.5, page 129) comme demandé par l'hypothèse précédente.

Dans la prémisse de E'-RAISE, je fais un léger abus de notation, puisque type(ξ) doit désormais être lu comme un type (sans variable de type libre), et non plus comme un type brut. Cette possibilité est justifiée par le lemme 1.6 (page 25). Enfin, le système MLIF($\mathcal{X}$) effectue la restriction sur la forme des ensembles d'exceptions rattrapées annoncée au chapitre 6 (page 113) : les clauses qui lient l'argument de l'exception rattrapée à une variable ne peuvent désormais considérer qu'un nombre *fini* de noms d'exceptions, on écrit $\bar{\xi}\,x \to e$ et $\bar{\xi}\,x \to e$ pgt. Cette hypothèse technique permet aux règles H'-VARPROP et H'-VARDONE de munir la variable x du schéma clos types($\bar{\xi}$) défini par

$$\mathrm{types}(\bar{\xi}) = \forall\alpha[\bigwedge_{\xi\in\bar{\xi}} \mathrm{type}(\xi) \leq \alpha].\alpha$$

La prémisse $C \Vdash \exists\,\mathrm{types}(\bar{\xi})$ présente dans les deux règles assure que le schéma types($\bar{\xi}$) est instanciable, c'est-à-dire que les types type(ξ) ont un super-type commun.

7.1.2 Correction et non-interférence

Je montre dans cette section la correction du système MLIF($\mathcal{X}$) en traduisant ses jugements dans MLIF($\mathcal{T}$). La preuve est similaire à celle donnée par Pottier [Pot01a] entre HM($\mathcal{X}$) et B($\mathcal{T}$). Le traitement des annotations et des contraintes relatives à l'analyse de flots d'information dans le codage est une difficulté supplémentaire d'ordre purement technique.

Lemme 7.2 (Affaiblissement) *Supposons $C_1 \Vdash C_2$. Si un jugement est dérivable sous l'hypothèse C_2 alors il est dérivable sous l'hypothèse C_1.* □

⌜ *Preuve.* Par induction sur la dérivation initiale. ⌟

Lemme 7.3 *Si $C, \Gamma \vdash v : \sigma$ est dérivable alors $C \Vdash \sigma$.* □

⌜ *Preuve.* Par induction sur la dérivation de $C, \Gamma \vdash v : \sigma$ **($\mathbf{H_1}$)**.

∘ *Cas* V'-VAR. Le jugement (H_1) est $C, \Gamma \vdash x : \Gamma(x)$. La prémisse de V'-VAR est notre but : $C \Vdash \exists \Gamma(x)$.

∘ *Cas* V'-GEN. Le jugement (H_1) est $C \wedge \exists \bar{\alpha}.D, \Gamma \vdash v : \forall \bar{\alpha}[D].\tau$. Le but est $C \wedge \exists \bar{\alpha}.D \Vdash \exists \bar{\alpha}.D$, une tautologie.

∘ *Cas* V'-HIDE. Le jugement (H_1) est $\exists \bar{\alpha}.C, \Gamma \vdash v : \sigma$. Parmi les prémisses de V'-HIDE, on a $C, \Gamma \vdash v : \sigma$ **(1)** et $\bar{\alpha} \# \mathrm{ftv}(\sigma)$ **(2)**. En appliquant l'hypothèse d'induction à (1), on obtient $C \Vdash \exists \sigma$. Les variables libres de $\exists \sigma$ étant parmi celles de σ, par (2), on a $\bar{\alpha} \# \mathrm{ftv}(\exists \sigma)$. On en déduit le but : $\exists \bar{\alpha}.C \Vdash \exists \sigma$.

∘ *Cas* V'-ABS, V'-CONSTRUCTOR, V'-INST *et* V'-SUB. Ces règles produisent un jugement dont le schéma est un simple type. Le but est ainsi une tautologie : $C \Vdash \mathsf{true}$. ⌟

Le théorème suivant donne l'interprétation des jugements MLIF($\mathcal{X}$) dans le système MLIF($\mathcal{T}$) : étant donnée une solution φ de son hypothèse C, chaque jugement dérivable dans le système MLIF($\mathcal{X}$) peut être traduit comme un jugement valide pour MLIF($\mathcal{T}$). Ce dernier est obtenu en appliquant l'affectation φ à chacune des composantes du jugement MLIF($\mathcal{X}$), environnement et types.

Théorème 7.4 (Interprétation) *Supposons $\varphi \vdash C$. Si $C, \Gamma \vdash v : \sigma$ alors $[\![\Gamma]\!]_\varphi, \varnothing \vdash v : [\![\sigma]\!]_\varphi$. Si $C, \pi, \Gamma \vdash e : \tau \ [\rho]$ alors $\varphi(\pi), [\![\Gamma]\!]_\varphi, \varnothing \vdash e : \varphi(\tau) \ [\varphi(\rho)]$.* □

⌜ *Preuve.* On procède par induction sur la dérivation du jugement $C, \Gamma \vdash v : \sigma$ ou $C, \pi, \Gamma \vdash e : \tau \ [\rho]$ **($\mathbf{H_1}$)**. On suppose que $\varphi \vdash C$ **($\mathbf{H_2}$)** et on veut montrer soit $[\![\Gamma]\!]_\varphi, \varnothing \vdash v : [\![\sigma]\!]_\varphi$ **($\mathbf{C_1}$)** soit $\varphi(\pi), [\![\Gamma]\!]_\varphi, \varnothing \vdash e : \varphi(\tau) \ [\varphi(\rho)]$ **($\mathbf{C_2}$)**.

∘ *Cas* V'-VAR. Le jugement (H_1) est $C, \Gamma \vdash x : \Gamma(x)$. Par V-VAR, on a, pour tout $t \in [\![\Gamma]\!]_\varphi(x)$, $[\![\Gamma]\!]_\varphi, \varnothing \vdash x : t$. On en déduit $[\![\Gamma]\!]_\varphi, \varnothing \vdash x : [\![\Gamma(x)]\!]_\varphi$, qui est le but ($C_1$).

∘ *Cas* V'-ABS. Le jugement (H_1) est $C, \Gamma \vdash \lambda x.e : \tau' \xrightarrow{\pi \ [\rho] \ \lambda} \tau$ et la prémisse de V'-ABS est $C, \pi, (\Gamma; x : \tau') \vdash e : \tau \ [\rho]$ **(1)**. En appliquant l'hypothèse d'induction à (1) et (H_2), on obtient le jugement $\varphi(\pi), [\![\Gamma]\!]_\varphi [\![x \mapsto \varphi(\tau')]\!], \varnothing \vdash e : \varphi(\tau) \ [\varphi(\rho)]$. Par une instance de V-ABS, on en déduit le but (C_1) : $[\![\Gamma]\!]_\varphi, \varnothing \vdash \lambda x.e : \varphi(\tau') \xrightarrow{\varphi(\pi) \ [\varphi(\rho)] \ \varphi(\lambda)} \varphi(\tau)$.

∘ *Cas* V'-CONSTRUCTOR. Le jugement (H_1) est $C, \Gamma \vdash k\, v_1 \cdots v_n : \tau$. Les prémisses de V'-CONSTRUCTOR sont $\vdash k : \forall \bar{\alpha}[D].\tau_1 \cdots c_n \to \tau$ **(1)**, pour tout $j \in [1, n]$, $C, \Gamma \vdash v_j : \tau_j$ **(2)** et $C \Vdash D$ **(3)**. (H_2) et (3) impliquent $\varphi \vdash D$. Par (1) et l'hypothèse 7.1 (page 137), on en déduit $\vdash k : \varphi(\tau_1) \cdots \varphi(\tau_n) \to \varphi(\tau)$ **(4)**. En appliquant l'hypothèse d'induction à (H_2) et chacun des jugements (2), on obtient, pour tout $j \in [1, n]$, $[\![\Gamma]\!]_\varphi, \varnothing \vdash v_j : \varphi(\tau_j)$ **(5)**. Par une instance de V-CONSTRUCTOR de prémisses (4) et (5), on obtient le but (C_1) : $[\![\Gamma]\!]_\varphi, \varnothing \vdash k\, v_1 \cdots v_n : \varphi(\tau)$.

∘ *Cas* V'-GEN. Le jugement (H_1) est $C \wedge \exists \bar{\alpha}.D, \Gamma \vdash v : \forall \bar{\alpha}[D].\tau$. Les prémisses de V'-GEN sont $C \wedge D, \Gamma \vdash v : \tau$ **(1)** et $\bar{\alpha} \# \mathrm{ftv}(C, \Gamma)$ **(2)**. Soit $t \in [\![\forall \bar{\alpha}[D].\tau]\!]_\varphi$; il existe φ' tel que $\varphi' \vdash D$ **(3)**, $\varphi'(\tau) \leq t$ **(4)** et $\varphi' = \varphi[\vec{\alpha} \mapsto \vec{t}]$ **(5)**. Par (H_2), (2) et (5), on a $\varphi' \vdash C$, ce qui, avec (3), donne $\varphi' \vdash C \wedge D$ **(6)**. En appliquant l'hypothèse d'induction à (6) et (1), on obtient $\varphi'(\Gamma), \varnothing \vdash v : \varphi'(\tau)$. Par V-SUB et (4), cela donne $\varphi'(\Gamma), \varnothing \vdash v : t$ **(7)**. Or, grâce à (2) et (5), $\varphi'(\Gamma) = [\![\Gamma]\!]_\varphi$. Puisque t est un élément arbitraire de $[\![\forall \bar{\alpha}[D].\tau]\!]_\varphi$, on déduit de (7) le but (C_1) : $[\![\Gamma]\!]_\varphi, \varnothing \vdash v : [\![\forall \bar{\alpha}[D].\tau]\!]_\varphi$.

∘ *Cas* V'-INST. Le jugement (H_1) est $C, \Gamma \vdash v : \tau$. Les prémisses de V'-INST sont $C, \Gamma \vdash v : \forall \bar{\alpha}[D].\tau$ **(1)** et $C \Vdash D$ **(2)**. En appliquant l'hypothèse d'induction à (H_2) et (1), on obtient $[\![\Gamma]\!]_\varphi, \varnothing \vdash v : [\![\forall \bar{\alpha}[D].\tau]\!]_\varphi$ **(3)**. Par (H_2) et (2), on a $\varphi \vdash D$. On en déduit que $\varphi(\tau) \in [\![\forall \bar{\alpha}[D].\tau]\!]_\varphi$. Ainsi (3) implique le but (C_1) : $[\![\Gamma]\!]_\varphi, \varnothing \vdash v : \varphi(\tau)$.

∘ *Cas* V'-HIDE. Le jugement (H_1) est $\exists\bar{\alpha}.C, \Gamma \vdash v : \sigma$. Les prémisses de V'-HIDE sont $C, \Gamma \vdash v : \sigma$ **(1)** et $\bar{\alpha} \# \mathrm{ftv}(\Gamma, \sigma)$ **(2)**. Par (H_2), il existe φ' tel que $\varphi' \vdash C$ **(3)** et $\varphi' = \varphi[\bar{\alpha} \mapsto \bar{t}]$ **(4)**. En appliquant l'hypothèse d'induction à (3) et (1), on obtient $\varphi'(\Gamma), \varnothing \vdash v : \varphi'(\sigma)$ **(5)**. Par (2), on a $\varphi'(\Gamma) = [\![\Gamma]\!]_\varphi$ et $\varphi'(\sigma) = [\![\sigma]\!]_\varphi$. Ainsi, (5) est le but $(\mathrm{C}_1) : [\![\Gamma]\!]_\varphi, \varnothing \vdash v : [\![\sigma]\!]_\varphi$.

∘ *Cas* E'-LET. Le jugement (H_1) est $C, \pi, \Gamma \vdash \mathsf{let}\, x = v \,\mathsf{in}\, e : \tau \;\; [\rho]$. Les prémisses de E'-LET sont $C, \Gamma \vdash v : \sigma$ **(1)** et $C, \pi, (\Gamma; x : \sigma) \vdash e : \tau \;\; [\rho]$ **(2)**. En appliquant l'hypothèse d'induction à (H_2) et (1), puis à (H_2) et (2), on obtient respectivement $[\![\Gamma]\!]_\varphi, \varnothing \vdash v : [\![\sigma]\!]_\varphi$ **(3)** et $\varphi(\pi), [\![\Gamma]\!]_\varphi [\![x \mapsto [\![\sigma]\!]_\varphi]\!], \varnothing \vdash e : \varphi(\tau) \;\; [\varphi(\rho)]$ **(4)**. Par le lemme 7.3, (1) implique $C \Vdash \exists\sigma$. Par (H_2), on en déduit que $[\![\sigma]\!]_\varphi \neq \varnothing$ **(5)**. Ainsi, par une instance de E-LET de prémisses (3), (5) et (4), on obtient le but $(\mathrm{C}_2) : \varphi(\pi), [\![\Gamma]\!]_\varphi, \varnothing \vdash \mathsf{let}\, x = v \,\mathsf{in}\, e : \varphi(\tau) \;\; [\varphi(\rho)]$.

Tous les cas restants sont soit immédiats par l'hypothèse d'induction, soit analogues à l'un des précédents. ⌟

Grâce à ce théorème établissant une correspondance entre les jugements de typage de MLIF($\mathcal{T}$) et MLIF($\mathcal{X}$), je peux donner une traduction dans le deuxième système du théorème de non-interférence établi pour le premier (théorème 6.21, page 134).

Corollaire 7.5 (Non-interférence) *Soit ℓ et t_H deux éléments de $\mathcal{L}$ tels que $t_H \not\leq \ell$. Soit C une contrainte satisfiable telle que $C \Vdash t_H \lhd \tau$ et $C, \pi, \{x \mapsto \tau\} \vdash e : \mathsf{int}\,\ell \;\; [\rho]$. Si, pour tout $i \in \{1, 2\}$, $C, \varnothing \vdash v_i : \tau$ et $e[x \leftarrow v_i] \to^* v_i'$ alors $v_1' = v_2'$.* □

⌜ *Preuve.* La contrainte C étant satisfiable, il existe une affectation φ telle que $\varphi \vdash C$. Ainsi, $C \Vdash t_H \lhd \tau$ implique $t_H \lhd \varphi(\tau)$ **(1)** et, par le théorème 7.4, $C, \pi, \{x \mapsto \tau\} \vdash e : \mathsf{int}\,\ell \;\; [\rho]$ implique $\varphi(\pi), \{x \mapsto \varphi(\tau)\}, \varnothing \vdash e : \mathsf{int}\,\ell \;\; [\varphi(\rho)]$ **(2)** et $C, \varnothing \vdash v_i : \tau$ implique $\varnothing, \varnothing \vdash v_i : \varphi(\tau)$ **(3)**. En appliquant le théorème 6.21 (page 134) à l'hypothèse $t_H \not\leq \ell$, à (1), (2), et (3), et à l'hypothèse $e[x \leftarrow v_i] \to^* v_i'$, on obtient le but recherché : $v_1' = v_2'$. ⌟

7.2 Génération de contraintes

7.2.1 Règles de génération

Un problème **(p)** d'inférence de type consiste en une expression e (le programme à typer), un environnement Γ (donnant les types des variables libres de e), et trois types π, τ et ρ de sortes respectives Atom, Type et $\mathsf{Row}_{\mathcal{E}}\,\mathsf{Atom}$. La question est de déterminer si, dans l'environnement Γ et un contexte de niveau π, l'expression e est bien typée avec le type τ et l'effet ρ, c'est-à-dire s'il existe une contrainte C *satisfiable* telle que le jugement $C, \pi, \Gamma \vdash e : \tau \;\; [\rho]$ **(j)** soit dérivable. Dans cette section, je définis un algorithme qui, étant donné le problème **(p)**, calcule une contrainte C *minimale* (au sens de l'implication) telle que le jugement **(j)** soit dérivable. Cette contrainte est précisément $\mathsf{let}\,\Gamma \,\mathsf{in}\, (\!|\pi \vdash e : \tau \;\; [\rho]|\!)$, le fragment $(\!|\pi \vdash e : \tau \;\; [\rho]|\!)$ étant défini inductivement sur la structure de l'expression e ci-après. Cette première phase de l'algorithme d'inférence est généralement appelée *génération de contrainte*. Une fois la formule $\mathsf{let}\,\Gamma \,\mathsf{in}\, (\!|\pi \vdash e : \tau \;\; [\rho]|\!)$ obtenue, il reste, pour répondre au problème **(p)**, à *résoudre* cette contrainte, c'est-à-dire déterminer si elle est satisfiable. Cette question, qui forme la seconde phase de l'algorithme d'inférence, est étudiée dans la troisième partie de cette thèse.

L'algorithme de génération de contraintes est défini par les égalités de la figure 7.4 (page 143), souvent appelées *règles de génération* : le membre gauche des égalités mentionne les entrées de l'algorithme, et le membre droit donne son résultat. Le deuxième groupe de règles définit, inductivement sur la structure de e, la contrainte $(\!|\pi \vdash e : \tau \;\; [\rho]|\!)$. Le système MLIF($\mathcal{X}$) fait intervenir deux formes de jugements particulières pour les valeurs et les clauses ; de même la fonction de génération utilise, dans ses appels récursifs, deux fonctions auxiliaires pour les valeurs et les clauses, définies respectivement par les règles des premier et troisième groupes de la figure 7.4 (page 143). De manière analogue au cas des expressions, ces règles sont conçues de manière à ce que $\mathsf{let}\,\Gamma \,\mathsf{in}\, (\!|v : \tau|\!)$

(resp. $\mathsf{let}\,\Gamma\,\mathsf{in}\,(\!|\pi \vdash h : \rho' \Rightarrow \tau\ [\rho]|\!)$) soit la contrainte C minimale permettant de dériver le jugement $C, \Gamma \vdash v : \tau$ (resp. $C, \Gamma, \pi \vdash h : \rho' \Rightarrow \tau\ [\rho]$).

Avant de décrire plus précisément les règles de génération, je dois donner quelques indications relatives à la gestion des noms, *i.e.* des variables de programme et de type, dans leur énoncé. Tout d'abord, la formulation des règles donnée dans la figure 7.4 assure que chaque variable (de programme ou de type) libre dans la contrainte produite par l'algorithme est libre dans au moins une des entrées présentées, *i.e.* on a les inclusions suivantes :

$$\begin{aligned} \mathrm{ftv}((\!|\pi \vdash e : \tau\ [\rho]|\!)) &\subseteq \mathrm{ftv}(\pi, \tau, \rho) & \mathrm{fpv}((\!|\pi \vdash e : \tau\ [\rho]|\!)) &\subseteq \mathrm{fpv}(e) \\ \mathrm{ftv}((\!|v : \tau|\!)) &\subseteq \mathrm{ftv}(\tau) & \mathrm{fpv}((\!|v : \tau|\!)) &\subseteq \mathrm{fpv}(v) \\ \mathrm{ftv}((\!|\pi \vdash h : \rho' \Rightarrow \tau\ [\rho]|\!)) &\subseteq \mathrm{ftv}(\pi, \rho', \tau, \rho) & \mathrm{fpv}((\!|\pi \vdash h : \rho' \Rightarrow \tau\ [\rho]|\!)) &\subseteq \mathrm{fpv}(h) \end{aligned}$$

En d'autres termes, l'algorithme de génération n'introduit pas de nouveaux noms libres arbitraires. Cette propriété ne peut pas former à proprement parler un lemme à montrer sur les fonctions $(\!|\cdots|\!)$, puisqu'elle est en fait nécessaire à la bonne formation de la définition donnée figure 7.4. Il me faut par ailleurs donner les hypothèses portant sur le « choix » des variables de type α, $\bar{\alpha}$, β, γ, δ et ε qui apparaissent *liées* dans les membres droits des règles de génération. Dans chacune de ces égalités, ces variables doivent d'une part être choisies distinctes et d'autre part *fraîches* vis-à-vis des entrées de l'algorithme, c'est-à-dire ne pas apparaître libre dans le membre gauche. On peut vérifier que, sous cette hypothèse, la contrainte formée par chacune des règles de l'algorithme ne dépend pas du choix de ces variables, les contraintes étant considérées modulo α-conversion. Là encore, cette propriété est nécessaire pour que l'algorithme défini par la figure 7.4 soit bien une fonction.

La plupart des règles de génération de contraintes reflètent directement les règles de typage du système MLIF($\mathcal{X}$). La principale différence réside dans la gestion de l'environnement de typage. La consultation de l'environnement, effectuée dans la règle V'-VAR, est remplacée par la génération d'une contrainte d'instanciation pour chaque occurrence d'une variable de programme dans l'expression à typer. La définition de $(\!|x : \tau|\!)$ peut être lue : *x a le type τ si et seulement si τ est une instance du schéma de type lié à x*. Cette contrainte acquiert tout son sens lorsqu'elle est placée, plus tard, dans un contexte $\mathsf{let}\,x : \sigma\,\mathsf{in}\,[\,]$. Ces contextes sont engendrés au niveau des constructions du langage qui lient des variables de programme, là où, dans la formulation logique de MLIF($\mathcal{X}$), l'environnement Γ est étendu.

7.2.2 Une présentation alternative de MLIF($\mathcal{X}$)

La présentation de MLIF($\mathcal{X}$) donnée dans les figures 7.1 et 7.2 sépare les règles de formation et d'instanciation des schémas de type des règles dirigées par la syntaxe. Il s'agit d'une bonne spécification du système ; cependant la preuve de la complétude de l'algorithme d'inférence est simplifiée en restreignant la forme des dérivations, de telle sorte que la généralisation ne soit autorisée que dans les constructions let et l'instanciation soit effectuée directement au niveau des références aux variables de programme. Cette restriction consiste à remplacer, dans la définition donnée figures 7.1 et 7.2, les règles V'-VAR, V'-INST, V'-GEN et E'-LET par les deux règles suivantes :

$$\textsc{v'-varinst}\ \frac{\Gamma(x) = \forall\bar{\alpha}[D].\tau}{C \wedge D, \Gamma \vdash x : \tau} \qquad \textsc{e'-genlet}\ \frac{C \wedge D, \Gamma \vdash v : \tau' \qquad \bar{\alpha} \# \mathrm{ftv}(C, \Gamma) \qquad C \wedge \exists\bar{\alpha}.D, \pi, (\Gamma; x : \forall\bar{\alpha}[D].\tau') \vdash e : \tau\ [\rho]}{C \wedge \exists\bar{\alpha}.D, \pi, \Gamma \vdash \mathsf{let}\,x = v\,\mathsf{in}\,e : \tau\ [\rho]}$$

La première est une combinaison de V'-VAR et V'-INST, et la seconde de V'-GEN et E'-LET. Notons que, avec cette nouvelle présentation, tous les jugements portent un simple type, et non un schéma. Il ne s'agit cependant que d'un détail technique : il est en effet facile de vérifier que, sous l'hypothèse $\bar{\alpha} \# \mathrm{ftv}(D, \Gamma)$, les jugements $C, \Gamma \vdash v : \forall\bar{\alpha}[D].\tau$ et $C \wedge D, \Gamma \vdash v : \tau$ sont équivalents.

Dans le reste de cette section, je montre que les deux formulations du système permettent de dériver les mêmes jugements. Pour clarifier les différents énoncés reliant les deux jeux de règles,

Valeurs

$$
\begin{aligned}
(\!|x : \tau|\!) &= x \preceq \tau \\
(\!|\lambda x.e : \tau|\!) &= \exists \alpha\beta\gamma\delta\varepsilon.(\mathsf{let}\, x \,:\, \alpha \,\mathsf{in}\, (\!|\beta \vdash e : \varepsilon \;\; [\gamma]|\!) \wedge \alpha \xrightarrow{\beta\,[\gamma]\,\delta} \varepsilon \leq \tau) \\
(\!|k\, v_1 \cdots v_n : \tau|\!) &= \exists \bar{\alpha}.(D \wedge (\!|v_1 : \tau_1|\!) \wedge \cdots \wedge (\!|v_n : \tau_n|\!) \wedge \tau' \leq \tau) \\
&\qquad\qquad \text{où } \vdash k : \forall \bar{\alpha}[D].\tau_1 \cdots \tau_n \to \tau'
\end{aligned}
$$

Expressions

$$
\begin{aligned}
(\!|\pi \vdash v : \tau \;\; [\rho]|\!) &= (\!|v : \tau|\!) \\
(\!|\pi \vdash \mathsf{raise}\, \xi\, v : \tau \;\; [\rho]|\!) &= \exists \alpha.((\!|v : \mathrm{type}(\xi)|\!) \wedge (\xi : \pi; \alpha) \leq \rho) \\
(\!|\pi \vdash v_1\, v_2 : \tau \;\; [\rho]|\!) &= \exists \alpha\beta\gamma.((\!|v_1 : \alpha \xrightarrow{\beta\,[\rho]\,\gamma} \tau|\!) \wedge (\!|v_2 : \alpha|\!) \wedge \pi \leq \beta \wedge \gamma \leq \beta \wedge \gamma \lhd \tau) \\
(\!|\pi \vdash f\, v_1 \cdots v_n : \tau \;\; [\rho]|\!) &= \exists \bar{\alpha}.(D \wedge (\!|v_1 : \tau_1|\!) \wedge \cdots \wedge (\!|v_n : \tau_n|\!) \wedge \pi \leq \pi' \wedge \tau' \leq \tau \wedge \rho' \leq \rho) \\
&\qquad\qquad \text{où } \vdash f : \forall \bar{\alpha}[D].\tau_1 \cdots \tau_n \xrightarrow{\pi'\,[\rho']} \tau' \\
(\!|\pi \vdash \mathsf{let}\, x = v \,\mathsf{in}\, e : \tau \;\; [\rho]|\!) &= \mathsf{let}\, x \,:\, \forall \alpha[(\!|v : \alpha|\!)].\alpha \,\mathsf{in}\, (\!|\pi \vdash e : \tau \;\; [\rho]|\!) \\
(\!|\pi \vdash \mathsf{bind}\, x = e_1 \,\mathsf{in}\, e_2 : \tau \;\; [\rho]|\!) &= \exists \alpha\beta\gamma.\begin{pmatrix}(\!|\pi \vdash e_1 : \alpha \;\; [\beta]|\!) \wedge \mathsf{let}\, x \,:\, \alpha \,\mathsf{in}\, (\!|\gamma \vdash e_2 : \tau \;\; [\rho]|\!) \\ \wedge\, \pi \leq \gamma \wedge \Uparrow\beta \leq \gamma \wedge \beta \leq \rho\end{pmatrix} \\
(\!|\pi \vdash e \,\mathsf{handle}\, \bar{h} : \tau \;\; [\rho]|\!) &= \exists \alpha.((\!|\pi \vdash e : \tau \;\; [\alpha]|\!) \wedge (\!|\pi \vdash \bar{h} : \alpha \Rightarrow \tau \;\; [\rho]|\!) \wedge \alpha \leq_{\mathrm{escape}(\bar{h})} \rho) \\
(\!|\pi \vdash e_1 \,\mathsf{finally}\, e_2 : \tau \;\; [\rho]|\!) &= (\!|\pi \vdash e_1 : \tau \;\; [\rho]|\!) \wedge \exists \alpha.(\!|\pi \vdash e_2 : \alpha \;\; [\partial\bot]|\!)
\end{aligned}
$$

Clauses

$$
\begin{aligned}
(\!|\pi \vdash \bar{\xi}\, x \to e : \rho' \Rightarrow \tau \;\; [\rho]|\!) &= \mathsf{let}\, x \,:\, \mathrm{types}(\bar{\xi}) \,\mathsf{in}\, \exists \alpha.\begin{pmatrix}(\!|\alpha \vdash e_2 : \tau \;\; [\rho]|\!) \wedge \pi \leq \alpha \\ \wedge\, \Uparrow_{\bar{\xi}} \rho' \leq \alpha \wedge \Uparrow_{\bar{\xi}} \rho' \lhd \tau\end{pmatrix} \\
(\!|\pi \vdash \bar{\xi}\, x \to e \,\mathsf{pgt} : \rho' \Rightarrow \tau \;\; [\rho]|\!) &= \mathsf{let}\, x \,:\, \mathrm{types}(\bar{\xi}) \,\mathsf{in}\, \exists \alpha\beta\gamma.\begin{pmatrix}(\!|\alpha \vdash e_2 : \beta \;\; [\gamma]|\!) \wedge \pi \leq \alpha \wedge \gamma \leq \rho \\ \wedge\, \Uparrow_{\bar{\xi}} \rho' \leq \alpha \wedge \rho \leq_{\bar{\xi}} \rho' \wedge \Uparrow_{\bar{\xi}} \gamma \leq \Downarrow_{\bar{\xi}} \rho\end{pmatrix} \\
(\!|\pi \vdash \Xi\_ \to xe : \rho' \Rightarrow \tau \;\; [\rho]|\!) &= \exists \alpha.((\!|\alpha \vdash e_2 : \tau \;\; [\rho]|\!) \wedge \pi \leq \alpha \wedge \Uparrow_{\Xi} \rho' \leq \alpha \wedge \Uparrow_{\Xi} \rho' \lhd \tau) \\
(\!|\pi \vdash \Xi\_ \to x \,\mathsf{pgt}\, e : \rho' \Rightarrow \tau \;\; [\rho]|\!) &= \exists \alpha\beta\gamma.\begin{pmatrix}(\!|\alpha \vdash e_2 : \beta \;\; [\gamma]|\!) \wedge \pi \leq \alpha \wedge \Uparrow_{\Xi} \rho' \leq \alpha \\ \wedge\, \gamma \leq \rho \wedge \rho \leq_{\Xi} \rho' \wedge \Uparrow_{\Xi} \gamma \leq \Downarrow_{\Xi} \rho\end{pmatrix}
\end{aligned}
$$

Figure 7.4 – Algorithme de génération de contraintes

j'annote le symbole $\vdash$ de chaque de jugement de typage d'un $\bullet$ s'il est obtenu à partir de la présentation originale du système, et d'un $\circ$ s'il est dérivé à l'aide de la présentation alternative. Les deux lemmes suivants montrent que cette distinction est en fait inutile.

Lemme 7.6 *Si* $C, \pi, \Gamma \vdash_\circ e : \tau \ [\rho]$ *(resp.* $C, \Gamma \vdash_\circ v : \tau$*) alors* $C, \pi, \Gamma \vdash_\bullet e : \tau \ [\rho]$ *(resp.* $C, \Gamma \vdash_\bullet v : \tau$*).* □

⌜ *Preuve.* Il suffit de vérifier que les règles V'-VARINST et E'-GENLET sont admissibles pour la présentation originale de MLIF($\mathcal{X}$).

∘ *Cas* V'-VARINST. Supposons $\Gamma(x) = \forall \bar{\alpha}[D].\tau$ **(1)**. On a $C \wedge D \Vdash \exists \bar{\alpha}.D$ **(2)**. Par V'-VAR, (1) et (2) impliquent $C \wedge D, \Gamma \vdash_\bullet x : \forall \bar{\alpha}[D].\tau$. Par V'-INST et LOG-DUP, on en déduit le but : $C \wedge D, \Gamma \vdash_\bullet x : \tau$.

∘ *Cas* E'-GENLET. Supposons $C \wedge D, \Gamma \vdash_\bullet v : \tau'$ **(1)**, $\bar{\alpha} \# \mathrm{ftv}(C, \Gamma)$ **(2)** et $C \wedge \exists \bar{\alpha}.D, \pi, (\Gamma; x : \forall \bar{\alpha}[D].\tau') \vdash_\bullet e : \tau \ [\rho]$ **(3)**. Par une instance de V'-GEN de prémisses (1) et (2), on a $C \wedge \exists \bar{\alpha}.D, \Gamma \vdash_\bullet v : \forall \bar{\alpha}[D].\tau'$ **(4)**. Par E'-LET, (4) et (3) donnent le but : $C \wedge \exists \bar{\alpha}.D, \pi, \Gamma \vdash_\bullet e : \tau \ [\rho]$. ⌟

Lemme 7.7 *Si* $C, \pi, \Gamma \vdash_\bullet e : \tau \ [\rho]$ *alors* $C, \pi, \Gamma \vdash_\circ e : \tau \ [\rho]$*. Si* $C, \Gamma \vdash_\bullet v : \forall \bar{\alpha}[D].\tau$ *alors* $C \wedge D, \Gamma \vdash_\circ v : \tau$*.* □

⌜ *Preuve.* On procède par induction sur la dérivation donnée en hypothèse : $C, \pi, \Gamma \vdash_\bullet e : \tau \ [\rho]$ **($\mathbf{H_1}$)** ou $C, \Gamma \vdash_\bullet v : \forall \bar{\alpha}[D].\tau$ **($\mathbf{H_2}$)**. Le résultat est immédiat par induction, sauf dans les cas suivants.

∘ *Cas* V'-VAR. On a $\Gamma(x) = \forall \bar{\alpha}[D].\tau$. Par V'-VARINST, on obtient le jugement $C \wedge D, \Gamma \vdash_\circ v : \tau$, qui est notre but.

∘ *Cas* V'-INST. Le jugement (H_2) est $C, \Gamma \vdash_\bullet v : \tau$, et les prémisses de V'-INST sont $C, \Gamma \vdash_\bullet v : \forall \bar{\alpha}[D].\tau$ **(1)** et $C \Vdash D$ **(2)**. En appliquant l'hypothèse d'induction à (1), on obtient $C \wedge D, \Gamma \vdash_\circ v : \tau$ **(3)**. Par (2) et LOG-DUP, la contrainte $C \wedge D$ est équivalente à C, de telle sorte que (3) est notre but.

∘ *Cas* V'-GEN. Le jugement (H_2) est $C \wedge \exists \bar{\alpha}.D, \Gamma \vdash_\bullet v : \forall \bar{\alpha}[D].\tau$. La première prémisse de V'-GEN est $C \wedge \phi D, \Gamma \vdash v : \phi\tau$ **(1)**, où ϕ est un renommage de $\bar{\alpha}$ frais pour $\mathrm{ftv}(\forall \bar{\alpha}[D].\tau)$ **(2)** tel que $\phi\bar{\alpha} \# \mathrm{ftv}(C, \Gamma)$ **(3)**. Par le lemme 7.2 (page 139) et V'-SUB, on déduit de (1) $C \wedge \phi D \wedge \phi\bar{\alpha} = \bar{\alpha}, \Gamma \vdash v : \tau$. Par V'-HIDE et LOG-EX-AND, grâce à (3), on obtient $C \wedge \exists \phi\bar{\alpha}.(\phi D \wedge \phi\bar{\alpha} = \bar{\alpha}), \Gamma \vdash v : \tau$ **(4)**. Par LOG-NAME-EQ et (2), la contrainte $\exists \phi\bar{\alpha}.(\phi D \wedge \phi\bar{\alpha} = \bar{\alpha})$ est équivalente à D de telle sorte que le jugement (4) est le but recherché.

∘ *Cas* E'-LET. Le jugement (H_2) est $C, \pi, \Gamma \vdash_\bullet \mathsf{let}\, x = v \,\mathsf{in}\, e : \tau \ [\rho]$. Les prémisses de E'-LET sont $C, \Gamma \vdash_\bullet v : \sigma$ **(1)** et $C, \pi, (\Gamma; x : \sigma) \vdash_\bullet e : \tau \ [\rho]$ **(2)**. Posons $\sigma = \forall \bar{\alpha}[D].\tau'$ avec $\bar{\alpha} \# \mathrm{ftv}(C, \Gamma)$ **(3)**. Par l'hypothèse d'induction, (1) et (2) impliquent respectivement $C \wedge D, \Gamma \vdash_\circ v : \tau'$ **(4)** et $C, \pi, (\Gamma; x : \sigma) \vdash_\circ e : \tau \ [\rho]$ **(5)**. Par le lemme 7.3 (page 140) et (1), on a $C \Vdash \exists \bar{\alpha}.D$, de telle sorte que, par LOG-DUP, $C \equiv C \wedge \exists \bar{\alpha}.D$. Ainsi, par une instance de E'-LET de prémisses (4), (3) et (5), on obtient le but : $C, \pi, \Gamma \vdash_\circ \mathsf{let}\, x = v \,\mathsf{in}\, e : \tau \ [\rho]$. ⌟

Ces deux lemmes peuvent être combinés pour former le théorème d'équivalence suivant.

Théorème 7.8 *Le jugement* $C, \pi, \Gamma \vdash e : \tau \ [\rho]$ *(resp.* $C, \Gamma \vdash v : \tau$*) est dérivable avec les règles alternatives si et seulement il est dérivable avec les règles originales* □

⌜ *Preuve.* Par les lemmes 7.6 et 7.7. ⌟

7.2.3 Correction et complétude

Pour terminer ce chapitre, je prouve l'équivalence entre la définition logique de MLIF($\mathcal{X}$) et l'algorithme d'inférence. La correspondance est donnée par deux théorèmes, qui énoncent successivement la *correction* et la *complétude* de l'algorithme d'inférence. Le premier montre que la

contrainte générée est une hypothèse suffisante (au sens de l'implication) pour former une dérivation de typage avec l'environnement et le(s) type(s) donné(s).

Théorème 7.9 (Correction) *Supposons* $\mathrm{fpv}(e) \subseteq \mathrm{dpv}(\Gamma)$. *Alors les jugements* $\pi, \mathsf{let}\,\Gamma\ \mathsf{in}\ (\!|\pi \vdash e : \tau\ [\rho]|\!), \Gamma \vdash e : \tau\ [\rho]$ *et* $\mathsf{let}\,\Gamma\ \mathsf{in}\ (\!|v : \tau|\!), \Gamma \vdash v : \tau$ *sont valides.* □

⌜ *Preuve.* Supposons $\mathrm{fpv}(e) \subseteq \mathrm{dpv}(\Gamma)$ **(H₁)**. On montre par induction sur la structure de l'expression e (resp. de la valeur v) que le jugement $\pi, \mathsf{let}\,\Gamma\ \mathsf{in}\ (\!|\pi \vdash e : \tau\ [\rho]|\!), \Gamma \vdash e : \tau\ [\rho]$ **(C₁)** (resp. $\mathsf{let}\,\Gamma\ \mathsf{in}\ (\!|v : \tau|\!), \Gamma \vdash v : \tau$ **(C₂)**) est dérivable.

∘ *Cas* $v = x$. Par (H_1), on a $x \in \mathrm{dpv}(\Gamma)$. Posons $\Gamma(x) = \forall\bar{\alpha}[D].\tau'$ **(1)** avec $\bar{\alpha} \# \mathrm{ftv}(\tau, \Gamma)$ **(2)**. Puisque $D \Vdash \exists\bar{\alpha}.D$, par une instance de V'-VAR, on a $D \wedge \tau' \leq \tau, \Gamma \vdash x : \forall\bar{\alpha}[D].\tau'$. Par V'-INST et V'-SUB, on en déduit $D \wedge \tau' \leq \tau, \Gamma \vdash x : \tau$, puis par V'-HIDE et (2), $\exists\bar{\alpha}.(D \wedge \tau' \leq \tau), \Gamma \vdash x : \tau$ **(3)**. Par (1) et (2), la contrainte de ce jugement est en fait $\Gamma(x) \preceq \tau$, laquelle est impliquée par $\mathsf{let}\,\Gamma\ \mathsf{in}\ x \preceq \tau$, grâce à LOG-IN-ID, puisque $\mathrm{fpv}(\Gamma(x)) = \varnothing$. Ainsi, par le lemme 7.2 (page 139), on en déduit que le jugement $(\mathrm{C}_2) : \mathsf{let}\,\Gamma\ \mathsf{in}\ x \preceq \tau, \Gamma \vdash x : \tau$ est dérivable.

∘ *Cas* $v = \lambda x.e$. Modulo un renommage de x dans e, on peut supposer $x \# \mathrm{dpv}(\Gamma)$. Soit α, β, γ, δ et ε des variables de type fraîches pour $\mathrm{ftv}(\Gamma, \tau)$ **(1)**. Par l'hypothèse d'induction, on a $\mathsf{let}\,\Gamma; x : \alpha\ \mathsf{in}\ (\!|\beta \vdash e : \gamma\ [\varepsilon]|\!), \beta, (\Gamma; x : \alpha) \vdash e : \gamma\ [\varepsilon]$. Par une instance de V'-ABS, on en déduit le jugement $\mathsf{let}\,\Gamma; x : \alpha\ \mathsf{in}\ (\!|\beta \vdash e : \gamma\ [\varepsilon]|\!), \Gamma \vdash \lambda x.e : \alpha \xrightarrow{\beta\,[\gamma]\,\delta} \varepsilon$. Par le lemme 7.2 (page 139) et V'-SUB, en utilisant LOG-IN-AND*, on obtient $\mathsf{let}\,\Gamma; x : \alpha\ \mathsf{in}\ ((\!|\beta \vdash e : \gamma\ [\varepsilon]|\!) \wedge \alpha \xrightarrow{\beta\,[\gamma]\,\delta} \varepsilon \leq \tau), \Gamma \vdash \lambda x.e : \tau$. Par (1), par V'-HIDE et LOG-IN-EX, on en déduit le but $(\mathrm{C}_2) : \mathsf{let}\,\Gamma\ \mathsf{in}\ \exists\alpha\beta\gamma\delta\varepsilon.(\mathsf{let}\,x : \alpha\ \mathsf{in}\ (\!|\beta \vdash e : \gamma\ [\varepsilon]|\!) \wedge \alpha \xrightarrow{\beta\,[\gamma]\,\delta} \varepsilon \leq \tau), \Gamma \vdash \lambda x.e : \tau$.

∘ *Cas* $e = v_1\, v_2$. Soient α, β et γ des variables de type fraîches pour $\mathrm{ftv}(\tau, \Gamma)$ **(1)**. Par l'hypothèse d'induction, on a $\mathsf{let}\,\Gamma\ \mathsf{in}\ (\!|v_1 : \alpha \xrightarrow{\beta\,[\rho]\,\gamma} \tau|\!), \Gamma \vdash v_1 : \alpha \xrightarrow{\beta\,[\rho]\,\gamma} \tau$ et $\mathsf{let}\,\Gamma\ \mathsf{in}\ (\!|v_2 : \alpha|\!), \Gamma \vdash v_2 : \alpha$. Par le lemme 7.2 (page 139), par une instance de E'-APP, par le lemme 6.4 (page 122), on en déduit le jugement $\mathsf{let}\,\Gamma\ \mathsf{in}\ (\!|v_1 : \alpha \xrightarrow{\beta\,[\rho]\,\gamma} \tau|\!) \wedge \mathsf{let}\,\Gamma\ \mathsf{in}\ (\!|v_2 : \alpha|\!) \wedge \pi \leq \beta \wedge \gamma \leq \beta \wedge \gamma \lhd \tau, \pi, \Gamma \vdash v_1\, v_2 : \tau\ [\rho]$. Par LOG-IN-AND et LOG-IN-AND*, la contrainte de ce jugement est équivalente à $\mathsf{let}\,\Gamma\ \mathsf{in}\ ((\!|v_1 : \alpha \xrightarrow{\beta\,[\rho]\,\gamma} \tau|\!) \wedge (\!|v_2 : \alpha|\!) \wedge \pi \leq \beta \wedge \gamma \leq \beta \wedge \gamma \lhd \tau)$. On en déduit qu'il s'agit du but (C_1).

∘ *Cas* $e = \mathsf{let}\,x = v\ \mathsf{in}\ e'$. Modulo un renommage de x dans e, on peut supposer $x \# \mathrm{dpv}(\Gamma)$ **(1)**. Soit α une variable de type fraîche pour $\mathrm{ftv}(\Gamma)$ **(2)**. Notons $\sigma = \forall\alpha[\mathsf{let}\,\Gamma\ \mathsf{in}\ (\!|v : \alpha|\!)].\alpha$. Par (H_1), on a $\mathrm{fpv}(\sigma) = \varnothing$, ainsi, par hypothèse d'induction, les jugements $\mathsf{let}\,\Gamma\ \mathsf{in}\ (\!|v : \alpha|\!), \Gamma \vdash v : \alpha$ **(3)** et $\mathsf{let}\,\Gamma; x : \sigma\ \mathsf{in}\ (\!|\pi \vdash e : \tau\ [\rho]|\!), \pi, (\Gamma; x : \sigma) \vdash e : \tau\ [\rho]$ **(4)** sont valides. Par V'-GEN, on déduit de (3) et (2) $\exists\sigma, \Gamma \vdash v : \sigma$. Par le lemme 7.2 (page 139), ce jugement peut être affaibli en $\mathsf{let}\,\Gamma; x : \sigma\ \mathsf{in}\ (\!|\pi \vdash e : \tau\ [\rho]|\!), \Gamma \vdash v : \sigma$ **(5)**. Par LOG-LET-DUP, par (H_1) et (2), la contrainte $\mathsf{let}\,\Gamma; x : \sigma\ \mathsf{in}\ (\!|\pi \vdash e : \tau\ [\rho]|\!)$ est équivalente à $\mathsf{let}\,\Gamma\ \mathsf{in}\ \mathsf{let}\,x : \forall\alpha[(\!|v : \alpha|\!)].\alpha\ \mathsf{in}\ (\!|\pi \vdash e : \tau\ [\rho]|\!)$. Ainsi, par une instance de E'-LET de prémisses (5) et (4), on obtient le but $(\mathrm{C}_1) : \mathsf{let}\,\Gamma\ \mathsf{in}\ \mathsf{let}\,x : \forall\alpha[(\!|v : \alpha|\!)].\alpha\ \mathsf{in}\ (\!|\pi \vdash e : \tau\ [\rho]|\!), \pi, \Gamma \vdash \mathsf{let}\,x = v\ \mathsf{in}\ e : \tau\ [\rho]$.

Les autres cas sont analogues aux précédents. ⌟

Le lemme suivant est utilisé dans la preuve du théorème de complétude. Il exprime le fait que $(\!|v : \tau|\!)$ est *covariant* pour τ, et $(\!|\pi \vdash e : \tau\ [\rho]|\!)$ pour τ et ρ. Intuitivement, cela signifie que l'algorithme d'inférence produit suffisamment de contraintes de sous-typage pour prendre en compte toute instance potentielle de V'-SUB et E'-SUB dans les dérivations.

Lemme 7.10 (Covariance) $(\!|\pi \vdash e : \tau\ [\rho]|\!) \wedge \tau \leq \tau' \wedge \rho \leq \rho'$ *implique* $(\!|\pi' \vdash e : \tau'\ [\rho']|\!)$. *De même,* $(\!|v : \tau|\!) \wedge \tau \leq \tau'$ *implique* $(\!|v : \tau'|\!)$. □

Le second théorème montre que la contrainte générée par l'algorithme d'inférence est minimale, c'est-à-dire impliquée par toute hypothèse suffisante pour construire une dérivation de typage. L'hypothèse $C \Vdash \exists\Gamma$ exclut le cas pathologique où l'environnement Γ contient quelque schéma qui n'est pas instanciable sous l'hypothèse C.

Théorème 7.11 (Complétude) *Supposons* $\mathrm{fpv}(C) = \varnothing$ *et* $C \Vdash \exists\Gamma$. *Si* $\pi, C, \Gamma \vdash e : \tau \ [\rho]$ *alors* $C \Vdash \mathsf{let}\,\Gamma\ \mathsf{in}\ (\!|\pi \vdash e : \tau \ [\rho]|\!)$. *Si* $C, \Gamma \vdash v : \tau$ *alors* $C \Vdash \mathsf{let}\,\Gamma\ \mathsf{in}\ (\!|v : \tau|\!)$. □

⌜ *Preuve.* On suppose $\mathrm{fpv}(C) = \varnothing$ et $C \Vdash \exists\Gamma$ **($\mathbf{H_1}$)**. On procède par induction sur la structure de la dérivation donnée en hypothèse, $\pi, C, \Gamma \vdash e : \tau \ [\rho]$ **($\mathbf{H_2}$)** ou $C, \Gamma \vdash v : \tau$ **($\mathbf{H_3}$)**. Grâce au théorème 7.8 (page 144), on peut raisonner sur le jeu de règles alternatif.

∘ *Cas* V'-VARINST. Le jugement (H_3) est $C \wedge D, \Gamma \vdash v : \tau$ et la prémisse de V'-VARINST est $\Gamma(x) = \forall\bar\alpha[D].\tau$. Soit ϕ un renommage de $\bar\alpha$ frais pour $\mathrm{ftv}(\forall\bar\alpha[D].\tau)$ **(1)**. Par LOG-IN-ID et LOG-IN*, on a $\mathsf{let}\,\Gamma\ \mathsf{in}\ (\!|x : \tau|\!) \equiv \exists\Gamma \wedge \exists\phi\bar\alpha.(\phi D \wedge \phi\tau \leq \tau)$ **(2)**. De plus, par (1) et LOG-NAME-EQ, on a $D \Vdash \exists\phi\bar\alpha.(\phi D \wedge \phi\tau \leq \tau)$ **(3)**. Grâce à (H_1), (2) et (3) impliquent le but recherché : $C \wedge D \Vdash \mathsf{let}\,\Gamma\ \mathsf{in}\ (\!|x : \tau|\!)$.

∘ *Cas* V'-ABS. Le jugement (H_3) est $C, \Gamma \vdash \lambda x.e : \tau' \xrightarrow{\pi\ [\rho]\ \lambda} \tau$. La prémisse de V'-ABS est $C, \pi, (\Gamma; x : \tau') \vdash e : \tau \ [\rho]$. Par hypothèse d'induction, on a $C \Vdash \mathsf{let}\,\Gamma\ \mathsf{in}\ \mathsf{let}\,x : \tau'\ \mathsf{in}\ (\!|\pi \vdash e : \tau \ [\rho]|\!)$. Par LOG-NAME-EQ, on en déduit le but recherché : $C \Vdash \mathsf{let}\,\Gamma\ \mathsf{in}\ \exists\alpha\beta\gamma\delta\varepsilon.(\mathsf{let}\,x : \alpha\ \mathsf{in}\ (\!|\beta \vdash e : \gamma \ [\delta]|\!) \wedge \alpha \xrightarrow{\beta\ [\gamma]\ \delta} \varepsilon \leq \tau' \xrightarrow{\pi\ [\rho]\ \lambda} \tau)$.

∘ *Cas* V'-SUB. Le jugement (H_3) est $C, \Gamma \vdash v : \tau$ et les prémisses de V'-SUB sont $C, \Gamma \vdash v : \tau'$ **(1)** et $C \Vdash \tau' \leq \tau$ **(2)**. Par l'hypothèse d'induction (1) donne $C \Vdash \mathsf{let}\,\Gamma\ \mathsf{in}\ (\!|v : \tau'|\!)$. Par le lemme 7.10, on a $(\!|v : \tau'|\!) \wedge \tau' \leq \tau \Vdash (\!|v : \tau|\!)$. Par (2), LOG-IN-AND* et LOG-DUP, on en déduit le but : $C \Vdash \mathsf{let}\,\Gamma\ \mathsf{in}\ (\!|v : \tau|\!)$.

∘ *Cas* V'-HIDE. Le jugement (H_3) est $\exists\bar\alpha.C, \Gamma \vdash v : \tau$. Les prémisses de V'-HIDE sont $C, \Gamma \vdash v : \tau$ **(1)** et $\bar\alpha \# \mathrm{ftv}(C, \Gamma, \tau)$ **(2)**. Par l'hypothèse d'induction, (1) implique $C \Vdash \mathsf{let}\,\Gamma\ \mathsf{in}\ (\!|v : \tau|\!)$ **(3)**. Par (2), on a $\bar\alpha \# \mathrm{ftv}(\mathsf{let}\,\Gamma\ \mathsf{in}\ (\!|v : \tau|\!))$, donc (3) implique le but : $\exists\bar\alpha.C \Vdash \mathsf{let}\,\Gamma\ \mathsf{in}\ (\!|v : \tau|\!)$.

∘ *Cas* E'-APP. Le jugement (H_2) est $C, \pi, \Gamma \vdash v_1\, v_2 : \tau \ [\rho]$. Les prémisses de E'-APP sont $C, \Gamma \vdash v_1 : \tau' \xrightarrow{\pi'\ [\rho]\ \lambda} \tau$ **(1)**, $C, \Gamma \vdash v_2 : \tau'$ **(2)**, $C \Vdash \pi \leq \pi'$ **(3)**, $C \Vdash \lambda \leq \pi'$ **(4)** et $C \Vdash \lambda \lhd \tau$ **(5)**. Soit α, β et γ des variables de type fraîches pour $\mathrm{ftv}(\Gamma, C, \pi, \tau, \rho)$ **(6)**. Par le lemme 7.2 (page 139) et V'-SUB, (1) implique $C \wedge \alpha = \tau' \wedge \beta = \pi' \wedge \gamma = \lambda, \Gamma \vdash v_1 : \alpha \xrightarrow{\beta\ [\rho]\ \gamma} \tau$ et (2) implique $C \wedge \alpha = \tau' \wedge \beta = \pi' \wedge \gamma = \lambda, \Gamma \vdash v_2 : \tau'$. En appliquant l'hypothèse d'induction à chacun de ces jugements, on en déduit $C \wedge \alpha = \tau' \wedge \beta = \pi' \wedge \gamma = \lambda \Vdash \mathsf{let}\,\Gamma\ \mathsf{in}\ (\!|v_1 : \alpha \xrightarrow{\beta\ [\rho]\ \gamma} \tau|\!)$ et $C \wedge \alpha = \tau' \wedge \beta = \pi' \wedge \gamma = \lambda \Vdash \mathsf{let}\,\Gamma\ \mathsf{in}\ (\!|v_2 : \alpha|\!)$. Par LOG-IN-AND, on en déduit $C \wedge \alpha = \tau' \wedge \beta = \pi' \wedge \gamma = \lambda \Vdash \mathsf{let}\,\Gamma\ \mathsf{in}\ ((\!|v_1 : \alpha \xrightarrow{\beta\ [\rho]\ \gamma} \tau|\!) \wedge (\!|v_2 : \alpha|\!))$, puis, en utilisant (3), (4) et (5), $C \wedge \alpha = \tau' \wedge \beta = \pi' \wedge \gamma = \lambda \Vdash \mathsf{let}\,\Gamma\ \mathsf{in}\ ((\!|v_1 : \alpha \xrightarrow{\beta\ [\rho]\ \gamma} \tau|\!) \wedge (\!|v_2 : \alpha|\!) \wedge \pi \leq \beta \wedge \gamma \leq \beta \wedge \beta \lhd \tau)$ **(7)**. Par (6), α, β et γ ne sont pas libre dans C ou Γ ; ainsi on déduit de (7), grâce à LOG-EX* et LOG-IN-EX, $C \wedge \exists\alpha\beta\gamma.(\alpha = \tau' \wedge \beta = \pi' \wedge \gamma = \lambda) \Vdash \mathsf{let}\,\Gamma\ \mathsf{in}\ \exists\alpha\beta\gamma.((\!|v_1 : \alpha \xrightarrow{\beta\ [\rho]\ \gamma} \tau|\!) \wedge (\!|v_2 : \alpha|\!) \wedge \pi \leq \beta \wedge \gamma \leq \beta \wedge \beta \lhd \tau)$. Par (6) et LOG-NAME, le membre gauche de cette implication est équivalent à C. On en déduit notre but : $C \Vdash \mathsf{let}\,\Gamma\ \mathsf{in}\ \exists\alpha\beta\gamma.((\!|v_1 : \alpha \xrightarrow{\beta\ [\rho]\ \gamma} \tau|\!) \wedge (\!|v_2 : \alpha|\!) \wedge \pi \leq \beta \wedge \gamma \leq \beta \wedge \beta \lhd \tau)$.

∘ *Cas* E'-GENLET. Le jugement (H_2) est $C \wedge \exists\bar\alpha.D, \pi, \Gamma \vdash \mathsf{let}\,x = v\ \mathsf{in}\ e : \tau \ [\rho]$. Les prémisses de E'-GENLET sont $C \wedge D, \Gamma \vdash v : \tau'$ **(1)**, $\bar\alpha \# \mathrm{ftv}(C, \Gamma)$ **(2)**, $C \wedge \exists\bar\alpha.D, \pi, (\Gamma; x : \forall\bar\alpha[D].\tau') \vdash e : \tau \ [\rho]$ **(3)**. Par l'hypothèse d'induction, (1) et (3) donnent respectivement $C \wedge D \Vdash \mathsf{let}\,\Gamma\ \mathsf{in}\ (\!|v : \tau'|\!)$ **(4)** et $C \wedge \exists\bar\alpha.D \Vdash \mathsf{let}\,\Gamma\ \mathsf{in}\ \mathsf{let}\,x : \forall\bar\alpha[D].\tau'\ \mathsf{in}\ (\!|\pi \vdash e : \tau \ [\rho]|\!)$ **(5)**. Soit α une variable fraîche pour $\bar\alpha \cup \mathrm{ftv}(\tau')$ **(6)**. On a :

$$
\begin{array}{ll}
C \wedge \mathsf{let}\,\Gamma\ \mathsf{in}\ \mathsf{let}\,x : \forall\bar\alpha[D].\tau'\ \mathsf{in}\ (\!|\pi \vdash e : \tau \ [\rho]|\!) & \\
\quad \Vdash \mathsf{let}\,\Gamma\ \mathsf{in}\ \mathsf{let}\,x : \forall\bar\alpha[C \wedge D].\tau'\ \mathsf{in}\ (\!|\pi \vdash e : \tau \ [\rho]|\!) & \textbf{(7)} \\
\quad \Vdash \mathsf{let}\,\Gamma\ \mathsf{in}\ \mathsf{let}\,x : \forall\bar\alpha[\mathsf{let}\,\Gamma\ \mathsf{in}\ (\!|v : \tau'|\!)].\tau'\ \mathsf{in}\ (\!|\pi \vdash e : \tau \ [\rho]|\!) & \textbf{(8)} \\
\quad \Vdash \mathsf{let}\,\Gamma\ \mathsf{in}\ \mathsf{let}\,x : \forall\bar\alpha[(\!|v : \tau'|\!)].\tau'\ \mathsf{in}\ (\!|\pi \vdash e : \tau \ [\rho]|\!) & \textbf{(9)} \\
\quad \Vdash \mathsf{let}\,\Gamma\ \mathsf{in}\ \mathsf{let}\,x : \forall\bar\alpha\alpha[(\!|v : \alpha|\!) \wedge \alpha = \tau].\alpha\ \mathsf{in}\ (\!|\pi \vdash e : \tau \ [\rho]|\!) & \textbf{(10)} \\
\quad \Vdash \mathsf{let}\,\Gamma\ \mathsf{in}\ \mathsf{let}\,x : \forall\alpha[(\!|v : \alpha|\!)].\alpha\ \mathsf{in}\ (\!|\pi \vdash e : \tau \ [\rho]|\!) & \textbf{(11)}
\end{array}
$$

où (7) est obtenu par (2), LOG-DUP et LOG-LET-AND, (8) est obtenu par (4), (9) est obtenu par LOG-LET-DUP, (10) est obtenu par LOG-NAME, LOG-EX-AND, LOG-LET-EX et LOG-LET-EQ, grâce à (6),

(11) est obtenu par LOG-LET-EX, LOG-EX-AND et LOG-NAME, grâce à (6). En combinant (5) et (11), on obtient le but : $C \wedge \exists \bar{\alpha}.D \Vdash \mathsf{let}\,\Gamma\ \mathsf{in}\ \mathsf{let}\,x\ :\ \forall\alpha[(\!|v : \alpha|\!)].\alpha\ \mathsf{in}\ (\!|\pi \vdash e : \tau\ \ [\rho]|\!)$. ⌟

CHAPITRE 8 HUIT

Extensions

Dans les chapitres précédents, j'ai donné plusieurs exemples simples de constructeurs et de primitives dont le langage Core ML peut être muni, puis montré comment ceux-ci peuvent être typés dans les systèmes MLIF($\mathcal{T}$) et MLIF($\mathcal{X}$), de manière à analyser les flots d'information qu'ils engendrent. Dans ce chapitre, je m'intéresse à deux extensions supplémentaires du langage : les *types de données algébriques* et les *primitives génériques*. Bien que leur traitement soit légèrement plus sophistiqué, ces deux extensions entrent dans le cadre formel étudié jusqu'à présent : elles consistent en effet en l'adjonction de nouvelles constantes dans le langage.

Avant de poursuivre, je souhaite résumer en quelques lignes le mécanisme général pour formuler chaque extension du système et prouver sa correction. Tout d'abord, après avoir décrit les constantes et leurs arités, je donne la sémantique des primitives dans le langage Core ML, qui doit être stable par renommage des adresses mémoire (hypothèse 5.1, page 100) et déterministe (hypothèse 5.2, page 101). Il faut ensuite étendre cette sémantique à Core ML^2, d'une manière qui respecte le résultat de simulation (hypothèses 5.6 et 5.7, page 107). L'étape suivante consiste à donner les règles de typage des constantes dans MLIF($\mathcal{T}$) et MLIF($\mathcal{X}$), lesquelles doivent être reliées comme indiqué par l'hypothèse 7.1 (page 137). Notons que cette hypothèse fournit en fait une procédure qui permet, si besoin est, de dériver pour chaque constante des règles de typage MLIF($\mathcal{T}$) à partir des règles MLIF($\mathcal{X}$) de manière systématique. Enfin, le résultat de non-interférence pour le langage étendu est obtenu en montrant que le typage est préservé par la réduction des primitives dans Core ML^2 (hypothèses 6.14 et 6.15, page 126).

8.1 Types de données algébriques

La popularité du langage ML tient, pour une large part, à la facilité offerte par les types de données algébriques (*algebraic data-types* en anglais) pour définir des structures de données complexes et les manipuler. Ils surpassent les paires et les sommes binaires étudiées dans les chapitres précédents — ainsi que leur généralisation immédiate en n-uplets et sommes n-aires — tout en palliant deux importantes limitations. Tout d'abord, l'accès aux composantes d'un type de donnée peut s'effectuer par des *noms* (au lieu d'index numériques). D'autre part, ils permettent de définir des structures de données dont la définition est récursive, comme des listes et des arbres, sans toutefois nécessiter l'introduction de types récursifs.

L'analyse de flots d'information typée décrite dans cette thèse se prête bien à l'étude de ces structures de données, car leur forme est, déjà en ML, précisément décrite par les types. Cette extension ne présente pas de difficulté technique notable par rapport au traitement des paires et des sommes binaires. Il est toutefois intéressant d'expliquer précisément et de manière systématique comment les types de données et leurs déclarations doivent être annotés.

8.1.1 Déclarations

Pour simplifier quelque peu les choses, je considère que les types de données sont définis de manière *globale*, dans un *prélude* constitué d'un ensemble de déclarations possiblement mutuellement récursives. Cette vision des choses est bien entendue contradictoire avec l'écriture modulaire des programmes, mais il s'agit essentiellement d'un problème orthogonal à celui qui m'intéresse ici.

Je suppose distingué un ensemble de constructeurs de types c appelés ***types de données***. Je présume également l'existence d'un ensemble de noms l utilisés indifféremment comme ***variants*** ou ***champs***. Une ***déclaration de type de données*** est soit une déclaration de ***type enregistrement***, soit une déclaration de ***type variante***, dont les formes respectives sont :

$$c\,\vec{\alpha} \triangleq \prod_{j\in[1,n]} l_j : \tau_j \qquad \text{et} \qquad c\,\vec{\alpha} \triangleq \sum_{j\in[1,n]}^{\lambda} l_j : \tau_j$$

La forme de ces déclarations est à rapprocher de celles des types produits et sommes introduits au chapitre 6 (page 113). Les déclarations de types enregistrements ont exactement la même forme qu'en ML : elles consistent simplement en la liste des types associés à chaque champ. Comme pour les paires, aucune annotation supplémentaire n'est requise, puisque toute l'information portée par un enregistrement est en fait contenue dans ses champs. À l'inverse, les déclarations de types variantes portent une annotation λ. Elle est utilisée pour décrire la quotité d'information associée, dans chaque valeur du type, au constructeur de données — comme dans le cas des sommes binaires. Le choix d'annoter les types variantes par *exactement* un niveau est quelque peu arbitraire, quoique naturel et très satisfaisant en pratique. Notons toutefois que cette annotation est superflue dans le cas d'un type variante à un constructeur ($n = 1$). Dans un autre travail [Sim02b], présenté brièvement section 9.3 (page 163), j'ai étudié la possibilité de raffiner les annotations des types sommes en utilisant un niveau pour *chaque couple* de variants.

Dans les deux formes de déclaration ci-dessus, l'entier n doit être strictement positif. Si c a la signature $\vec{\kappa}$ alors les variables de type $\vec{\alpha}$ doivent avoir les sortes $\vec{\kappa}$. De plus, chaque type τ_j doit avoir la sorte Type, et l'annotation λ des variantes la sorte Atom. Les variables de type $\vec{\alpha}$, appelées paramètres du type de données, sont liées dans les types $\tau_1, \ldots, \tau_n$ des champs ou des variants, ainsi que dans l'annotation λ pour les types variantes. La déclaration doit être close, *i.e.* $\mathrm{ftv}(\tau_1, \ldots, \tau_n) \subseteq \bar{\alpha}$ pour les enregistrements et $\mathrm{ftv}(\tau_1, \ldots, \tau_n, \lambda) \subseteq \bar{\alpha}$ pour les variantes. Enfin, je me restreins aux déclarations de types qui ne mentionnent pas de constante atomique (*i.e.* d'élément de $\mathcal{L}$) : ce choix est peu restrictif dans la pratique, puisque chaque occurrence d'une constante dans le corps d'une déclaration peut être remplacée par un paramètre supplémentaire de sorte Atom. En l'absence de cette hypothèse, il serait nécessaire d'introduire une nouvelle définition pour la garde ($\lhd$), plus sophistiquée que celle considérée jusqu'à présent.

Pour qu'une déclaration de type de données soit valide, il est nécessaire que le comportement du constructeur de type c vis-à-vis du sous-typage ($\leq$) et de la garde ($\lhd$) corresponde au corps de la définition, c'est-à-dire au type produit ou somme auquel le type de données est déclaré isomorphe. Ce critère est formalisé par les deux définitions suivantes.

Définition 8.1 (Type enregistrement valide) *Soient* $c\,\vec{\alpha} \triangleq \prod_{j\in[1,n]} l_j : \tau_j$ *et* $c\,\vec{\alpha}' \triangleq \prod_{j\in[1,n]} l_j : \tau'_j$ *deux* α*-variantes d'une déclaration de type. Cette déclaration est valide si et seulement si* ***(i)*** $c\,\vec{\alpha} \leq c\,\vec{\alpha}' \Vdash \tau_1 \leq \tau'_1 \wedge \cdots \wedge \tau_n \leq \tau'_n$ *et* ***(ii)*** $\lambda \lhd c\,\vec{\alpha} \Vdash \lambda \lhd \tau_1 \wedge \cdots \wedge \lambda \lhd \tau_n$. □

Le point (i) est relatif au sous-typage, plus précisément aux variances du constructeur c. L'idée est que, puisque le type $c\,\vec{\alpha}$ est déclaré isomorphe au produit $\tau_1 \times \cdots \times \tau_n$, à chaque fois que deux types construits avec c sont comparables, il doit en être de même de leurs expansions. L'implication (ii) porte sur la définition de la garde sur le constructeur c, c'est-à-dire l'ensemble guarded(c). De même que pour le sous-typage, si un type construit avec c est gardé par un niveau λ, alors son expansion doit également être gardée par λ.

Voici la contrepartie de cette définition pour les types variantes.

Définition 8.2 (Type variante valide) *Soient* $c\,\vec{\alpha} \triangleq \sum^{\lambda}{}_{j\in[1,n]} l_j : \tau_j$ *et* $c\,\vec{\alpha}' \triangleq \sum^{\lambda'}{}_{j\in[1,n]} l_j : \tau'_j$ *deux* α*-variantes d'une déclaration de type. Cette déclaration est valide si et seulement si* ***(i)*** $c\,\vec{\alpha} \leq c\,\vec{\alpha}' \Vdash \tau_1 \leq \tau'_1 \wedge \cdots \wedge \tau_n \leq \tau'_n \wedge \lambda \leq \lambda'$ *et* ***(ii)*** $\lambda'' \lhd c\,\vec{\alpha} \Vdash \lambda'' \leq \lambda$. □

Le point (i) porte à nouveau sur la définition du sous-typage. En plus des types associés aux variants, elle fait également intervenir l'annotation λ du type somme, qui est traitée de manière covariante. Le point (ii) exprime le rôle de cette annotation, qui est utilisée pour la garde.

On peut donner une définition équivalente de ces critères, d'une manière plus syntaxique, en examinant les occurrences des paramètres d'une déclaration dans le corps de cette dernière. J'ai introduit, au chapitre 1 (page 19), les ensembles $\mathrm{ftv}^+(\tau)$ et $\mathrm{ftv}^-(\tau)$ des variables de type apparaissant en positions respectivement positives et négatives dans un type τ. De même, je définis l'ensemble $\mathrm{ftv}^{\lhd}(\tau)$ des variables de type apparaissant en position gardée dans τ par les égalités suivantes :

$$\begin{aligned}
\mathrm{ftv}^{\lhd}(\alpha) &= \{\alpha\} \\
\mathrm{ftv}^{\lhd}(c\,\tau_1 \cdots \tau_n) &= \bigcup\{\, \mathrm{ftv}^{\lhd}(\tau_i) \mid i \in \mathrm{guarded}(c)\,\} \\
\mathrm{ftv}^{\lhd}(\xi : \tau_1; \tau_2) &= \mathrm{ftv}^{\lhd}(\tau_1) \cup \mathrm{ftv}^{\lhd}(\tau_2) \\
\mathrm{ftv}^{\lhd}(\partial\tau) &= \mathrm{ftv}^{\lhd}(\tau)
\end{aligned}$$

Propriété 8.3 (Type enregistrement valide) *La déclaration* $c\,\alpha_1 \cdots \alpha_m \triangleq \prod_{j\in[1,n]} l_j : \tau_j$ *est valide si et seulement si : pour tous* $i \in [1,m]$ *et* $j \in [1,n]$, ***(i)*** *si* $\alpha_i \in \mathrm{ftv}^+(\tau_j)$ *alors* $+ \in c.i$, ***(ii)*** *si* $\alpha_i \in \mathrm{ftv}^-(\tau_j)$ *alors* $- \in c.i$, ***(iii)*** *si* $\alpha_i \in \mathrm{ftv}^{\lhd}(\tau_j)$ *alors* $i \in \mathrm{guarded}(c)$. □

Les points (i) et (ii) assurent que la variance de chaque occurrence d'un paramètre dans le corps de la définition est pris en compte par le constructeur de type c. Le point (iii) agit de manière analogue sur la garde.

Propriété 8.4 (Type variante valide) *La déclaration* $c\,\alpha_1 \cdots \alpha_m \triangleq \sum^{\lambda}{}_{j\in[1,n]} l_j : \tau_j$ *est valide si et seulement si : pour tous* $i \in [1,m]$ *et* $j \in [1,n]$, ***(i)*** *si* $\alpha_i \in \mathrm{ftv}^+(\tau_j)$ *alors* $+ \in c.i$, ***(ii)*** *si* $\alpha_i \in \mathrm{ftv}^-(\alpha_i)$ *alors* $- \in c.i$ *; et* ***(iii)*** λ *est* α_j *pour* $j \in [1,n]$ *tel que* $j \in \mathrm{guarded}(c)$. □

Dans une implémentation de ce système, comme Flow Caml [Sim], on souhaite *a priori* déterminer les variances et paramètres gardés de chaque constructeur de type de données de manière automatique, à partir d'un ensemble de déclarations mutuellement récursives. En utilisant les propriétés 8.3 et 8.4, il est facile de générer, à partir d'un ensemble de déclarations de types de données, une contrainte booléenne pour résoudre ce problème. Cette contrainte fait intervenir trois variables booléennes pour chaque paramètre de chaque constructeur de type, qui indiquent respectivement si le paramètre est covariant, contravariant et gardé. On peut vérifier que, pour tout prélude, ces contraintes admettent systématiquement une solution *minimale*, à condition d'admettre la variance vide (ni covariante, ni contravariante). Cependant, les déclarations nécessitant cette variance supplémentaire peuvent raisonnablement être rejetées : il est facile de vérifier qu'un paramètre ni covariant ni invariant peut être éliminé d'un ensemble de déclarations.

Pour illustrer ce formalisme, je donne quelques exemples simples de déclaration de type. Tout d'abord, les Booléens peuvent naturellement être définis comme un type de données :

$$\mathsf{bool}\,\alpha \triangleq \sum^{\beta}(\mathsf{true} : \mathsf{unit}; \mathsf{false} : \mathsf{unit})$$

Puisque je n'ai considéré que des variants unaires, il me faut équiper les constructeurs true et false d'un argument factice, de type unit. Il est cependant immédiat de généraliser les déclarations de types variantes de manière à autoriser des constructeurs d'arité arbitraire. Je définis maintenant le type variante des listes, list, puis un type enregistrement, tree, permettant de représenter des arbres.

$$\mathsf{list}\,\alpha\,\beta \triangleq \textstyle\sum^{\beta}(\mathsf{Nil} : \mathsf{unit}; \mathsf{Cons} : \alpha \times \mathsf{list}\,\alpha\,\beta)$$
$$\mathsf{tree}\,\alpha\,\beta \triangleq \textstyle\prod\ (\mathsf{root} : \alpha; \mathsf{sons} : \mathsf{list}(\mathsf{tree}\,\alpha\,\beta)\,\beta)$$

Le constructeur de type list a deux paramètres : α, de sorte Type, est le type des éléments de la liste et β, de sorte Atom, est le niveau associé au type somme sous-jacent. Il décrit la quotité d'information portée par la *structure* de la liste. La première occurrence de α dans le corps de la déclaration de list est en position covariante, et la première occurrence de β, comme annotation du type somme, est en position covariante et gardée. Il est aisé de vérifier qu'il s'agit des seules contraintes portant sur les paramètres de la déclaration ; ainsi, le constructeur list peut être muni de la signature (minimale) suivante :

$$\mathsf{Type}^{+} \cdot \mathsf{Atom}^{+\lhd}$$

Comme list, le constructeur de type tree a également deux paramètres : α est le type des éléments de l'arbre, et β est un niveau de sécurité associé à la structure de l'arbre. La première occurrence de α et la dernière occurrence de β dans le corps de la déclaration sont en positions covariantes et gardées. Là encore, on peut vérifier qu'il s'agit des seules contraintes portant sur les paramètres de la déclaration, de telle sorte que le constructeur tree peut être muni de la signature suivante :

$$\mathsf{Type}^{+\lhd} \cdot \mathsf{Atom}^{+\lhd}$$

Considérons enfin un type de données permettant de représenter des listes dont les éléments peuvent être modifiés en place :

$$\mathsf{mlist}\,\alpha\,\beta \triangleq \textstyle\sum^{\beta}(\mathsf{MNil} : \mathsf{unit}; \mathsf{MCons} : \mathsf{ref}\,\alpha\,\beta \times \mathsf{mlist}\,\alpha\,\beta)$$

Dans cette déclaration, le niveau β représente à la fois l'information attachée à la structure de la liste et à l'identité des références. (On pourrait également distinguer ces deux quotités en annotant le type mlist par deux niveaux au lieu d'un seul.) L'occurrence de α comme paramètre du constructeur ref est en position invariante dans le corps de la déclaration, tandis que toutes les occurrences de β sont en position covariante. On en déduit que le constructeur de type mlist peut recevoir la signature suivante :

$$\mathsf{Type}^{\pm} \cdot \mathsf{Atom}^{+\lhd}$$

8.1.2 Sémantique et typage

Un ***prélude*** est un ensemble de déclarations de types de données, où chaque type de données est déclaré au plus une fois, et où chaque nom de variant ou champ apparaît au plus une fois. L'effet d'un prélude est d'étendre le langage de programmation avec de nouvelles constantes, comme expliqué ci-après.

Supposons tout d'abord que le prélude contient la déclaration de type enregistrement

$$c\,\vec{\alpha} \triangleq \prod_{j\in[1,n]} l_j : \tau_j.$$

Alors, un constructeur d'arité n nommé record_c est introduit, ainsi, que pour chaque $j \in [1, n]$, un destructeur nommé l_j d'arité 1. On écrit $\{\, l_j : v_j \,\}_{j=1}^{n}$ pour la valeur enregistrement $\mathsf{record}_c\, v_1 \cdots v_n$, et $v.l_j$ pour l'application $l_j\, v$. La sémantique opérationnelle de Core ML est étendue par les règles de réduction suivantes, pour $j \in \{1, \ldots, n\}$:

$$\{\, l_{j'} : v_{j'} \,\}_{j'=1}^{n}.l_j \,/\, \mu \longrightarrow_{l_j} v_j + \varnothing \qquad \text{(record)}$$

Axiomes

$$\frac{c\,\vec{\alpha} \triangleq \prod_{j\in[1,n]} l_j : \tau_j}{\vdash \mathsf{record}_c : \forall\bar{\alpha}[\mathsf{true}].\tau_1 \cdots \tau_n \to c\,\vec{\alpha}} \qquad \frac{c\,\vec{\alpha} \triangleq \prod_{j\in[1,n]} l_j : \tau_j}{\vdash l_j : \forall\bar{\alpha}\beta\gamma[\mathsf{true}].c\,\vec{\alpha} \xrightarrow{\beta\,[\gamma]} \tau_j}$$

Règles dérivées

$$\frac{\forall j\ \ C,\Gamma \vdash v_j : \tau_j \qquad c\,\vec{\alpha} \triangleq \prod_{j\in[1,n]} l_j : \tau_j}{C,\Gamma \vdash \{\, l_j : v_j \,\}_{j=1}^{n} : c\,\vec{\alpha}} \qquad \frac{C,\Gamma \vdash v : c\,\vec{\alpha} \qquad c\,\vec{\alpha} \triangleq \prod_{j\in[1,n]} l_j : \tau_j}{C,\pi,\Gamma \vdash v.l_j : \tau_j\ \ [\rho]}$$

Figure 8.1 – Règles de typage des enregistrements

Axiomes

$$\frac{c\,\vec{\alpha} \triangleq \sum^{\lambda}_{j\in[1,n]} l_j : \tau_j}{\vdash l_j : \forall\bar{\alpha}[\mathsf{true}].\tau_j \to c\,\vec{\alpha}}$$

$$\frac{c\,\vec{\alpha} \triangleq \sum^{\lambda}_{j\in[1,n]} l_j : \tau_j}{\vdash \mathsf{case}_c : \forall\bar{\alpha}\beta\gamma\delta\varepsilon[\beta \le \delta \wedge \lambda \le \delta \wedge \lambda \lhd \varepsilon].c\,\vec{\alpha} \cdot (\tau_1 \xrightarrow{\delta\,[\gamma]\,\lambda} \varepsilon) \cdots (\tau_n \xrightarrow{\delta\,[\gamma]\,\lambda} \varepsilon) \xrightarrow{\beta\,[\gamma]} \tau_j}$$

Règles dérivées

$$\frac{C,\Gamma \vdash v : \tau_j \qquad c\,\vec{\alpha} \triangleq \sum^{\lambda}_{j\in[1,n]} l_j : \tau_j}{C,\Gamma \vdash l_j\, v : c\,\vec{\alpha}}$$

$$\frac{\begin{array}{c} C,\Gamma \vdash v : c\,\vec{\alpha} \qquad \forall j\ \ C,\Gamma \vdash v : \tau_j \xrightarrow{\pi'\,[\rho]\,\lambda} \tau \qquad c\,\vec{\alpha} \triangleq \sum^{\lambda}_{j\in[1,n]} l_j : \tau_j \\ C \Vdash \pi \le \pi' \qquad C \Vdash \lambda \le \pi' \qquad C \Vdash \lambda \lhd \tau \end{array}}{C,\pi,\Gamma \vdash \mathsf{case}\, v\, [l_j : v_j]_{j=1}^{n} : \tau\ \ [\rho]}$$

Figure 8.2 – Règles de typage des variantes

Les constructeurs d'enregistrements et les primitives d'accès aux champs sont typées dans le système MLIF($\mathcal{X}$) par les règles de la figure 8.1, qui sont analogues à celles utilisées pour les paires.

Supposons maintenant que le prélude contient la définition

$$c\,\vec{\alpha} \triangleq \sum_{j\in[1,n]}^{\lambda} l_j : \tau_j.$$

Alors, pour chaque $j \in [1,n]$, un constructeur d'arité 1 nommé l_j est introduit, ainsi qu'un destructeur d'arité $n+1$ nommé case_c. L'application $\mathsf{case}_c\, v\, v_1 \cdots v_n$ est notée $\mathsf{case}\, v\, [l_j : v_j]_{j=1}^{n}$. La sémantique de Core ML est étendue par les règles de réduction suivantes, pour $j \in \{1,\ldots,n\}$:

$$\mathsf{case}\, l_j\, v\, [l_{j'} : v_{j'}]_{j'=1}^{n} \,/\, \mu \longrightarrow_{\mathsf{case}_c} v_j\, v + \varnothing \qquad \text{(variant)}$$

Les règles de typage relatives aux variants et à la construction case sont données figure 8.2. Elles sont analogues à celles utilisées pour les sommes binaires.

J'omets la preuve de correction du typage des enregistrements et des variants : cette question se traite en effet exactement de la même manière que pour les paires et les sommes binaires. Les règles de réduction Core ML^2 appropriées pour les types de données sont syntaxiquement identiques à celles de Core ML. Il faudrait ensuite traduire les règles de typage données figures 8.1 et 8.2 dans le système MLIF($\mathcal{T}$), comme indiqué par l'hypothèse 7.1 (page 137), puis montrer que celles-ci satisfont les hypothèses 6.14 et 6.15.

8.2 Primitives génériques

La plupart des langages de la famille ML, comme Caml [Ler, LDG+b] ou SML [sml] offrent des primitives génériques de comparaison, de hachage ou de sérialisation (*marshalling* en anglais). Je présente maintenant une méthode qui permet de typer de telles opérations, sans connaissance particulière de leur sémantique, c'est-à-dire en les considérant comme des « boîtes noires » qui utilisent potentiellement *toute* l'information contenue dans leurs arguments. La pertinence de ces primitives peut être discutée, notamment parce qu'elles permettent de briser certaines propriétés normalement garanties par les mécanismes d'*abstraction de type* offerts par les systèmes de modules à la ML. Cependant, puisqu'elles sont *de facto* largement utilisées, il me semble utile de m'y intéresser.

Dans toute la suite de cette section, je considère une primitive f fixée d'arité n. Mon but est de montrer comment, à partir de sa sémantique et de son type non annoté dans Core ML, on peut obtenir une règle de typage sûre pour l'analyse de flots d'information.

8.2.1 Sémantique

L'***accessibilité*** relativement à un état-mémoire (Core ML) μ, est la plus petite relation réflexive entre valeurs (Core ML) telle que **(*i*)** pour toute valeur v, chaque sous-terme de v est accessible depuis v et **(*ii*)** pour toute adresse mémoire m, $\mu(m)$ est accessible depuis m. De manière abrégée, on dit que v' est accessible depuis $\bar{v}$ relativement à μ si et seulement si il existe $v \in \bar{v}$ tel que v' est accessible depuis v relativement à μ. Enfin, étant donné un état mémoire μ et des valeurs $\bar{v}$, je définis l'état mémoire $|\mu|_{\bar{v}}$ par :

$$|\mu|_{\bar{v}}(m) = \begin{cases} \mu(m) & \text{si } m \text{ est accessible depuis } \bar{v} \text{ relativement à } \mu \\ \mathsf{null} & \text{sinon} \end{cases}$$

Ainsi, $|\mu|_{\bar{v}}$ représente la partie de μ accessible depuis les valeurs $\bar{v}$.

La sémantique de la primitive f dans Core ML doit être définie par une relation $-f\!\!\rightarrow$, comme expliqué à la section 5.1.2 (page 99). Celle-ci doit en particulier être stable par renommage des adresses mémoire (hypothèse 5.1, page 100) et déterministe (hypothèse 5.2, page 101). Je formule deux hypothèses supplémentaires sur cette sémantique, de manière à pouvoir étudier les flots d'information qu'elle génère.

Hypothèse 8.5 *Si* $f\,\vec{v} \,/\, \mu \; -f\!\!\rightarrow\; a + \dot{\mu}$ *alors* $\dot{\mu} = \varnothing$. □

Hypothèse 8.6 *Si* $f\,\vec{v} \,/\, \mu \; -f\!\!\rightarrow\; a + \varnothing$ *et* $|\mu|_{\vec{v}} = |\mu'|_{\vec{v}}$ *alors* $f\,\vec{v} \,/\, \mu' \; -f\!\!\rightarrow\; a + \varnothing$. □

Par l'hypothèse 8.5, la primitive f ne peut modifier l'état-mémoire. Notons qu'elle peut cependant lever une exception, car aucune restriction n'est posée sur la forme du résultat a. L'hypothèse 8.6 assure quant à elle que la sémantique d'une primitive ne dépend que de la partie de l'état mémoire accessible depuis les arguments : cela empêche la primitive d'avoir un état interne, en conservant par exemple un pointeur sur une adresse mémoire privée. Cette hypothèse est nécessaire pour la sûreté de la règle de typage présentée ci-après.

La sémantique de la primitive f pour Core ML est étendue au langage Core ML^2 de manière systématique par la règle suivante :

$$\frac{f\,\lfloor\vec{v}\rfloor_1 \,/\, \lfloor\mu\rfloor_1 \; -f\!\!\rightarrow\; a + \varnothing \qquad f\,\lfloor\vec{v}\rfloor_2 \,/\, \lfloor\mu\rfloor_2 \; -f\!\!\rightarrow\; a + \varnothing}{f\,\vec{v} \,/\, \mu \; -f\!\!\twoheadrightarrow\; a + \varnothing} \qquad \text{(primitive)}$$

En d'autres termes, la configuration Core ML^2 $f\,\vec{v} \,/\, \mu$ peut être réduite par la règle (δ) si chacune de ses projections prise séparément produit le *même* résultat. Grâce à l'hypothèse 8.6, ce cas intervient en particulier lorsque les projections des arguments, $\lfloor\vec{v}\rfloor_1$ et $\lfloor\vec{v}\rfloor_2$, ainsi que celles des parties accessibles de l'état mémoire, $|\lfloor\mu\rfloor_1|_{\lfloor\bar{v}\rfloor_1}$ et $|\lfloor\mu\rfloor_2|_{\lfloor\bar{v}\rfloor_2}$, coïncident. Dans les autres cas, chaque

projection de la configuration $f\,\vec{v}\,/\mu$ doit être réduite séparément, après application de la règle (lift-δ). Il s'agit d'une vision pessimiste de la sémantique de la primitive f, qui l'autorise potentiellement à exploiter toute l'information contenue dans ses arguments.

Lemme 8.7 *La primitive f vérifie les hypothèses 5.5, 5.6 et 5.7.* □

⌜ *Preuve.* Supposons $f\,\vec{v}\,/\mu \multimap\!\!\!\!\rightarrow a+\dot{\mu}$. Les prémisses de (primitive) sont $f\,\lfloor\vec{v}\rfloor_i\,/\lfloor\mu\rfloor_i \multimap\!\!\rightarrow a+\varnothing$ **(1)** pour tout $i\in\{1,2\}$ et $\dot{\mu}=\varnothing$. Soit ϕ un renommage des adresses mémoire. Par l'hypothèse 5.1 (page 100) et (1), on a $f\,\lfloor\phi\vec{v}\rfloor_i\,/\,\lfloor\phi\mu\rfloor_i \multimap\!\!\rightarrow \phi a+\varnothing$ **(2)** pour tout $i\in\{1,2\}$. Par (primitive), on en déduit $f\,\phi\vec{v}\,/\,\phi\mu \multimap\!\!\!\!\rightarrow \phi a+\varnothing$, d'où l'hypothèse 5.5 (page 106). Par ailleurs, par (1), le résultat a est un résultat du langage Core ML, et ne comporte donc pas de crochet. On en déduit que $\multimap\!\!\!\!\rightarrow$ vérifie l'hypothèse 5.6 (page 107). Enfin, a ne comportant pas de crochets, (1) implique $f\,\lfloor\vec{v}\rfloor_i\,/\,\lfloor\mu\rfloor_i \multimap\!\!\rightarrow \lfloor a\rfloor_i+\varnothing$, pour tout $i\in\{1,2\}$, d'où l'hypothèse 5.7 (page 107). ⌟

8.2.2 Typage

Dans l'implémentation la plus récente du langage Caml, Objective Caml [LDG+b], certaines primitives génériques peuvent être appliquées à des valeurs contenant des λ-abstractions (représentées, à l'exécution, par des *clôtures*). Par exemple, lorsqu'elle est appliquée à des valeurs fonctionnelles, l'égalité polymorphe (`=`) renvoie `true` si les deux fonctions sont représentées « physiquement » par la même clôture en mémoire, et lève une exception sinon. Ce comportement relativement *ad hoc* n'est pas spécifié dans la documentation du langage. De plus, il a probablement été retenu pour des raisons d'efficacité de l'implémentation plus que pour sa pertinence. De manière plus intéressante, les primitives de sérialisation, comme `Marshal.to_string` traitent les valeurs fonctionnelles en considérant *grosso modo* les clôtures comme un tableau de valeurs associé à un pointeur de code. Dans la formalisation du langage adoptée dans cette thèse, ce traitement revient à inspecter le code situé sous les λ-abstractions. Il paraît difficile de prendre en compte cette possibilité dans l'analyse de flots d'information étudiée jusqu'ici : en effet, les types flèches décrivent les résultats et les effets produits par les fonctions, mais pas leur implémentation. Ainsi, deux fonctions dont les implémentations diffèrent par des données secrètes peuvent recevoir des types publics : par exemple, quelque soit le niveau associé à la variable y, la fonction $\lambda x.(y;\widehat{0})$ peut recevoir le type $\mathsf{unit} \xrightarrow{\top\,[\partial\bot]\,\bot} \mathsf{int}\,\bot$, alors qu'une primitive telle que `Marshal.to_string` est susceptible de distinguer les clôtures représentant les fonctions obtenues pour des valeurs différentes de y.

À l'inverse, le langage SML interdit, par le typage, l'application des primitives génériques aux valeurs fonctionnelles : le système de type est étendu en introduisant une notion de *types égalité* (*equality types* en anglais), auxquels les primitives génériques sont restreintes. Il s'agit de l'approche suivie dans cette thèse. Pour cela, je suppose distingué un sous-ensemble $\mathcal{C}_{eq}$ de ***constructeurs de types égalité***. Puisque ces constructeurs sont utilisés pour décrire des valeurs ne contenant pas de λ-abstraction, leurs paramètres sont tous supposés être de sorte Atom ou Type, et covariants ou invariants. De plus, je suppose que chaque constructeur de type égalité a au moins un paramètre gardé : cette hypothèse est utilisée pour prouver le lemme 8.11. Un type brut est égalité si et seulement si tous les constructeurs de types qu'ils contient sont dans $\mathcal{C}_{eq}$. L'ensemble des types bruts égalité est noté $\mathcal{T}_{eq}$. Pour assurer que toute valeur close admettant un type égalité dans MLIF($\mathcal{T}$) ne contient pas de λ-abstraction, il suffit de supposer que $\to$ n'est pas dans $\mathcal{C}_{eq}$ et que les schémas des constructeurs de valeurs vérifient la propriété suivante.

Hypothèse 8.8 *Pour tout constructeur k, si $\vdash k : \vec{t}\to t$ et $t\in\mathcal{T}_{eq}$ alors $\bar{t}\subseteq\mathcal{T}_{eq}$* □

Pour formuler la règle de typage des primitives génériques, j'introduis un prédicat binaire $\blacktriangleleft$ de signature $\mathsf{Atom}\cdot\mathsf{Type}$, défini inductivement par la règle suivante :

$$\frac{c\in\mathcal{C}_{eq} \qquad c :: \kappa_1\cdots\kappa_n \qquad \forall j\in[1,n]\begin{cases}\kappa_j=\mathsf{Atom}\Rightarrow t_j\le\ell\\ \kappa_j=\mathsf{Type}\Rightarrow t_j\blacktriangleleft\ell\end{cases}}{c\,t_1\cdots t_n\blacktriangleleft\ell}$$

En bref, $t \blacktriangleleft \ell$ est valide si et seulement si t est un type égalité et *toutes* les annotations de sécurité qui apparaissent dans t, y compris dans ses sous-termes, sont inférieures ou égales à ℓ. Cette définition mime le comportement des primitives génériques qui traversent les structures de données de manière récursive. L'hypothèse suivante généralise l'hypothèse 8.8 sur les types égalité à la définition de $\blacktriangleleft$ en prenant en compte les annotations de sécurité.

Hypothèse 8.9 *Pour tout constructeur k, si $\vdash k : \vec{t} \to t$ et $t \blacktriangleleft \ell$ alors, pour tout $t' \in \bar{t}$, $t' \blacktriangleleft \ell$.* □

Ainsi, si un constructeur produit des valeurs ayant un type égalité dont les annotations sont bornées par ℓ, alors chacun de ses arguments doit également avoir un type égalité dont les annotations sont bornées par ℓ. Il est facile de vérifier que chacun des constructeurs considérés jusqu'à présent remplit cette hypothèse. Le prédicat $\blacktriangleleft$ satisfait les propriétés suivantes.

Lemme 8.10 *Soit t, t' des types de sorte* Type *et ℓ un atome. Si $t' \leq t$ et $t \blacktriangleleft \ell$ alors $t' \blacktriangleleft \ell$.* □

⌜ *Preuve.* Par induction sur la structure du type t. Puisque t est de sorte Type, il existe $t_1, \ldots, t_n$ et c tels que $t = c\,t_1 \cdots t_n$ **(1)**. Posons $c :: \kappa_1 \cdots \kappa_n$. Puisque $t \blacktriangleleft \ell$, on a $c \in \mathcal{C}_{eq}$ **(2)** et, pour tout $j \in [1, n]$, si $\kappa_j =$ Atom alors $t_j \leq \ell$ **(3)**, et si $\kappa_j =$ Type alors $t_j \blacktriangleleft \ell$ **(4)**. Par (2), tous les paramètres de c sont covariants ou invariants. Puisque $t' \leq t$, on déduit de (1) qu'il existe $t'_1, \ldots, t'_n$ tels que $t' = c\,t'_1 \cdots t'_n$ **(5)** et, pour tout $j \in [1, n]$, $t'_j \leq t_j$ **(6)**. Soit $j \in [1, n]$. Si $\kappa_j =$ Atom alors, par (6) et (3), on a $t'_j \leq \ell$ **(7)**. Si $\kappa_j =$ Type alors, par l'hypothèse d'induction appliquée à (6), (4), on a $t'_j \blacktriangleleft \ell$ **(8)**. Par (2), (7) et (8), grâce à (5), on obtient le but recherché : $t \blacktriangleleft \ell$. ⌟

Lemme 8.11 *Soit t un type de sorte* Type *et ℓ, ℓ' des atomes. Si $\ell \lhd t$ et $t \blacktriangleleft \ell'$ alors $\ell \leq \ell'$.* □

⌜ *Preuve.* Par induction sur la structure du type t. Puisque t est de sorte Type, il existe $t_1, \ldots, t_n$ et c tels que $t = c\,t_1 \cdots t_n$ **(1)**. Posons $c :: \kappa_1 \cdots \kappa_n$. Puisque $t \blacktriangleleft \ell'$, on a $c \in \mathcal{C}_{eq}$ **(2)** et pour tout $j \in [1, n]$, si $\kappa_j =$ Atom alors $t_j \leq \ell'$ **(3)**, si $\kappa_j =$ Type alors $t_j \blacktriangleleft \ell'$ **(4)**. Par (2), guarded(c) n'est pas vide ; soit j l'un de ses éléments. Si $\kappa_j =$ Atom alors, grâce à (2) et puisque $\ell \lhd t$, on a $\ell \leq t_j$. En utilisant (3), on obtient le but recherché : $\ell \leq \ell'$. Si $\kappa_j =$ Type alors, grâce à (2) et puisque $\ell \lhd t$, on a $\ell \lhd t_j$ **(5)**. En appliquant l'hypothèse d'induction à (5) et (4), on obtient également le but : $\ell \leq \ell'$. ⌟

Lemme 8.12 *Supposons $\varnothing, M \vdash_H v : t$ et $M \vdash_H \mu$ avec $t \blacktriangleleft \ell$. Si $\lfloor v \rfloor_1 \neq \lfloor v \rfloor_2$ ou $|\lfloor \mu \rfloor_1|_{\lfloor v \rfloor_1} \neq |\lfloor \mu \rfloor_2|_{\lfloor v \rfloor_2}$ alors $\ell \in H$.* □

⌜ *Preuve.* Supposons $\varnothing, M \vdash_H v : t$ **($\mathbf{H_1}$)** et $M \vdash_H \mu$ **($\mathbf{H_2}$)** avec $t \blacktriangleleft \ell$ **($\mathbf{H_3}$)**. Grâce au lemme 8.10, je peux supposer, sans perte de généralité, que la dérivation de (H_1) ne se termine pas par une instance de V-SUB. Je procède par induction sur la valeur v.

∘ *Cas $v = x$.* L'environnement du jugement (H_1) étant vide, ce cas ne peut intervenir.

∘ *Cas $v = m$.* La dérivation du jugement (H_1) se termine par une instance de V-LOC. On a donc $t =$ ref $M(m)\star$ **(1)**. Grâce à (H_2), on en déduit $\varnothing, M \vdash \mu(m) : M(m)$ **(2)**. Par (1), grâce à l'invariance du constructeur de type ref pour son premier argument, (H_3) implique $M(m) \blacktriangleleft \ell$ **(3)**. L'hypothèse d'induction appliquée à (2), (H_2) et (3) permet de conclure.

∘ *Cas $v = \lambda x.e$.* La dérivation du jugement (H_1) se termine par une instance de V-ABS. On en déduit que t est un type flèche. Puisque $\to \notin \mathcal{C}_{eq}$, l'hypothèse ($H_3$) donne une contradiction.

∘ *Cas $v = k\,v_1 \cdots v_n$.* Il existe j tel que $\lfloor v_j \rfloor_1 \neq \lfloor v_j \rfloor_2$ ou $|\lfloor \mu \rfloor_1|_{\lfloor v_j \rfloor_1} \neq |\lfloor \mu \rfloor_2|_{\lfloor v_j \rfloor_2}$. La dérivation du jugement (H_1) se termine par une instance de V-CONSTRUCTOR, parmi les prémisses de laquelle on trouve $\varnothing, M \vdash v_j : t_j$ **(1)** et $\vdash k : t_1 \cdots t_n \to t$ **(2)**. Par l'hypothèse 8.9, (2) et (H_3) impliquent $t_j \blacktriangleleft \ell$ **(3)**. En appliquant l'hypothèse d'induction à (1), (H_2) et (3), on obtient le but : $\ell \in H$.

∘ *Cas $v = \langle v_1 \mid v_2 \rangle$.* La dérivation du jugement (H_1) se termine par une instance de V-BRACKET, parmi les prémisses de laquelle on a $pc \in H$ **(1)** et $pc \lhd t$ **(2)**. Puisque $t \blacktriangleleft \ell$, (2) et le lemme 8.11 donnent $pc \leq \ell$. L'ensemble H étant supérieurement clos, on en déduit, grâce à (1), $\ell \in H$. ⌟

Je souhaite donner une règle de typage pour la primitive f indépendante de sa sémantique. Pour réaliser cela, je vais supposer que cette primitive est donnée avec une règle de typage suffisante pour assurer la sûreté du typage dans le sens usuel, puis montrer comment la raffiner dans le contexte de l'analyse de flots d'information.

Hypothèse 8.13 *Supposons* $\vdash f : t_1 \cdots t_n \xrightarrow{pc\,[r]} t$. *Soit* $v_1, \ldots, v_n$ *et* μ *des valeurs et un état mémoire Core ML. Si* $f\, v_1 \cdots v_n \,/\, \mu \multimap_f a + \varnothing$, *pour tout* $j \in [1,n]$, $\varnothing, M \vdash v_j : t_j$ *et* $M \vdash \mu$ *alors* $pc, \varnothing, M \vdash a : t\ \ [r]$. □

Hypothèse 8.14 *Si* $\vdash f : t_1 \cdots t_n \xrightarrow{pc\,[r]} t$ *alors il existe* $\ell \in \mathcal{L}$ *tel que, pour tout* $j \in [1,n]$, $t_j \blacktriangleleft \ell$, *et* $\ell \lhd t$ *et* $\vdash f : t_1 \cdots t_n \xrightarrow{pc \sqcup \ell\,[r]} t$. □

L'hypothèse 8.13 assure que le typage de la primitive f est sûr dans le sens usuel. Cet énoncé est identique à celui de l'hypothèse 6.14 (page 126), mais il est restreint à des valeurs et un état mémoire Core ML, de telle sorte que les annotations de sécurité des types n'interviennent pas. Notons toutefois que les annotations portées par la flèche, pc et r, doivent correctement décrire les exceptions potentiellement levées par la primitive : dans le cas où $a = \mathsf{raise}\,\xi\,\star$, l'inégalité $pc \leq r(\xi)$ doit être vérifiée. La deuxième hypothèse donne une condition suffisante pour obtenir la non-interférence : les types des arguments doivent être des types égalité, et tout niveau de sécurité apparaissant dans l'un de ses types doit être injecté dans le type t du résultat, et dans la rangée r, pour les exceptions potentiellement levées par f. Il est intéressant de lire ces deux hypothèses de la manière suivante : *étant donnée une règle de typage quelconque pour la primitive* f *assurant la sûreté du typage dans le sens habituel (hypothèse 8.13), on peut former une règle correcte pour la non-interférence en ajoutant les contraintes indiquées par l'hypothèse 8.14*. Pour montrer la correction de ce mécanisme, il suffit d'établir que, pour la primitive f munie de la sémantique décrite à la section précédente, les hypothèses 8.13 et 8.14 impliquent les hypothèses 6.14 et 6.15.

Propriété 8.15 *Les hypothèses 8.13 et 8.14 impliquent les hypothèses 6.14 et 6.15.* □

⌜ *Preuve.*

∘ *Hypothèse 6.14.* Supposons $\vdash f : t_1 \cdots t_n \xrightarrow{pc\,[r]} t$ **(1)**, $\forall j \in [1,n]\ \varnothing, M \vdash_H v_j : t_j$ **(2)**, $M \vdash_H \mu$ **(3)** et $f\, v_1 \cdots v_n \,/\, \mu \multimap_f\!\!\!\!\rightarrow\ a + \varnothing$ **(4)**. On pose $M' = M$. Par (4) et (primitive), on a $f\, \lfloor v_1 \rfloor_1 \cdots \lfloor v_n \rfloor_1 / \lfloor \mu \rfloor_1 \multimap_f a + \varnothing$ **(5)**. Par le lemme 6.5 (page 123), (2) implique $\forall j \in [1,n]\ \ \varnothing, M \vdash \lfloor v_j \rfloor_1 : t_j$ **(6)**, et (3) implique $\lfloor M \rfloor_1 \vdash_H \mu$ **(7)**. En utilisant l'hypothèse 8.13, (1), (5), (6) et (7) donnent le but : $pc, \varnothing, M' \vdash a : t\ \ [r]$.

∘ *Hypothèse 6.15.* Supposons $\vdash f : t_1 \cdots t_n \xrightarrow{pc\,[r]} t$ **(1)**, $\forall j \in [1,n]\ \ \varnothing, M \vdash_H v_j : t_j$ **(2)**, $M \vdash_H \mu$ **(3)**, $\vec{v} \,/\, \mu \not\downarrow_f$ **(4)** et $\forall i \in \{1,2\}\ \ \lfloor \vec{v} \rceil_i \,/\, \lfloor \mu \rfloor_i \downarrow_f$ **(5)**. Par (1) et l'hypothèse 8.14, il existe $\ell \in \mathcal{L}$ tel que, pour tout $j \in [1,n]$, $t_j \blacktriangleleft \ell$ **(6)**, $\ell \lhd t$ **(7)** et $\vdash f : t_1 \cdots t_n \xrightarrow{pc\,[r]} t$ **(8)**. Par (4) et (5), il existe a_1 et a_2 tels que $a_1 \neq a_2$ et, pour tout $i \in \{1,2\}$, $f\, \lfloor \vec{v} \rceil_i \,/\, \lfloor \mu \rfloor_i \multimap_f a_i + \varnothing$. Par les hypothèses 5.2 et 8.6, on en déduit que $\lfloor \vec{v} \rceil_1 \neq \lfloor \vec{v} \rceil_2$ ou $|\lfloor \mu \rfloor_1|_{\lfloor \vec{v} \rceil_1} \neq |\lfloor \mu \rfloor_2|_{\lfloor \vec{v} \rceil_2}$. Par le lemme 8.12, (2), (3) et (6), $\ell \in H$ **(9)** s'ensuit. Les assertions (9), (8) et (7) donnent le but. ⌟

Je viens de donner une méthode générale pour prouver la correction des règles de typage pour les primitives génériques dans le système MLIF($\mathcal{T}$) : grâce à la propriété 8.15, il suffit en effet de vérifier les hypothèses 8.13 et 8.14 pour chacune d'entre elles. Je n'ai pas besoin de m'intéresser ici aux règles de typage MLIF($\mathcal{X}$) : leur correction peut (doit) toujours être établie *via* leur interprétation dans le modèle, comme expliqué dans l'hypothèse 7.1 (page 137). Dans la section suivante, j'illustre la mise en œuvre de ce mécanisme général à travers quelques exemples.

8.2.3 Applications

Notons tout d'abord que la propriété 8.15 permet d'établir la correction du typage de l'addition sur les entiers d'une manière plus simple que la méthode directe que j'ai utilisée section 6.5

(page 129). Pour mémoire, la primitive $\widehat{+}$ est typée par la règle suivante :

$$\vdash \widehat{+} : \mathsf{int}\,\ell \cdot \mathsf{int}\,\ell \xrightarrow{\top\,[\partial\bot]} \mathsf{int}\,\ell$$

Puisque $\widehat{+}$ produit toujours comme résultat une valeur entière, il est immédiat que cette règle satisfait l'hypothèse 8.13. On a de plus $\mathsf{int}\,\ell \blacktriangleleft \ell$ et $\ell \lhd \mathsf{int}\,\ell$, ce qui donne l'hypothèse 8.14.

Le traitement des opérateurs de comparaison générique est plus intéressant, et constitue la vraie motivation pour le développement de cette approche. Considérons par exemple l'opérateur d'égalité polymorphe $=$:

$$\frac{t \blacktriangleleft \ell}{\vdash = : t \cdot t \xrightarrow{\top\,[\partial\bot]} \mathsf{bool}\,\ell}$$

Je ne définis pas sa sémantique, puisque cela nécessiterait une définition co-inductive quelque peu compliquée, qui n'est pas nécessaire pour mon propos. Disons simplement qu'il retourne toujours une valeur booléenne, de telle sorte que l'hypothèse 8.13 est toujours vérifiée. Puisque cet opérateur traverse les structures de données récursivement, le résultat d'une comparaison peut révéler des informations sur des sous-termes arbitraires. La prémisse $t \blacktriangleleft \ell$ reflète ce flot potentiel en contraignant le niveau ℓ à être supérieur ou égal à chaque annotation qui apparaît dans t. Elle assure la validité de l'hypothèse 8.14 : on a $t \blacktriangleleft \ell$ et $\ell \lhd \mathsf{bool}\,\ell$. Une règle similaire peut être donnée pour chaque opérateur de comparaison polymorphe du langage Caml. Une exception est toutefois l'égalité physique ==, qui ne peut être définie dans la formalisation du langage étudiée dans cette thèse, puisque seules les valeurs mutables ont une adresse dans la sémantique de Core ML.

Enfin, les fonctions génériques de hachage et de sérialisation peuvent être traitées similairement, ce qui donne les règles suivantes :

$$\frac{t \blacktriangleleft \ell}{\vdash \mathsf{hash} : t \xrightarrow{\top\,[\partial\bot]} \mathsf{int}\,\ell} \qquad \frac{t \blacktriangleleft \ell}{\vdash \mathsf{marshal} : t \xrightarrow{\top\,[\partial\bot]} \mathsf{int}\,\ell}$$

Dans l'analyse de flots d'information développée par Myers pour le langage Java [Mye99a, Mye99b], le hachage des structures de données est effectué en équipant chaque classe d'une méthode hashCode, qui est déclarée dans la classe Object avec la signature int{this} hashCode (). Une ré-implémentation de hashCode par une sous-classe de Object doit également satisfaire cette signature. Par conséquent, elle peut seulement utiliser des champs étiquetés this. Par exemple, la classe paramétrique Vector[L] doit calculer un haché d'une manière qui ne dépend pas de la longueur du vecteur ou de son contenu, puisque leur étiquette est L. Naturellement, cela limite sérieusement l'utilité de la méthode hashCode.

CHAPITRE 9 NEUF

Discussion

9.1 À propos de l'ordre d'évaluation

Comme expliqué au chapitre 5 (page 97), la syntaxe du langage Core ML rend l'ordre d'évaluation des différentes composantes d'une expression totalement explicite. En pratique, il est possible d'autoriser une syntaxe plus flexible, à condition de déterminer préalablement une stratégie d'évaluation. Par exemple, si on choisit d'évaluer les expressions de gauche à droite, alors $e_1\, e_2$ (l'application d'une expression à une autre expression) est du sucre syntaxique pour $\mathsf{bind}\, x_1 = e_1\ \mathsf{in}\ \mathsf{bind}\, x_2 = e_2\ \mathsf{in}\ x_1\, x_2$ (où $x_1 \# \mathrm{fpv}(e_2)$). Cela donne la règle de typage dérivée suivante :

$$\frac{pc, \Gamma, M \vdash e_1 : t' \xrightarrow{pc \sqcup \ell \sqcup \Uparrow(r_1 \sqcup r_2)\,[r]\,\ell} t\ \ [r_1] \qquad pc \sqcup \Uparrow r_1, \Gamma, M \vdash e_2 : t'\ \ [r_2] \qquad \ell \lhd t}{pc, \Gamma, M \vdash e_1\, e_2 : t\ \ [r \sqcup r_1 \sqcup r_2]}$$

(Je donne dans cette section les règles pour MLIF($\mathcal{T}$), celles de MLIF($\mathcal{X}$) s'obtenant de manière analogue.) Inversement, avec une stratégie d'évaluation fixée de droite à gauche, l'application $e_1\, e_2$ est encodée comme $\mathsf{bind}\, x_2 = e_2\ \mathsf{in}\ \mathsf{bind}\, x_1 = e_1\ \mathsf{in}\ x_1\, x_2$ (où $x_2 \# \mathrm{fpv}(e_1)$). Cela donne une autre règle de typage, qui diffère de la précédente de par le niveau de sécurité sous lequel chaque sous-expression est typée :

$$\frac{pc \sqcup \Uparrow r_2, \Gamma, M \vdash e_1 : t' \xrightarrow{pc \sqcup \ell \sqcup \Uparrow(r_1 \sqcup r_2)\,[r]\,\ell} t\ \ [r_1] \qquad pc, \Gamma, M \vdash e_2 : t'\ \ [r_2] \qquad \ell \lhd t}{pc, \Gamma, M \vdash e_1\, e_2 : t\ \ [r \sqcup r_1 \sqcup r_2]}$$

Dans chaque cas, la sous-expression qui est évaluée en second est typée sous un niveau plus haut, qui reflète le fait que, si elle est exécutée, alors l'évaluation de l'autre sous-expression s'est terminée normalement.

Certaines variantes de ML, comme Caml Light [Ler] ou Objective Caml [LDG+b], laissent l'ordre d'évaluation non spécifié. Il est possible de donner une règle de typage conservative, qui est sûre à la fois pour l'évaluation de gauche à droite et de droite à gauche :

$$\frac{pc \sqcup \Uparrow r_2, \Gamma, M \vdash e_1 : t' \xrightarrow{pc \sqcup \ell \sqcup \Uparrow(r_1 \sqcup r_2)\,[r]\,\ell} t\ \ [r_1] \qquad pc \sqcup \Uparrow r_1, \Gamma, M \vdash e_2 : t'\ \ [r_2] \qquad \ell \lhd t}{pc, \Gamma, M \vdash e_1\, e_2 : t\ \ [r \sqcup r_1 \sqcup r_2]}$$

Cette règle type l'expression e_{i_1} sous le niveau $pc \sqcup \Uparrow r_{i_2}$, pour tous $\{i_1, i_2\} = \{1, 2\}$. Elle n'est cependant pas satisfaisante. Supposons en effet que e_1 et e_2 lèvent respectivement des exceptions

ξ_1 et ξ_2. Alors, puisque E-RAISE annote chaque exception avec le niveau *pc* courant, et puisque ce niveau ne peut que croître dans les dérivations de typage, on a nécessairement $\Uparrow r_1 \leq r_2(\xi_2)$ et $\Uparrow r_2 \leq r_1(\xi_1)$. Bien sûr, par définition, nous avons également $r_1(\xi_1) \leq \Uparrow r_1$ et $r_2(\xi_2) \leq \Uparrow r_2$. En combinant ces quatre inégalités, on obtient $r_1(\xi_1) = \Uparrow r_1$ et $r_2(\xi_2) = \Uparrow r_2$. En d'autres termes, si les deux expressions e_1 et e_2 sont susceptibles de lever au moins une exception, alors *toutes* les exceptions levées par e_1 et e_2 doivent avoir le *même* niveau dans r_1 et r_2, respectivement. Pour conclure, ne pas spécifier l'ordre d'évaluation entraîne une perte de précision importante de l'analyse. L'implémentation actuelle des langages Caml Light et Objective Caml utilise l'ordre d'évaluation de droite à gauche. Pour notre dessein, celui-ci doit être inclus dans la spécification du langage.

9.2 À propos des exceptions

▶ **Une comparaison avec JFlow/JIF** Le lecteur peut remarquer que les résultats normaux et exceptionnels ne sont pas traités d'une manière symétrique dans le système que j'ai présenté. En effet, dans un jugement $pc, X, M \vdash e : t \ [r]$, la rangée r associe un niveau de sécurité à chaque nom d'exception, pour enregistrer la quotité d'information obtenue en observant ce nom d'exception particulier. Cependant, aucun niveau d'information n'est explicitement associé à la terminaison normale de l'expression e. À la place, la règle de typage pour la composition séquentielle, E-BIND, utilise $\Uparrow r$ comme approximation.

À l'inverse, les ensembles de *path labels* X de Myers [Mye99a, Mye99b] enregistrent le niveau de sécurité associé à la terminaison normale grâce à un *label* spécial $\underline{\mathtt{n}}$, qui est ensuite utilisé dans la règle de séquence. Cependant, ce niveau est typiquement une borne supérieure pour la valeur du *pc* à l'intérieur de chaque sous-expression de l'expression considérée, de telle sorte que, avec cette seule règle, le système serait très restrictif. Pour pallier à ce problème, Myers ajoute une règle non dirigée par la syntaxe, la règle *single-path*, qui permet de re-descendre le niveau $X[\underline{\mathtt{n}}]$ à $\underline{\emptyset}$ si on peut montrer que l'expression considérée termine toujours normalement.

Le système MLIF($\mathcal{X}$) ne nécessite pas de règle *single-path* : en effet, quand r_1 est $\partial\bot$ alors $\Uparrow r_1$ est $\bot$, et E-BIND type les expressions e_1 et e_2 avec le même niveau *pc*, comme désiré. Le système de Myers est plus précis dans quelques situations, qui font intervenir une expression qui ne termine *jamais* normalement. L'expérience montre que cette différence est marginale en pratique. La règle *single-path* nécessite de distinguer $\underline{\emptyset}$ de $\bot$ (*i.e.* les expressions qui ne lèvent pas d'exceptions de celles qui lèvent des exceptions de niveau $\bot$), ce que je n'ai pas fait, obtenant ainsi un système plus simple. De manière plus importante, elle nécessite de compter le nombre d'entrées distinctes de $\underline{\emptyset}$ dans une rangée. En présence de variables de rangées, cela nécessite des formes de contraintes complexes, qu'il est souhaitable d'éviter. Cette difficulté n'apparaît pas dans le système de Myers puisqu'il est basé sur les clauses `throws` monomorphes de Java.

▶ **Exceptions de première classe** Comme je l'ai expliqué au chapitre 5 (page 97), j'ai choisi de faire des exceptions des entités de seconde classe du langage Core ML. Cette restriction, qui est partiellement compensée par l'ajout des constructions `propagate` et `finally`, est motivée par la volonté d'obtenir des types simples et compréhensibles par le programmeur. Je donne ici les principales modifications à apporter au système pour retrouver les exceptions parmi les valeurs du langage.

La modification de la syntaxe et de la sémantique du langage est immédiate. Il suffit en effet de considérer que les noms d'exceptions sont des constructeurs unaires, et d'étendre les expressions comme suit :

$$e ::= \ldots \mid \mathsf{raise}\, v$$

Les résultats sont toujours les valeurs et les exceptions non rattrapées de la forme $\mathsf{raise}\,\xi\, v$, ces dernières devant désormais être lues $\mathsf{raise}(\xi\, v)$. On peut également considérer de nouvelles formes de clauses, afin de profiter du statut de première classe des exceptions lors de leur interception,

comme par exemple

$$h ::= \ldots \mid x \to e$$

La clause $x \to e$ rattrape toutes les exceptions. Elle lie l'exception rattrapée à la variable x avant d'exécuter e. Cette expression peut alors effectuer des opérations arbitraires sur cette valeur, comme par exemple faire des tests sur le nom de l'exception, accéder à ses arguments ou la propager.

Le typage des exceptions de première classe nécessite de permettre la distinction entre une expression qui lève une exception ξ au niveau $\bot$ et une expression qui ne lève pas l'exception ξ. Pour cela, il faut modifier la nature des rangées qui deviennent des fonctions des noms d'exceptions vers les ***alternatives***, similaires aux *path labels* de Myers :

$$b ::= \underline{\emptyset} \mid \ell \qquad \text{(alternative)}$$

Je note Alt la sorte associée aux alternatives dans la nouvelle définition des types et des types bruts qui correspond à cette extension. La forme des jugements de typage n'est pas modifiée. Étant donnée une rangée brute r décrivant les effets d'une expression e dans le jugement $pc, X, M \vdash e : t \ [r]$, on a $r(\xi) = \underline{\emptyset}$ si l'expression e ne peut lever l'exception ξ, et $r(\xi) = \ell$ si e peut lever l'exception ξ avec un certain niveau ℓ. L'ordre partiel sur les niveaux est étendu aux alternatives supposant $\underline{\emptyset} \leq \ell$ pour tout niveau ℓ. Les valeurs exceptions sont typées en utilisant un constructeur de type spécifique exn, de signature $\mathsf{Row}_{\mathcal{E}}\,\mathsf{Alt}^{+} \cdot \mathsf{Atom}^{+\lhd}$. Chaque constructeur de valeur correspondant à un nom d'exception ξ est typé par l'axiome

$$\vdash \xi : \mathrm{type}(\xi) \to \mathsf{exn}(\xi : \bot; \partial\underline{\emptyset})\ \bot$$

Ainsi, quand une valeur d'exception a le type $\mathsf{exn}\, r\, \ell$, alors la rangée r donne de l'information sur le nom de l'exception. Plus précisément, pour tout $\xi \in \mathcal{E}$, si $r(\xi) = \underline{\emptyset}$ alors le nom de l'exception ne peut pas être ξ. Au contraire, si $r(\xi) \geq \bot$ alors l'exception peut être nommée ξ. Le niveau ℓ, deuxième paramètre du constructeur exn, décrit quant à lui la quotité d'information associée à la valeur exception elle-même, comme dans un type somme habituel. La règle de typage pour la construction $\mathsf{raise}\, v$ est la suivante :

$$\frac{X, M \vdash v : \mathsf{exn}\, r\, \ell \qquad \forall \xi \in \mathcal{E}\ \ (r(\xi) \neq \underline{\emptyset}) \Rightarrow (pc \sqcup \ell \leq r(\xi))}{pc, X, M \vdash \mathsf{raise}\, v : \star\ \ [r]}$$

La première prémisse assure que la valeur v a un type d'exception. La deuxième prémisse contraint le niveau du contexte pc et le niveau ℓ de la valeur v à être des bornes inférieures pour les entrées de la rangée r qui ne sont pas égales à $\underline{\emptyset}$. Puisqu'une instance de V-SUB peut être utilisée pour dériver la première prémisse, cette contrainte ne restreint pas le type de la valeur v mais porte en fait sur la rangée apparaissant dans la conclusion. Le typage des autres formes d'expressions ne nécessite pas de modification importante du système. La principale nouveauté réside dans la définition de l'opérateur de borne supérieure des rangées, $\Uparrow$, qui doit désormais ignorer les entrées égales à $\underline{\emptyset}$: on pose $\Uparrow r = \sqcup\{\, r(\xi) \mid r(\xi) \neq \underline{\emptyset} \,\}$. La nouvelle forme de clauses peut être typée grâce à la règle suivante :

$$\frac{pc \sqcup \Uparrow r_{|\Xi}, X[\![x \mapsto \mathsf{exn}\, r\, \star]\!], M \vdash e_2 : t\ \ [r'] \qquad \Uparrow r_{|\Xi} \lhd t}{pc, X, M \vdash (\Xi\, x \to e_2) : r \Rightarrow t\ \ [r']}$$

La rangée r, qui décrit les exceptions potentiellement rattrapées par la clause, est utilisée pour former le type associé à la variable x dans l'environnement sous lequel e_2 est typé. Par ailleurs, l'information supplémentaire portée par les rangées grâce à l'alternative $\underline{\emptyset}$ permet éventuellement de donner une règle de typage plus flexible pour la construction $\mathsf{finally}$, qui ne restreint pas la seconde expression à ne pas lever d'exception :

$$\frac{pc, X, M \vdash e_1 : t\ \ [r_1] \qquad pc, X, M \vdash e_2 : \star\ \ [r_2] \qquad \forall \xi \in \mathcal{E}\ \ (r_1(\xi) \neq \underline{\emptyset}) \Rightarrow (\Uparrow r_2 \leq r_1(\xi))}{pc, X, M \vdash e_1\ \mathsf{finally}\ e_2 : t\ \ [r_1 \sqcup r_2]}$$

La troisième prémisse reflète la prédominance des exceptions levées par e_2 sur celles produites par e_1 : observer que le résultat retourné par la construction finally est une exception ξ originaire de e_1 indique que l'évaluation de e_2 s'est terminée normalement, de telle sorte que son niveau, donné par $r_1(\xi)$, doit être supérieur ou égal à $\Uparrow r_2$.

Cette extension du système introduit deux complications dans le processus de synthèse des types et dans la forme des informations de typage présentées à l'utilisateur. Tout d'abord, elle requiert l'introduction d'une nouvelle notion, les alternatives, qui apparaissent nécessairement dans les types inférés, *via* les rangées. D'autre part, la deuxième prémisse de la règle relative à la construction raise (de même que la troisième prémisse pour finally) nécessite l'introduction de formes conditionnelles dans le langage de contraintes. La résolution de ces contraintes ne pose *a priori* pas de problème technique ; la condition portant systématiquement sur la borne inférieure d'un type, il suffit de disposer d'un prédicat de la forme $(b \leq \tau)\ ?\ (\tau_1 \leq \tau_2)$, qui est satisfait si et seulement si (l'interprétation de) τ n'est pas supérieur(e) ou égal(e) à l'alternative (constante) b ou bien si $\tau_1 \leq \tau_2$. Elles entrent ainsi dans un cadre pour lequel des techniques de résolution efficaces sont connues [Pot00]. Cependant ces conditionnelles sont susceptibles d'apparaître dans les formes résolues des contraintes, ainsi que d'interférer avec les techniques de simplification des schémas de type, comme celles présentées dans la troisième partie de cette thèse, détériorant ainsi la lisibilité des informations de types affichées. C'est la raison pour laquelle il m'est apparu plus raisonnable de les éviter.

▶ **Codage des exceptions dans les sommes** Il existe un codage monadique simple des exceptions dans les sommes [Mog89, Wad92]. Ainsi, il est en principe possible de dériver un système de type pour les exceptions d'un système qui traite les sommes. Cette approche semble intéressante, puisqu'elle est systématique et produirait *a priori* un traitement symétrique des résultats normaux et exceptionnels. Cependant, les types sommes que j'ai présentés dans les chapitres précédents — aussi bien ceux des sommes binaires (section 6.5, page 129) que ceux des types variants (section 8.1, page 149) — ne sont pas suffisamment précis pour être utilisés dans le langage cible d'un tel codage.

Pour expliquer les problèmes posés par cette approche, je m'intéresse au cas simple où l'ensemble des noms d'exceptions est réduit à un élément, noté ξ_0. Le codage du langage Core ML *avec* exceptions dans Core ML *sans* exceptions consiste à traduire chaque expression e en une expression $[\![e]\!]$ telle que si e retourne la valeur v alors $[\![e]\!]$ se réduit en la valeur $\mathsf{inj}_1\,[\![v]\!]$ et si e lève l'exception $\mathsf{raise}\,\xi_0\,v$ alors $[\![e]\!]$ se réduit en $\mathsf{inj}_2\,[\![v]\!]$. La traduction se définit de manière compositionnelle sur chaque construction du langage. On a par exemple

$$[\![\mathsf{bind}\,x = e_1\ \mathsf{in}\ e_2]\!] = \ [\![e_1]\!]\ \mathsf{case}\ (\lambda x.[\![e_2]\!])\,(\lambda y.\mathsf{inj}_2\,y)$$

Avec ce codage, chaque expression du langage source reçoit un type de la forme $(t + t')^\ell$ où t est le type de la valeur potentiellement produite par e, t' le type de l'argument de l'exception potentiellement levée par e et ℓ un niveau d'information qui décrit la quotité d'information obtenue en observant la nature du résultat produit par e, valeur ou exception. La construction bind reçoit alors la règle de typage dérivée suivante :

$$\frac{pc, X \vdash e_1 : (t_1 + t')^{\ell_1} \qquad pc \sqcup \ell_1, X \vdash e_2 : (t + t')^{\ell} \qquad \ell_1 \leq \ell}{pc, X \vdash \mathsf{bind}\,x = e_1\ \mathsf{in}\ e_2 : (t + t')^{\ell}}$$

Cette règle n'est pas suffisamment précise, même dans le cas où les expressions e_1 et e_2 ne lèvent pas d'exceptions. En effet, le niveau de sécurité associé à l'expression $\mathsf{bind}\,x = e_1\ \mathsf{in}\ e_2$ doit, dans tous les cas, être supérieur ou égal à celui de l'expression e_1, quelle que soit l'expression e_2. Ainsi, si e_1 produit un résultat *secret*, alors il en est de même de $\mathsf{bind}\,x = e_1\ \mathsf{in}\ e_2$, y compris si la variable x n'est pas utilisée dans e_2.

Ce constat suggère naturellement de chercher des systèmes de types plus fins pour décrire les flots d'information en présence de types sommes. En plus de son intérêt immédiat, un tel système

permettrait éventuellement d'obtenir une analyse satisfaisante des exceptions. Cette question est l'objet de la prochaine section.

9.3 À propos des types sommes

Pendant mon travail de thèse, je me suis intéressé à un système de type particulièrement fin pour l'analyse de flots d'information dans un langage doté de types sommes. Cette étude, dont je donne ici un bref aperçu à travers un exemple, a été publiée en anglais dans les actes du quinzième *IEEE Computer Security Foundations Workshop* [Sim02b].

Supposons le langage équipé de trois constructeurs de variantes constants — notés A, B et D — appartenant au même type de données. Je considère le fragment de programme composé des deux phrases suivantes :

$$\begin{aligned} \mathsf{let}\, y_1 \;&=\; \mathsf{if}\, x_1 \,\mathsf{then}\, (\mathsf{if}\, x_2 \,\mathsf{then}\, A \,\mathsf{else}\, B) \\ &\qquad\quad \mathsf{else}\, (\mathsf{if}\, x_3 \,\mathsf{then}\, A \,\mathsf{else}\, D) \\ \mathsf{let}\, y_2 \;&=\; \mathsf{case}\, y_1\, [A \mid B \mapsto 1; D \mapsto 0] \end{aligned}$$

La valeur à laquelle la variable y_1 est liée — A, B ou D — dépend des variables booléennes x_1, x_2 et x_3. Le typage des sommes que j'ai considéré jusqu'à présent décrit ce flot d'information en contraignant le niveau de sécurité de y_1 à être supérieur ou égal au niveau de chacun des trois booléens. La même contrainte porte ensuite sur le niveau de sécurité de y_2, puisque la valeur de cette dernière est calculée en observant y_1. En résumé, le système de type détecte un potentiel flot d'information de x_1, x_2 et x_3 vers y_2. Cependant, la valeur de y_2 ne dépend clairement pas de celle de x_2, puisque tester si y_1 vaut D plutôt que A ou B ne donne pas d'information sur x_2. En effet, si la valeur de cette variable a été consultée lors de l'évaluation de la première phrase, alors y_1 vaut soit A soit B, mais pas D.

Dans le système fin, chaque type somme est annoté par une *rangée* et une *matrice triangulaire*. La rangée (le terme est utilisé ici dans un sens légèrement différent du reste de cette thèse) comporte une entrée pour chaque variant du type somme qui est soit Pre (si le variant peut être *présent* dans la valeur décrite) soit Abs (pour *absent*). La matrice comporte quant-à-elle un niveau d'information pour chaque paire (non ordonnée) de variants. Pour les Booléens (un type somme à deux constructeurs), cette matrice ne comporte donc qu'un seul niveau, qui a le même rôle que dans MLIF($\mathcal{T}$). Notons ainsi ℓ_1, ℓ_2 et ℓ_3 les niveaux des booléens x_1, x_2 et x_3. Similairement, une matrice de trois niveaux

$$\begin{pmatrix} \ell_{\{A,B\}} & \ell_{\{A,D\}} \\ & \ell_{\{B,D\}} \end{pmatrix}$$

est attachée au type de la variable y_1. Le niveau $\ell_{\{A,B\}}$ décrit la quotité d'information obtenue en testant si la valeur de y_1 est A ou B. Il doit ainsi être (au moins) égal à l'union des niveaux attachés au booléens x_1 et x_2, c'est-à-dire $\ell_1 \sqcup \ell_2$. De même, $\ell_{\{A,D\}}$ doit être (au moins) $\ell_1 \sqcup \ell_3$; et $\ell_{\{B,D\}}$ supérieur ou égal à ℓ_1. Dans la deuxième phrase, puisque le test permet de déterminer si la valeur de y_1 est A ou B ou bien D, ce système va approximer la quotité d'information obtenue par le niveau

$$\sqcup\{\, \ell_{\{c_1,c_2\}} \mid c_1 \in \{A,B\} \text{ et } c_2 \in \{D\} \,\}$$

c'est-à-dire $\ell_{\{A,D\}} \sqcup \ell_{\{B,D\}}$, qui est égal à $\ell_1 \sqcup \ell_3$. C'est ainsi que le système est capable d'établir l'absence de dépendance entre y_2 et x_2.

Cet exemple simple illustre la forme des types utilisé par ce système, et montre qu'il est plus expressif que celui étudié dans cette thèse. De plus, il est possible de dériver à partir de ce nouveau système une analyse des exceptions plus précise que celles développée dans cette thèse ou offerte par Jif [Mye99a, Mye99b, MNZZ01], où les effets d'une expression sont décrits à la fois par une rangée associant à chaque nom d'exception (et à un constructeur supplémentaire η pour les résultats normaux) une indication de présence, et une matrice associant à chaque paire de noms d'exceptions

un niveau d'information. On peut également voir les approches suivies dans cette thèse ou Jif comme deux restrictions particulières « à une dimension » de ce système. Le lecteur intéressé par cette correspondance est renvoyé à l'article [Sim02b].

Toutefois, il ne m'a pas semblé pertinent d'utiliser ce travail pour obtenir, dans cette thèse, l'analyse des exceptions. Du point de vue théorique, il me semble que l'approche directe est plus simple que celle procédant par codage *via* le système fin ou une restriction de celui-ci. Du point de vue pratique, le système fin me semble trop complexe — à la fois dans la formulation de ses règles de typage et dans la taille des types engendrés — pour être utilisé de manière réaliste.

TROISIÈME PARTIE

III

Synthèse de types en présence de sous-typage structurel

Dans le chapitre 7 (page 135), j'ai montré comment l'inférence de types pour le système MLIF($\mathcal{X}$) pouvait être ramenée, de manière algorithmique, à la résolution de contraintes faisant intervenir les prédicats de sous-typage et de garde. Pour terminer la définition d'un algorithme d'inférence, je donne, dans cette troisième et dernière partie de la thèse, la description d'un solveur efficace pour ces contraintes, accompagnée d'une preuve de sa correction. Celui-ci n'est toutefois pas spécifique au problème de l'analyse de flots d'information, et pourrait être utilisé pour tout système de types à base de contraintes de sous-typage structurel. En plus de déterminer la satisfiabilité des contraintes qui lui sont présentées, ce solveur s'attache à les simplifier, c'est-à-dire à les réécrire sous une forme équivalente mais plus compacte. Ce point est essentiel aussi bien pour l'efficacité du processus d'inférence, que pour l'affichage d'informations de type compréhensibles par le programmeur. Je décompose la formalisation de ce processus en deux parties. La première, décrite dans le chapitre 10 (page 171), est un algorithme de résolution et de simplification pour le noyau du langage de contraintes formé des conjonctions d'inégalités et gardes. La seconde, présentée au chapitre 11 (page 217), considère quant à elle les formes d'introduction et d'instanciation des schémas de type.

Le premier algorithme pour l'inférence de type en présence de sous-typage atomique a été proposé par Mitchell [Mit84, Mit91], puis étendu par Fuh et Mishra au sous-typage structurel [FM88, FM89]. Cependant, citant Hoang et Mitchell [HM95], « il n'a eu presque aucune utilisation pratique » avant tout parce qu'il « est inefficace et sa sortie, même pour des expressions relativement simples, apparaît excessivement longue et difficile à lire ». La complexité théorique de ce problème a été largement étudiée. Tiuryn [Tiu92] a montré que la satisfaction de contraintes de sous-typage entre atomes est PSPACE-difficile, mais, dans le cas usuel d'une union disjointe de treillis, décidable en temps linéaire. La complexité du cas général a été totalement établie par Frey [Fre97], qui a montré qu'il s'agit d'un problème PSPACE-complet. Hoang et Mitchell [HM95] ont par ailleurs

prouvé l'équivalence de la résolution de contraintes avec la typabilité dans le λ-calcul simplement typé avec sous-typage. Rehof [Reh97] a établi l'existence de types minimaux pour le sous-typage atomique, et donné une borne inférieure exponentielle sur leur taille. Dernièrement, Kuncak et Rinard [KR03] ont montré que la théorie du premier ordre du sous-typage structurel avec types non récursifs est décidable. L'algorithme exhibé par ces derniers a cependant une complexité non élémentaire. Le sous-typage structurel a également été étudié à travers des applications spécifiques. Parmi elles, on peut mentionner le travail de Foster sur les « qualificateurs de types » [FFA99, FTA02] et l'analyse de temps de liaison par contraintes Booléennes de Glynn et al. *[GSSS01].*

Dans cette thèse, mon approche de la résolution des contraintes de sous-typage structurel est tournée vers la pratique : je suis intéressé par l'obtention d'un algorithme efficace et utilisable pour typer des programmes de taille réelle. Plusieurs chercheurs ont travaillé dans cette direction pour d'autres formes de sous-typage, comme les set constraints *[AW92, AW93, Aik94, AF96, AWP96, FF97, Aik99] ou le sous-typage non structurel [TS96, Pot01b]. Bien qu'ils ne soient pas directement applicables au cas du sous-typage structurel, leurs travaux sont intéressants car ils font intervenir des techniques (généralement heuristiques) visant à réduire la taille des contraintes au cours de leur résolution. L'algorithme que je décris reprend plusieurs de ces heuristiques, en les adaptant au cas du sous-typage structurel. Son fonctionnement général est cependant très différent de celui adopté par Pottier [Pot01b] qui effectue une clôture transitive du graphe formé par les contraintes de sous-typage, entrelacée avec leur décomposition sur les types construits. Ce choix donne un algorithme au comportement cubique, difficilement améliorable car il réalise une sorte de clôture transitive dynamique.*

La stratégie de résolution mise en œuvre dans le chapitre 10 (page 171) repose de manière essentielle sur la forme de sous-typage considérée — le sous-typage structurel — pour laquelle deux types comparables doivent avoir la même structure ou squelette*, et ne différer que par des atomes. Pour exploiter efficacement cette propriété, j'introduis une nouvelle structure de données, les* multi-squelettes *(section 10.1.2, page 175), inspirée des multi-équations utilisées pour l'unification dans les théories équationnelles [Rém92]. En quelques mots, les multi-squelettes permettent d'exprimer simultanément des égalités entre types (notées =) et entre des squelettes de types (notées ≈), dans le but de réaliser un processus d'unification sur les deux en temps quasi-linéaire [Tar75, Hue76, Rém92] (section 10.2.1, page 179). La stratégie consiste ensuite à retarder autant que possible la clôture transitive en expansant la structure des types puis décomposant les contraintes de sous-typage (section 10.2.3, page 184) jusqu'à obtenir un problème atomique (section 10.2.4, page 192). Bien que l'expansion soit théoriquement susceptible d'introduire un nombre de variables exponentiel en la taille du problème initial, ce comportement est rare en pratique : par exemple, sous l'hypothèse de types de taille bornée, sa complexité reste linéaire (section 10.2.5, page 194). De plus, de manière à limiter autant que possible son impact, je propose plusieurs heuristiques à appliquer pendant le processus d'expansion dans le but de réduire le nombre de variables créées (section 10.3.2, page 199). Je termine la description de l'algorithme par une deuxième série*

de techniques de simplification, qui peuvent être appliquées au terme de la résolution, lorsqu'un problème atomique a été obtenu (section 10.3.3, page 209). En plus de permettre l'obtention d'un résultat compréhensible par le programmeur, elles sont essentielles au regard de la deuxième strate du solveur qui gère le polymorphisme : les formes d'introduction et d'instanciation de schémas nécessitent en effet de dupliquer des contraintes. Il est donc important de les rendre préalablement aussi compactes que possible.

La deuxième strate du solveur est indépendante de la nature des prédicats du langage de contraintes, ainsi que du solveur utilisé pour les résoudre. Le chapitre 11 (page 217) montre en effet, comment un solveur arbitraire — dit solveur primaire — pour le noyau du langage de contraintes défini par la grammaire

$$I ::= p\ \vec{\tau} \mid I \wedge I \mid \exists \bar{\alpha}.I$$

(où p parcourt un ensemble quelconque de prédicats) peut être étendu de manière à obtenir une procédure de résolution pour le langage complet

$$C ::= p\ \vec{\tau} \mid C \wedge C \mid \exists \bar{\alpha}.C \mid \mathsf{let}\, x : \sigma\, \mathsf{in}\, C \mid x \preceq \tau$$

En instanciant ce procédé avec l'algorithme présenté au chapitre 10 (page 171), on obtient un solveur pour le langage de contraintes utilisé par le système MLIF($\mathcal{X}$). Cette description modulaire de l'algorithme de résolution simplifie sa formalisation, en séparant des étapes orthogonales. Elle permet également de réutiliser la partie supérieure et son implémentation pour d'autres langages de contraintes, impliquant d'autres prédicats sur les types, dès lors qu'un solveur primaire est fourni. Cette séparation empêche toutefois les techniques employées par l'algorithme principal pour traiter les formes d'introduction et d'instanciation des schémas d'exploiter quelque propriété particulière des prédicats p ou du solveur primaire. Cela pourrait éventuellement menacer son efficacité. L'expérience montre toutefois qu'il ne s'agit pas d'un problème en pratique (section 12.1, page 227).

Les résultats présentés dans cette partie ont été publiés en anglais, dans les actes de la conférence *Asian Symposium on Programming Languages and Systems (APLAS'03)*.

CHAPITRE DIX

Résolution et simplification des contraintes de sous-typage structurel

Ce chapitre présente un solveur primaire efficace pour des contraintes impliquant les prédicats de sous-typage structurel et de garde. Il est en particulier approprié pour traiter les contraintes produites par le système MLIF($\mathcal{X}$). Le langage de contraintes accepté est cependant légèrement plus général ; je le précise à la section 10.1. Je procède ensuite en deux étapes. Tout d'abord, la section 10.2 (page 178) décrit le corps de la procédure de résolution, sous la forme d'un système de réécriture sur les contraintes. Cette première spécification du solveur est suffisante pour déterminer la satisfiabilité des contraintes. Cependant, elle ne fournit pas un algorithme efficace en pratique, et les résultats qu'elle produit sont généralement d'une taille trop importante pour être compréhensibles par l'utilisateur. Sa structure est cependant appropriée pour l'introduction de techniques de simplification, dont le but est de réduire la taille des contraintes manipulées tout au long du processus de résolution. Celles-ci sont décrites dans un deuxième temps, à la section 10.3 (page 195). J'espère que cette présentation par raffinement successifs permettra au lecteur de le découvrir l'algorithme de résolution de manière progressive, en s'affranchissant dans un premier temps des aspects non essentiels.

10.1 Contraintes et structures de données

10.1.1 Une généralisation de la garde

La spécification du prédicat de garde que j'ai donnée dans la deuxième partie de ce mémoire (section 6.2, page 115) est directement spécialisée pour son utilisation dans le cadre de l'analyse de flots d'information. Dans cette partie, je préfère généraliser légèrement cette définition, de manière à la rendre aussi abstraite que possible, et symétrique. Ce nouveau prédicat, noté $\lessdot$ et toujours appelé *garde*, est défini par les règles de la figure 10.1. Il peut relier deux types de sorte arbitraire (et non plus seulement un atome à un type de sorte Type). Comme celle de $\lhd$, sa définition est donnée par des règles permettant de décomposer un jugement $t_1 \lessdot t_2$ suivant la structure des types t_1 et t_2, jusqu'à obtenir des atomes.

Sur les atomes, $\lessdot$ coïncide avec l'ordre du treillis atomique $\leq_{\mathcal{L}}$, comme exprimé par la règle GRD-ATOM. Les autres règles précisent la décomposition du prédicat lorsque son membre gauche

$$\textsc{grd-atom}\ \frac{\ell_1 \leq \ell_2}{\ell_1 \lessdot \ell_2} \qquad \textsc{grd-type-left}\ \frac{\forall i \in \text{guarded-l}(c)\ \ t_i \lessdot t}{c\,t_1 \cdots t_n \lessdot t} \qquad \textsc{grd-type-right}\ \frac{\forall i \in \text{guarded-r}(c)\ \ t \lessdot t_i}{t \lessdot c\,t_1 \cdots t_n}$$

$$\textsc{grd-row-left}\ \frac{\forall \xi \in \Xi\ \ t_\xi \lessdot t}{\{\xi \mapsto t_\xi\}_{\xi\in\Xi} \lessdot t} \qquad \textsc{grd-row-right}\ \frac{\forall \xi \in \Xi\ \ t \lessdot t_\xi}{t \lessdot \{\xi \mapsto t_\xi\}_{\xi\in\Xi}}$$

Figure 10.1 – Définition de la garde ($\lessdot$)

(règles GRD-TYPE-LEFT et GRD-ROW-LEFT) ou droit (règles GRD-TYPE-RIGHT et GRD-ROW-RIGHT) est respectivement un type brut construit ou une rangée brute. Pour chaque constructeur de type c d'arité n, on suppose donnés deux ensembles non disjoints de positions covariantes de c, c'est-à-dire des parties de $\{\, i \in [1,n] \mid c.i = + \,\}$, notés guarded-l($c$) et guarded-r($c$). Ces ensembles indiquent, pour les règles GRD-TYPE-LEFT et GRD-TYPE-RIGHT, les paramètres du constructeur de c sur lesquels la garde se décompose respectivement à gauche et à droite. Enfin, sur les rangées, par les règles GRD-ROW-LEFT et GRD-ROW-RIGHT, la décomposition a lieu sur tous les champs. Notons que cette nouvelle garde permet de coder le prédicat $\lhd$ utilisé par le système MLIF($\mathcal{X}$) : à condition de choisir guarded-r(c) = guarded(c), on peut écrire $\ell \lessdot t$ pour $\ell \lhd t$. Le traitement du prédicat $\blacktriangleleft$ introduit par l'extension de MLIF($\mathcal{X}$) présentée à la section 8.2 (page 154) est discuté à la section 12.3 (page 230).

Bien que les règles définissant la relation $\lessdot$ ne soient pas dirigées par la syntaxe, elles sont toutefois des équivalences. En d'autres termes, toutes les stratégies de décomposition sur deux types bruts produisent le même ensemble d'inégalités entre atomes.

Propriété 10.1 *Les règles de la figure 10.1 sont des équivalences.* □

⌜ *Preuve.* Je montre, par induction sur la structure des types t et t', que si le jugement $t \lessdot t'$ **($\mathbf{H_1}$)** est la conclusion d'une instance d'une des règles de la figure 10.1 alors sa prémisse est également vérifiée. Je considère successivement les différentes règles dont (H_1) peut être la conclusion.

∘ *Cas* GRD-ATOM. Les types t et t' sont des atomes. On en déduit que la dérivation de (H_1) se termine par une instance de GRD-ATOM dont la prémisse est le but recherché : $t \leq_{\mathcal{L}} t'$.

∘ *Cas* GRD-TYPE-LEFT. On a $t = c\,t_1 \cdots t_n$ et on veut montrer que, pour tout $i \in$ guarded-l(c), $t_i \lessdot t'$ **(1)**. On raisonne par cas sur la règle qui termine la dérivation de (H_1). Puisque t a la sorte Type, seuls les sous-cas suivants sont envisageables :

· *Sous-cas* GRD-TYPE-LEFT. La prémisse de la règle est exactement le but recherché.

· *Sous-cas* GRD-TYPE-RIGHT. On a $t' = c'\,t'_1 \cdots t'_{n'}$ **(2)**. Soit $i' \in$ guarded-r(c') **(3)**. Parmi les prémisses de GRD-TYPE-RIGHT, on a $c\,t_1 \cdots t_n \lessdot t'_{i'}$. Ce jugement est identique à la conclusion d'une instance de GRD-TYPE-LEFT de prémisses $t_i \lessdot t_{i'}$ **(4)**, pour tout $i \in$ guarded-l(c). Par l'hypothèse d'induction, $t'_{i'}$ étant un sous-terme strict de t', on en déduit que les jugements (4) sont valides. En déchargeant (3), on en déduit que, pour tous $i \in$ guarded-l(c) et $i' \in$ guarded-r(c'), $t_i \lessdot t'_{i'}$ **(5)**. Par GRD-TYPE-LEFT, on obtient, pour tout $i \in$ guarded-l(c), $t_i \lessdot c'\,t'_1 \cdots t'_{n'}$, ce qui est, grâce à (2), le but recherché.

· *Sous-cas* GRD-ROW-RIGHT. On a $t' = \{\xi \mapsto t'_\xi\}_{\xi\in\Xi}$ **(6)**. Soit $\xi \in \Xi$ **(7)**. Parmi les prémisses de GRD-ROW-RIGHT, on a $c\,t_1 \cdots t_n \lessdot t'_\xi$. Ce jugement est identique à la conclusion d'une instance de GRD-ROW-LEFT de prémisses $t_i \lessdot t'_\xi$ **(8)**, pour tout $i \in$ guarded-l(c). Par l'hypothèse d'induction, t'_ξ étant un sous-terme strict de t', on en déduit que les jugements (8) sont valides. En déchargeant (7), on en déduit que, pour tous $i \in$ guarded-l(c) et $\xi \in \Xi$, $t_i \lessdot t'_\xi$ **(9)**. Par GRD-TYPE-LEFT, on obtient, pour tout $i \in$ guarded-l(c), $t_i \lessdot \{\xi \mapsto t'_\xi\}_{\xi\in\Xi}$, ce qui est, grâce à (6), le but recherché.

Le sous-cas GRD-ROW-LEFT est analogue au précédent, les cas GRD-TYPE-RIGHT et GRD-ROW-RIGHT s'obtiennent de manière symétrique. ⌟

De plus, puisque pour tout constructeur de type c, les ensembles guarded-l(c) et guarded-r(c) sont d'intersection non-disjointe et ne contiennent que des positions covariantes de c, la relation $\lessdot$ est transitive et compatible avec l'ordre de sous-typage : l'énoncé suivant est ainsi une généralisation de la propriété 6.1 (page 116) relative à $\lhd$.

Propriété 10.2 *Si $t_1 \lessdot t_2$ et $t_2 \lessdot t_3$ alors $t_1 \lessdot t_3$. Si $t_1 \lessdot t_2$ et $t_2 \leq t_3$ ou bien $t_1 \leq t_2$ et $t_2 \lessdot t_3$ alors $t_1 \lessdot t_3$. Si $t_1 \approx t_2$ alors ($t_1 \lessdot t$ et $t_2 \lessdot t$) est équivalent à ($t_1 \sqcup t_2 \lessdot t$) et de même ($t \lessdot t_1$ et $t \lessdot t_2$) est équivalent à ($t \lessdot t_1 \sqcap t_2$).* □

⌜ *Preuve.* Je suppose $t_1 \lessdot t_2$ **($\mathbf{H_1}$)** et $t_2 \leq t_3$ **($\mathbf{H_2}$)**. Je montre que $t_1 \lessdot t_3$ **($\mathbf{C_1}$)**, en raisonnant par induction sur la dérivation de (H_1).

∘ *Cas* GRD-ATOM. t_1 et t_2 sont des atomes. Par (H_2), t_3 est également un atome et $t_2 \leq_{\mathcal{L}} t_3$ **(1)**. La prémisse de GRD-ATOM est $t_1 \leq_{\mathcal{L}} t_2$. Grâce à la transitivité de $\leq_{\mathcal{L}}$ et par (1), on en déduit $t_1 \leq_{\mathcal{L}} t_3$. Une instance de GRD-ATOM donne le but (C_1).

∘ *Cas* GRD-TYPE-LEFT. On a $t_1 = c\,t'_1 \cdots t'_n$ **(1)** et, pour tout $i \in$ guarded-l(c), $t'_i \lessdot t_2$ **(2)**. En appliquant l'hypothèse d'induction à (2) et (H_2), on en déduit que, pour tout $i \in$ guarded-l(c), $t'_i \lessdot t_3$. Par une instance de GRD-TYPE-LEFT, on obtient $c\,t'_1 \cdots t'_n \lessdot t_3$, ce qui est, grâce à (1), le but (C_1).

∘ *Cas* GRD-TYPE-RIGHT. On a $t_2 = c\,t'_1 \cdots t'_n$ **(1)** et, pour tout $i \in$ guarded-r(c), $t_1 \lessdot t'_i$ **(2)**. Par (1) et (H_2), il existe $t''_1, \ldots, t''_n$ tel que $t_3 = c\,t''_1 \cdots t''_n$ **(3)**. Soit $i \in$ guarded-r(c) **(4)**. On a $c.i = +$, donc grâce aux égalités (1) et (3), (H_2) implique $t'_i \leq t''_i$ **(5)**. En appliquant l'hypothèse d'induction à (2) et (5), on obtient $t_1 \lessdot t''_i$. En déchargeant l'hypothèse (4), on en conclut $\forall i \in$ guarded-r(c) $\; t_1 \lessdot t''_i$, ce qui donne, par une instance de GRD-TYPE-RIGHT, $t_1 \lessdot c\,t''_1 \cdots t''_n$. Grâce à (3), il s'agit du but (C_1).

Les cas GRD-ROW-LEFT et GRD-ROW-RIGHT sont analogues aux précédents. Les autres propriétés se montrent d'une manière similaire. ⌟

La relation $\lessdot$ n'est toutefois pas une relation d'ordre (ou même un pré-ordre) puisqu'elle n'est en général pas réflexive, sauf sur les atomes : par exemple, si r est une rangée brute de sorte $\mathsf{Row}_{\mathcal{E}}\,\mathsf{Atom}$, la relation $r \lessdot r$ est vérifiée si et seulement si $\forall \xi_1, \xi_2 \in \mathcal{E}\ r(\xi_1) \leq r(\xi_2)$, *i.e.* r est constante.

Je termine cette présentation de la garde par deux résultats techniques qui me seront utiles aux sections 10.2.4 et 10.3.3 pour construire des solutions particulières de contraintes. La première permet d'obtenir des types de sorte arbitraire reliés par une inégalité ou une garde, à partir d'une inégalité entre atomes. Elle est utilisée dans la preuve du théorème 10.33 (page 192).

Propriété et définition 10.3 *Il existe une fonction, notée Λ, qui associe, à une sorte κ et un atome ℓ, un type brut $\Lambda_\kappa(\ell)$ de sorte κ, telle que : **(i)** si $\ell \leq_{\mathcal{L}} \ell'$ alors, pour toutes sortes κ, κ', on a $\Lambda_\kappa(\ell) \leq \Lambda_\kappa(\ell')$ et $\Lambda_\kappa(\ell) \lessdot \Lambda_{\kappa'}(\ell')$ et **(ii)** Λ_{Atom} est l'identité.* □

⌜ *Preuve.* Soit c un constructeur de type de signature $\kappa_1 \cdots \kappa_n$, tel que, pour tout $i \in [1, n]$, κ_i est de la forme $\mathsf{Row}_{\Xi_1 \cdots \Xi_{m_i}}\,\mathsf{Atom}$, pour $m_i \geq 0$. L'existence d'un tel constructeur a été supposée à la section 1.2.1 (page 22). Soit $G =$ guarded-l(c) $\cup$ guarded-r(c). Je définis la fonction Λ comme suit :

$$\begin{aligned}
\Lambda_{\mathsf{Atom}}(\ell) &= \ell \\
\Lambda_{\mathsf{Type}}(\ell) &= c\,\Lambda_{\kappa_1}(\ell_1) \cdots \Lambda_{\kappa_n}(\ell_n) \quad \text{où } \ell_i = \begin{cases} \ell & \text{si } i \in G \\ \bot & \text{sinon} \end{cases} \\
\Lambda_{\mathsf{Row}_\Xi\,\kappa}(\ell) &= \partial_\Xi \Lambda_\kappa(\ell)
\end{aligned}$$

Puisque tous les paramètres du constructeur de type c ont une sorte se terminant par un atome, cette définition par induction sur les sortes est bien fondée. Soient ℓ et ℓ' deux atomes tels que $\ell \leq_{\mathcal{L}} \ell'$ **(H$_1$)**. Je montre par induction sur la définition de Λ que $\Lambda_\kappa(\ell) \leq \Lambda_\kappa(\ell')$ **(C$_1$)** et $\Lambda_\kappa(\ell) \lessdot \ell'$ **(C$_2$)**.

◦ *Cas* $\kappa = \mathsf{Atom}$. Par définition, $\Lambda_{\mathsf{Atom}}(\ell) = \ell$ et $\Lambda_{\mathsf{Atom}}(\ell') = \ell'$. On en déduit que les buts (C$_1$) et (C$_2$) sont équivalents à l'hypothèse (H$_1$).

◦ *Cas* $\kappa = \mathsf{Type}$. Par définition, on a $\Lambda_{\mathsf{Type}}(\ell) = c\,\Lambda_{\kappa_1}(\ell_1)\cdots\Lambda_{\kappa_n}(\ell_n)$ **(1)** et $\Lambda_{\mathsf{Type}}(\ell') = c\,\Lambda_{\kappa_1}(\ell'_1)\cdots\Lambda_{\kappa_n}(\ell'_n)$ **(2)** avec, pour tout $i \in G$, $\ell_i = \ell$ **(3)** et $\ell'_i = \ell'$ **(4)**, et, pour tout $i \in [1,n]\backslash G$, $\ell_i = \ell'_i = \bot$ **(5)**. Grâce à (3), (4) et (H$_1$), en appliquant l'hypothèse d'induction, on a, pour tout $i \in G$, $\Lambda_{\kappa_i}(\ell_i) \leq \Lambda_{\kappa_i}(\ell'_i)$ **(6)** et $\Lambda_{\kappa_i}(\ell_i) \lessdot \ell'$ **(7)**. Puisque, pour tout $i \in G$, $c.i = +$, on déduit de (5) et (6) le but (C$_2$) : $c\,\Lambda_{\kappa_1}(\ell_1)\cdots\Lambda_{\kappa_n}(\ell_n) \leq c\,\Lambda_{\kappa_1}(\ell'_1)\cdots\Lambda_{\kappa_n}(\ell'_n)$. Enfin, (7) implique le but (C$_2$) : $c\,\Lambda_{\kappa_1}(\ell_1)\cdots\Lambda_{\kappa_n}(\ell_n) \lessdot \ell'$.

◦ *Cas* $\kappa = \mathsf{Row}_\Xi\,\kappa'$. Par définition, on a $\Lambda_\kappa(\ell) = \partial_\Xi\Lambda_{\kappa'}(\ell)$ **(1)** et $\Lambda_\kappa(\ell') = \partial_\Xi\Lambda_{\kappa'}(\ell')$ **(2)**. Par l'hypothèse d'induction, on a $\Lambda_{\kappa'}(\ell) \leq \Lambda_{\kappa'}(\ell')$ **(3)** et $\Lambda_{\kappa'}(\ell) \lessdot \ell'$ **(4)**. Grâce aux égalités (1) et (2), (3) et (4) impliquent respectivement les buts (C$_1$) et (C$_2$).

(C$_1$) forme notre premier but. De manière symétrique à (C$_2$), on peut montrer que si $\ell' \leq_{\mathcal{L}} \ell''$ alors $\ell' \lessdot \Lambda_{\kappa'}(\ell)$ **(C$_3$)**. Par transitivité avec (C$_2$) (propriété 10.2), on en déduit que, si $\ell \leq_{\mathcal{L}} \ell''$ alors $\Lambda_\kappa(\ell) \lessdot \Lambda_{\kappa'}(\ell'')$, ce qui est le deuxième but. ⌟

Cette deuxième propriété montre qu'une garde peut être traduite en une inégalité dès lors que la structure de ses membres est connue. Elle est utilisé pour montrer le théorème 10.52 (page 210)

Propriété et définition 10.4 *Soit T un squelette brut et t_o un type brut. Il existe un (unique) type brut dans T, noté* $\mathrm{lift}(t_o/T)$*, tel que, pour tout $t \in T$, $t_o \lessdot t \Leftrightarrow \mathrm{lift}(t_o/T) \leq t$. On a de plus, pour tous t, T_1 et T_2,* $\mathrm{lift}(t/T_1) \lessdot \mathrm{lift}(t/T_2)$. □

⌜ *Preuve.* Par la propriété 10.2, le premier résultat à prouver est équivalent à l'existence d'un plus petit élément (au sens de $\leq$) pour l'ensemble $\{\, t \in T \mid t_o \lessdot t \,\}$ **(1)** : s'il existe, ce minimum est précisément l'unique type $\mathrm{lift}(t_o/T)$ vérifiant l'équivalence $\forall t \in T\ \ t_o \lessdot t \Leftrightarrow \mathrm{lift}(t_o/T) \leq t$ **(2)**. Je m'intéresse tout d'abord au cas où t_o est un atome ℓ_o : je montre par induction sur la hauteur du squelette brut T que l'ensemble $\{\, t \in T \mid \ell_o \lessdot t \,\}$ **(3)** admet un plus petit élément, noté $\mathrm{lift}(\ell_o/T)$, **(P$_1$)**, qui vérifie $\mathrm{lift}(\ell_o/T) \lessdot \ell_o$ **(P$_2$)**.

◦ *Cas T est de sorte* Atom. Alors $T = \mathcal{L}$. L'ensemble (3) est $\{\, \ell \in \mathcal{L} \mid \ell_o \leq_{\mathcal{L}} \ell \,\}$. Il s'agit du cône supérieur de $\mathcal{L}$ de sommet ℓ_o. Cet ensemble admet un plus petit élément : ℓ_o. On en déduit les buts (P$_1$) et (P$_2$).

◦ *Cas T est de sorte* Type. Soit $\bot_T = c\,t_1^\bot\cdots t_n^\bot$ le plus petit élément du squelette brut T (propriété 1.3, page 23) ; on a $T = \{\, c\,t_1\cdots t_n \mid \forall i \in [1,n]\ \ t_i \approx^{c.i} t_i^\bot \,\}$ **(4)**. Soit $G = \text{guarded-l}(c)$ et $G' = [1,n]\backslash G$. Pour tout $i \in G$, on a $c.i = +$ et $\{\, t_i \mid t_i \approx t_i^\bot \,\}$ est un squelette brut T_i de hauteur strictement inférieure à celle de T. On peut donc lui appliquer l'hypothèse d'induction : $\{\, t_i \in T_i \mid \ell_o \lessdot t_i \,\}$ admet un plus petit élément, $\mathrm{lift}(\ell_o/T_i)$ **(5)**, tel que $\mathrm{lift}(\ell_o/T_i) \lessdot \ell_o$ **(6)**. Par (4), l'ensemble (3) peut s'écrire $\{\, c\,t_1\cdots t_n \mid \forall i \in G\ (t_i \in T_i \text{ et } \ell_o \lessdot t_i) \text{ et } \forall i \in G'\ t_i \approx^{c.i} t_i^\bot \,\}$. En utilisant (5), on en déduit qu'il admet un plus petit élément qui est $c\,t'_1\cdots t'_n$ avec, pour tout $i \in [1,n]$, $t'_i = \mathrm{lift}(\ell_o/T_i)$ **(7)** si $i \in G$ et $t'_i = t_i^\bot$ si $i \in G'$, d'où le but (P$_1$). En utilisant (6) et (7), par GRD-TYPE-LEFT, on obtient le but (P$_2$) : $c\,t'_1\cdots t'_n \lessdot \ell_o$.

◦ *Cas T est de sorte* $\mathsf{Row}_\Xi\,\kappa$. Il existe une famille quasi-constante indexée par Ξ de squelettes bruts (T_ξ), de hauteur strictement inférieure à celle de T, telle que $T = \{\, r \in \mathcal{T}_{\mathsf{Row}_\Xi\,\kappa} \mid r(\xi) \in T_\xi \,\}$. L'ensemble (3) peut donc s'écrire $\{\, r \in \mathcal{T}_{\mathsf{Row}_\Xi\,\kappa} \mid (\forall \xi \in \Xi\ \ r(\xi) \in T_\xi) \text{ et } \ell \lessdot r \,\}$. Par GRD-ROW-RIGHT, il est égal à $\{\, r \in \mathcal{T}_{\mathsf{Row}_\Xi\,\kappa} \mid \forall \xi \in \Xi\ \ (r(\xi) \in T_\xi \text{ et } \ell \lessdot r(\xi)) \,\}$ **(8)**. Par l'hypothèse d'induction, l'ensemble $\{\, t \in T_\xi \mid \ell_o \lessdot t \,\}$ admet un plus petit élément, noté $\mathrm{lift}(\ell_o/T_\xi)$ **(9)**, tel que $\mathrm{lift}(\ell_o/T_\xi) \lessdot \ell_o$ **(10)**. La famille (T_ξ) étant quasi-constante, on déduit de (9) que l'ensemble (8) admet un plus petit

élément : $\{\xi \mapsto \mathrm{lift}(\ell/T_\xi)\}_{\xi\in\Xi}$, d'où le but ($\mathrm{P}_1$). Par (10) et GRD-ROW-LEFT, on obtient le but (P_2) : $\{\xi \mapsto \mathrm{lift}(\ell/T_\xi)\}_{\xi\in\Xi} \lessdot \ell_o$.

Pour traiter le cas général où t_o a une sorte quelconque, je montre par induction sur sa structure qu'il existe un atome ℓ_o tel que, pour tout type t, $t_o \lessdot t$ soit équivalent à $\ell_o \lessdot t$ **($\mathbf{P_3}$)**

◦ *Cas t_o est un atome.* Le résultat (P_3) est immédiat en choisissant $\ell_o = t_o$.

◦ *Cas $t_o = c\,t_1 \cdots t_n$.* Soit t un type brut. Par GRD-TYPE-LEFT et la propriété 10.1 (page 172), $t_o \lessdot t$ équivalent à $\forall i \in \text{guarded-l}(c)\ \ t_i \lessdot t$ **(1)**. Soit $i \in \text{guarded-l}(c)$, par hypothèse d'induction, il existe ℓ_i tel que $t_i \lessdot t$ soit équivalent à $\ell_i \lessdot t$ **(2)**. Grâce à la propriété 10.2 (page 173), on en déduit que $t_o \lessdot t$ est équivalent à $\bigsqcup\{\,\ell_i \mid i \in \text{guarded-l}(c)\,\} \lessdot t$, ce qui permet d'obtenir (P_3) en posant $\ell_0 = \bigsqcup\{\,\ell_i \mid i \in \text{guarded-l}(c)\,\}$.

◦ *Cas t_o est une rangée brute.* Ce cas est analogue au précédent.

Les propriétés (P_1), (P_2) et (P_3) permettent de conclure : soit t_o un type de sorte arbitraire. Par (P_3), il existe un atome ℓ_o tel que $\{\,t \in T \mid t_o \lessdot t\,\} = \{\,t \in T \mid \ell_o \lessdot t\,\}$. Par ($\mathrm{P}_1$), cet ensemble admet un plus petit élément, $\mathrm{lift}(\ell_o/T)$, qui est, par la remarque initiale, le type $\mathrm{lift}(t_o/T)$ recherché, d'où le premier but. Il vérifie par définition $\ell_o \lessdot \mathrm{lift}(t_o/T)$ et, par (P_2), $\mathrm{lift}(t_o/T) \lessdot \ell_o$. Par la propriété 10.2 (page 173), on déduit de ces deux dernières relations le deuxième but : $\mathrm{lift}(t_o/T_1) \lessdot \mathrm{lift}(t_o/T_2)$. ⌟

10.1.2 Contraintes

L'algorithme de résolution décrit dans ce chapitre est formalisé comme un (ensemble de) système(s) de réécriture sur les contraintes. J'étais jusqu'à présent essentiellement intéressé par la *sémantique* des contraintes, et les propriétés logiques qu'elles permettent d'exprimer sur les variables de type et de programme. Par exemple, dans la définition du système MLIF($\mathcal{X}$), j'ai considéré les jugements modulo équivalence logique des contraintes (section 7.1.1, page 135). Dans ce chapitre, les contraintes jouent un double rôle : en plus d'être utilisées pour formuler les entrées et sorties de l'algorithme, elles servent à symboliser son état interne au cours de l'exécution. Cette deuxième lecture des contraintes donne de l'importance à leur *syntaxe*, qui doit permettre une représentation fidèle des structures de données mises en œuvre. Cela m'amène en particulier à introduire une nouvelle forme de prédicats, les multi-squelettes, qui, bien qu'exprimables à l'aide des autres constructions du langage, correspondent à une construction importante dans l'implémentation de ce solveur primaire.

Les types et contraintes considérés dans ce chapitre sont définis par la grammaire suivante :

$$
\begin{array}{lll}
\tau ::= \alpha \mid \ell \mid c\,\vec{\tau} \mid (\xi : \tau; \tau) & & \text{(type)} \\
\iota ::= \circ \mid \bullet & & \text{(marque)} \\
I ::= \langle \tau = \cdots = \tau \rangle^{\iota} \approx \cdots \approx \langle \tau = \cdots = \tau \rangle^{\iota} \mid \alpha \leq^{\iota} \alpha \mid \alpha \lessdot \alpha & & \text{(contrainte)} \\
\quad \mid \ \mathsf{true} \mid \mathsf{false} \mid I \wedge I \mid \exists \bar{\alpha}.I & & \\
\mathbb{I} ::= [\,] \mid \mathbb{I} \wedge I \mid \exists \bar{\alpha}.\mathbb{I} & & \text{(contexte)} \\
\mathbb{X} ::= [\,] \mid \exists \bar{\alpha}.\mathbb{X} & & \text{(contexte existentiel)}
\end{array}
$$

Les contraintes sont considérées modulo la commutativité et l'associativité de la conjonction. Les trois premières formes qui apparaissent dans la définition des contraintes ci-dessus correspondent aux prédicats de base du langage. Tout d'abord, la contrainte $\langle \tau_{1,1} = \cdots = \tau_{1,m_1} \rangle^{\iota_1} \approx \cdots \approx \langle \tau_{n,1} = \cdots = \tau_{n,m_n} \rangle^{\iota_n}$ est un ***multi-squelette***. Il s'agit d'un multi-ensemble non vide dont les éléments sont des multi-équations, chacune annotée par une marque booléenne ι. Chaque ***multi-équation*** $\tau_{i,1} = \cdots = \tau_{i,m_i}$ est elle-même un multi-ensemble non vide de types. Ce multi-squelette est bien formé si et seulement si tous les types $\tau_{i,j}$ qu'il contient ont la même sorte. Sa sémantique est alors définie par la règle suivante :

$$
\frac{\begin{array}{c}\forall i_1, i_2 \in [1,n]\ \ \forall j_1 \in [1,m_{i_1}]\ \ \forall j_2 \in [1,m_{i_2}]\ \ \varphi(\tau_{i_1,j_1}) \approx \varphi(\tau_{i_2,j_2}) \\ \forall i \in [1,n]\ \ \forall j_1, j_2 \in [1,m_i]\ \ \varphi(\tau_{i,j_1}) = \varphi(\tau_{i,j_2})\end{array}}{\varphi \vdash \langle \tau_{1,1} = \cdots = \tau_{1,m_1} \rangle^{\iota_1} \approx \cdots \approx \langle \tau_{n,1} = \cdots = \tau_{n,m_n} \rangle^{\iota_n}}
$$

La première prémisse contraint tous les types apparaissant dans le multi-squelette à appartenir au même squelette brut, *i.e.* la même classe d'équivalence de $\approx$. Par la deuxième prémisse, tous les types d'une multi-équation doivent avoir la même interprétation. Il faut noter que les marques portées par un multi-squelette n'ont pas d'incidence sur sa sémantique : elles sont utilisées seulement par la première étape de l'algorithme de résolution (section 10.2.1, page 179) pour enregistrer la réalisation de certaines opérations. Dans la suite de ce chapitre, j'utilise les meta-variables $\tilde{\bar{\tau}}$ et $\tilde{\bar{\alpha}}$ (resp. $\bar{\tau}$ et $\bar{\alpha}$) pour désigner des multi-squelettes (resp. multi-équations) de taille arbitraire, la deuxième étant réservée au cas où tous les types contenus sont des variables. De plus, si $\bar{\tau}$ est $\tau_1 = \cdots = \tau_n$ alors $\tau = \bar{\tau}$ et $\bar{\tau} = \tau$ dénotent la multi-équation obtenue en ajoutant une occurrence de τ à $\bar{\tau}$, c'est-à-dire $\tau_1 = \cdots = \tau_n = \tau$. De même, si $\tilde{\bar{\tau}}$ est $\langle\bar{\tau}_1\rangle^{\iota_1} \approx \cdots \approx \langle\bar{\tau}_n\rangle^{\iota_n}$ alors $\tilde{\bar{\tau}} \approx \langle\bar{\tau}\rangle^{\iota}$ et $\langle\bar{\tau}\rangle^{\iota} \approx \tilde{\bar{\tau}}$ désignent le multi-squelette $\langle\bar{\tau}_1\rangle^{\iota_1} \approx \cdots \approx \langle\bar{\tau}_n\rangle^{\iota_n} \approx \langle\bar{\tau}\rangle^{\iota}$. Enfin, j'écris de manière abrégée $\tau_1 \approx \tau_2$ pour le multi-squelette $\langle\tau_1\rangle^{\bullet} \approx \langle\tau_2\rangle^{\bullet}$ et $\tau_1 = \tau_2$ pour $\langle\tau_1 = \tau_2\rangle^{\bullet}$.

Les membres des inégalités doivent syntaxiquement être des variables, et non des types arbitraires. Toutefois, cela ne restreint pas l'expressivité du langage de contraintes puisque l'inégalité $\tau_1 \leq \tau_2$ peut être codée par $\exists\alpha_1\alpha_2.(\tau_1 = \alpha_1 \wedge \alpha_1 \leq \alpha_2 \wedge \alpha_2 = \tau_2)$, avec $\alpha_1\alpha_2 \# \mathrm{ftv}(\tau_1, \tau_2)$ et $\alpha_1 \neq \alpha_2$. Les inégalités portent également une marque, qui est utilisée d'une manière similaire à celles des multi-squelettes et n'a pas d'incidence sur leur interprétation. Dans la suite de ce chapitre, j'écris $\alpha_1 \leq \alpha_2$ pour $\alpha_1 \leq^{\circ} \alpha_2$. Comme les inégalités, les membres des contraintes gardes doivent également être des variables de type. Notons que, si les variables α_1 et α_2 sont de sorte Atom alors les contraintes $\alpha_1 \leq^{\iota} \alpha_2$ et $\alpha_1 \lessdot \alpha_2$ sont équivalentes.

La grammaire des types qui apparaissent dans ces contraintes est un sous-ensemble de celle donnée section 1.3 (page 24) qui n'inclut pas la forme $\partial_{\Xi}\tau$. Cependant, les contraintes générées par le système MLIF($\mathcal{X}$) n'utilisent cette construction que pour former des rangées dont les entrées sont des atomes. Dans ce cas, un codage simple est possible : si τ est de sorte Atom, la rangée $\partial_{\Xi}\tau$ peut être remplacée par une variable fraîche α de sorte $\mathsf{Row}_{\Xi}\,\mathsf{Atom}$ accompagnée de la contrainte $\tau \lessdot \alpha \wedge \alpha \lessdot \tau$. Le solveur que je présente est donc suffisant pour traiter les problèmes produits par MLIF($\mathcal{X}$). Le traitement de la forme $\partial_{\Xi}\tau$ de manière complète et native compliquant notablement la description de l'algorithme de résolution, j'ai préféré ne pas l'inclure dans mon développement.

Les types construits et les termes de rangée sont traités d'une manière similaire par une grande partie de l'algorithme : en effet, ils ne sont distingués que par les règles SPU-CLASH et SPU-MUTATE de l'algorithme d'unification (section 10.2.1, page 179). C'est la raison pour laquelle il est opportun d'introduire une notation commune pour ces deux notions. Un ***symbole*** est soit un constructeur de type c, soit un triplet formé d'une étiquette de rangée ξ, d'une partie co-finie Ξ de $\mathcal{E}$ telle que $\xi \notin \Xi$ et d'une sorte κ. Si le symbole d est le constructeur de type c de signature $\kappa_1 \cdots \kappa_n$ et si $\tau_1, \ldots, \tau_n$ sont des types de sortes respectives $\kappa_1, \ldots, \kappa_n$ alors $d\,\tau_1 \cdots \tau_n$ désigne le type $c\,\tau_1 \cdots \tau_n$ qui a la sorte Type. Si le symbole d est le triplet (ξ, Ξ, κ), et τ_1, τ_2 sont des types de sortes respectives κ et $\mathsf{Row}_{\Xi}\,\kappa$ alors $d\,\tau_1\tau_2$ désigne le type $(\xi : \tau_1; \tau_2)$ qui a la sorte $\mathsf{Row}_{\xi.\Xi}\,\kappa$. Les notions de signature, variances et positions gardées définies sur les constructeurs de types sont étendues aux symboles de manière naturelle : le symbole (ξ, Ξ, κ) a la signature $\kappa \cdot \mathsf{Row}_{\Xi}\,\kappa$, ses deux paramètres étant à la fois covariants (*i.e.* $(\xi, \Xi, \kappa).1 = (\xi, \Xi, \kappa).2 = +$) et gardés à gauche et à droite (*i.e.* $\text{guarded-l}(\xi, \Xi, \kappa) = \text{guarded-r}(\xi, \Xi, \kappa) = \{1, 2\}$). Un ***descripteur*** est un type de hauteur exactement 1, c'est-à-dire de la forme $d\,\vec{\alpha}$ ou ℓ.

Une opération de l'algorithme de résolution — la règle SPU-MUTATE de l'algorithme d'unification (section 10.2.1, page 179) — nécessite d'effectuer un choix *a priori* arbitraire entre deux étiquettes de rangées. De manière à diriger ce choix, et assurer la terminaison de l'algorithme, je suppose que l'ensemble $\mathcal{E}$ des étiquettes dispose d'une relation d'ordre total et bien fondée $\preceq$ (la relation d'ordre strict correspondante étant notée $\prec$). J'étends cette relation aux symboles par les deux règles suivantes :

$$c \preceq c \qquad \frac{\xi_1 \preceq \xi_2}{(\xi_1, \xi_2.\Xi, \kappa) \preceq (\xi_2, \xi_1.\Xi, \kappa)}$$

Ainsi étendue, $\preceq$ permet de comparer des symboles de mêmes signatures. Sur les constructeurs de type, $\preceq$ coïncide avec l'égalité.

Soit I une contrainte sans quantificateur existentiel. Je note $\tau_1 = \tau_2 \in I$ si et seulement si τ_1 et τ_2 apparaissent dans la même multi-équation de I, c'est-à-dire si I peut s'écrire sous la forme $\langle \tau_1 = \tau_2 = \bar{\bar{\tau}} \rangle^\iota \approx \tilde{\tilde{\tau}} \wedge I'$. Similairement $\tau_1 \approx \tau_2 \in I$ est valide si et seulement si τ_1 et τ_2 apparaissent dans le même multi-squelette de I, *i.e.* si $\tau_1 = \tau_2 \in I$ ou bien si I peut se mettre sous la forme $\langle \tau_1 = \bar{\bar{\tau}}_1 \rangle^{\iota_1} \approx \langle \tau_2 = \bar{\bar{\tau}}_2 \rangle^{\iota_2} \approx \tilde{\tilde{\tau}} \wedge I'$. De plus, j'écris $\alpha \approx \beta \in^* I$ (respectivement $* = \alpha \in \beta I$) si et seulement si il existe $\alpha_1, \ldots, \alpha_n$ tels que $\alpha = \alpha_1$, $\beta = \alpha_n$ et, pour tout $i \in [1, n-1]$, $\alpha_i \approx \alpha_{i+1} \in I$ (respectivement $\alpha_i = \alpha_{i+1} \in I$). Je note $\alpha_1 \leq \alpha_2 \in I$ (respectivement $\alpha_1 \lessdot \alpha_2 \in I$) si et seulement si I peut être écrite $\alpha_1 \leq^\iota \alpha_2 \wedge I'$ (respectivement $\alpha_1 \lessdot \alpha_2 \wedge I'$). Enfin, dans toute la suite de ce chapitre, chaque occurrence de $\approx^\nu$ doit être lue soit comme le symbole $\approx$ si la variance ν est $+$ ou $-$, soit comme le symbole $=$ si ν est $\pm$.

10.1.3 Représentants nommés et contraintes bien marquées

J'ai inclu les quantificateurs existentiels dans le langage des contraintes manipulées par le solveur primaire pour permettre à ce dernier d'introduire de nouvelles variables de types lors de la résolution. Comme je l'expliquerai dans la section 10.2.1 (page 179), ceux-ci n'ont pas d'incidence sur la représentation en machine des contraintes car ils ne peuvent apparaître qu'au sommet d'une forme normale pour l'algorithme de résolution. Toutefois, leur présence complique légèrement la définition de certaines propriétés relatives aux contraintes dont l'énoncé nécessite de disposer de noms distincts et globaux pour chaque variable de type, qu'elle soit libre ou liée. Pour contourner cette difficulté, j'introduis la notion de représentant nommé d'une contrainte.

Définition 10.5 (Représentant nommé) *Une contrainte I sans quantificateur existentiel est son propre représentant nommé, de support vide. Si I'_1 et I'_2 sont des représentants nommés de I_1 et I_2 de supports respectifs $\bar{\alpha}_1$ et $\bar{\alpha}_2$, et si $\bar{\alpha}_1 \# \mathrm{ftv}(I_2)$ et $\bar{\alpha}_2 \# \mathrm{ftv}(I_1)$, alors $I'_1 \wedge I'_2$ est un représentant nommé de $I_1 \wedge I_2$. Si I' est un représentant nommé de I de support $\bar{\alpha}$ alors I' est un représentant nommé de $\exists \bar{\beta}.I$ de support $\bar{\alpha} \cup \bar{\beta}$.* □

Un représentant nommé I' de support $\bar{\alpha}$ de la contrainte I est une contrainte sans quantificateur existentiel qui a la même structure que I. Intuitivement, elle est obtenue en retirant tous les quantificateurs existentiels de I, en prenant soin de choisir des noms frais pour chaque variable de type exhibée. Ces nouvelles variables de type forment le support $\bar{\alpha}$. La contrainte I n'admet en général pas de représentant nommé unique, puisque les noms introduits sont arbitraires. Ils sont toutefois identifiables modulo un renommage des variables de types. La propriété suivante relie de manière logique une contrainte à ses représentants nommés.

Propriété 10.6 *Si I' est un représentant nommé de I de support $\bar{\alpha}$, alors $\exists \bar{\alpha}.I' \equiv I$.* □

⌜ *Preuve.* Par induction sur la contrainte I.

∘ *Cas I est $p\,\vec{\tau}$.* Alors $\bar{\alpha} = \varnothing$ et $I' = I$. Le but est une tautologie.

∘ *Cas I est $I_1 \wedge I_2$.* On a $I' = I'_1 \wedge I'_2$ où I'_1 est un représentant nommé de I_1 de support $\bar{\alpha}_1$ **(1)**, I'_2 est un représentant nommé de I_2 de support $\bar{\alpha}_2$ **(2)**, $\bar{\alpha}_1 \# \mathrm{ftv}(I_2)$ **(3)**, $\bar{\alpha}_2 \# \mathrm{ftv}(I_1)$ **(4)** et $\bar{\alpha} = \bar{\alpha}_1 \cup \bar{\alpha}_2$ **(5)**. En appliquant l'hypothèse d'induction à (1) puis (2), on en déduit que $I_1 \equiv \exists \bar{\alpha}_1.I'_1$ et $I_2 \equiv \exists \bar{\alpha}_2.I'_2$. Ces équivalences impliquent $I_1 \wedge I_2 \equiv (\exists \bar{\alpha}_1.I'_1) \wedge (\exists \bar{\alpha}_2.I'_2)$. Par (3), (4), LOG-EX-AND et LOG-EX-EX, on en déduit le but : $I_1 \wedge I_2 \equiv \exists \bar{\alpha}_1 \bar{\alpha}_2.(I'_1 \wedge I'_2)$.

∘ *Cas I est $\exists \bar{\beta}.I_0$.* Alors I' est un représentant nommé de I_0 **(1)** dont le support $\bar{\alpha}_0$ vérifie $\bar{\alpha}_0 \cup \bar{\beta} = \bar{\alpha}$ **(2)**. En appliquant l'hypothèse d'induction à (1), on obtient $I_0 \equiv \exists \bar{\alpha}_0.I'$. Cette équivalence implique $\exists \bar{\beta}.I_0 \equiv \exists \bar{\beta}.\exists \bar{\alpha}_0.I'_0$. Par LOG-EX-EX et (2), cela donne le but recherché : $I \equiv \exists \bar{\alpha}.I_0$. ⌟

Je précise maintenant comment les marques ∘ et • portées par les multi-équations et les inégalités sont utilisées par l'algorithme de résolution pour enregistrer son avancement. J'énonce pour cela l'invariant préservé par le solveur qui leur est relatif. Il est exprimé par la notion de contrainte bien marquée.

Définition 10.7 (Contrainte bien marquée) *Une contrainte I est* **bien marquée** *si et seulement si elle contient la constante* false *ou bien si l'un de ses représentants nommés I' vérifie les propriétés suivantes :*

(i) *Si $\alpha_1 \leq^\circ \alpha_2$ est une inégalité de I' alors $\alpha_1 \approx \alpha_2 \in I'$.*

(ii) *Si $\langle \bar{\bar{\tau}} \rangle^\circ \approx \tilde{\tilde{\tau}}$ est une multi-équation de I' alors il existe $d\,\alpha_1 \cdots \alpha_n$ dans $\bar{\bar{\tau}}$ et $d'\,\beta_1 \cdots \beta_{n'}$ dans une multi-équation marquée • de $\tilde{\tilde{\tau}}$ tels que soit $d = d'$, $n = n'$ et, pour tout $i \in [1, n]$, $\alpha_i \approx^{d.i} \beta_i \in^* I'$, soit $d' \prec d$.*

(iii) *Toute variable libre dans I' est membre d'au moins une de ses multi-équations.*

On peut vérifier que cette propriété ne dépend pas du choix représentant nommé I'. Son premier point est relatif aux marques portées par les inégalités : une marque ∘ désigne une inégalité qui a déjà été considérée par la solveur pour générer une contrainte de squelette entre ses membres. Le deuxième point concerne les marques des multi-squelettes. Une multi-équation marquée ∘ doit contenir un descripteur $d\,\alpha_1 \cdots \alpha_n$. De plus, une multi-équation • du même multi-squelette doit mentionner un descripteur de forme incompatible avec d ou bien de la forme $d\,\beta_1 \cdots \beta_n$, avec les feuilles $\alpha_1, \ldots, \alpha_n$ et $\beta_1, \ldots, \beta_n$ reliées deux à deux par des multi-squelettes. Si cet invariant est préservé par l'algorithme de résolution, il doit toutefois être respecté par les contraintes données en entrée au solveur primaire. Cela peut être réalisé grâce aux deux propriétés suivantes.

Propriété 10.8 *Si toutes les marques portées par I sont • et toutes ses variables, libres ou liées, sont membre d'une multi-équation alors I est bien marquée.* □

Propriété 10.9 *Si I_1 et I_2 sont bien marquées alors $I_1 \wedge I_2$ est bien marquée.* □

La première donne un procédé pour introduire des marques dans les contraintes initiales — telles que celles produites par l'algorithme de génération du système MLIF($\mathcal{X}$) — de manière à obtenir des contraintes bien marquées : il suffit d'utiliser systématiquement la marque • qui indique qu'aucun travail n'a encore été effectué. L'intérêt de la deuxième propriété apparaîtra avec la description de l'algorithme de résolution complet : certaines étapes de ce dernier nécessitent en effet de former la conjonction de deux contraintes obtenues en sortie du solveur primaire. La propriété 10.9 montre qu'il n'est pas utile de réinitialiser toutes les marques à • après cette opération (c'est-à-dire de perdre toute trace du travail effectué sur chaque membre), mais que les marques existantes peuvent être conservées.

Pour parachever cette présentation des contraintes, il me faudrait expliquer comment celles-ci peuvent être représentées en machine pour implémenter l'algorithme de manière efficace. Cependant, la représentation que je propose est étroitement liée à la première étape de l'algorithme de résolution, puisque seules ses formes normales seront effectivement représentables. C'est pourquoi je préfère commencer par sa description.

10.2 Résolution des contraintes

La spécification des deux premières étapes de l'algorithme de résolution est donnée par deux jeux de règles de réécriture entre contraintes, donnés figures 10.2 et 10.3, qui sont l'objet des sous-sections suivantes. Dans les deux cas, la réécriture est effectuée modulo α-conversion et sous un contexte arbitraire. La spécification de l'algorithme n'est pas déterministe : plusieurs instances de règles peuvent être simultanément applicables. Cependant, je montre dans ce qui suit que toutes les stratégies possibles sont correctes et terminent.

$$
\begin{array}{rcll}
(\exists\bar\alpha.I_1)\wedge I_2 & \rightarrow_u & \exists\bar\alpha.(I_1\wedge I_2) & \textsc{spu-ex-and}\\
 & & \textit{si } \bar\alpha \# \mathrm{ftv}(I_2) & \\
\langle\bar{\bar\tau} = d\,\vec\tau_1\tau\vec\tau_2\rangle^\iota \approx \tilde{\tilde\tau} & \rightarrow_u & \exists\alpha.(\alpha=\tau\wedge\langle\bar{\bar\tau} = d\,\vec\tau_1\alpha\vec\tau_2\rangle^\iota\approx\tilde{\tilde\tau}) & \textsc{spu-name}\\
 & & \textit{si } \alpha\notin\mathrm{ftv}(\bar{\bar\tau},\vec\tau_1,\tau,\vec\tau_2,\tilde{\tilde\tau}) & \\
 & & \textit{et } \tau \textit{ n'est pas une variable} & \\
\alpha_1\leq^\bullet\alpha_2 & \rightarrow_u & \alpha_1\leq^\circ\alpha_2\wedge\alpha_1\approx\alpha_2 & \textsc{spu-leq}\\
\tilde{\tilde\tau}_1\wedge\tilde{\tilde\tau}_2 & \rightarrow_u & \tilde{\tilde\tau}_1\approx\tilde{\tilde\tau}_2 & \textsc{spu-fuse-atom}\\
 & & \textit{si } \tilde{\tilde\tau}_1 \textit{ et } \tilde{\tilde\tau}_2 \textit{ de sorte } \mathsf{Row}_{\Xi_1\cdots\Xi_n}\,\mathsf{Atom} & \\
\langle\alpha=\bar{\bar\tau}_1\rangle^{\iota_1}\approx\tilde{\tilde\tau}_1\wedge\langle\alpha=\bar{\bar\tau}_2\rangle^{\iota_2}\approx\tilde{\tilde\tau}_2 & \rightarrow_u & \langle\alpha=\bar{\bar\tau}_1=\bar{\bar\tau}_2\rangle^{\iota_1\vee\iota_2}\approx\tilde{\tilde\tau}_1\approx\tilde{\tilde\tau}_2 & \textsc{spu-fuse-}\approx\\
\langle\alpha=\bar{\bar\tau}_1\rangle^{\iota_1}\approx\langle\alpha=\bar{\bar\tau}_2\rangle^{\iota_2}\approx\tilde{\tilde\tau} & \rightarrow_u & \langle\alpha=\bar{\bar\tau}_1=\bar{\bar\tau}_2\rangle^{\iota_1\vee\iota_2}\approx\tilde{\tilde\tau} & \textsc{spu-fuse-}=\\
\langle\bar{\bar\alpha}=d\,\vec\alpha\rangle^\bullet\approx\langle\bar{\bar\beta}=d\,\vec\beta\rangle^\bullet\approx\tilde{\tilde\tau} & \rightarrow_u & \begin{pmatrix}\langle\bar{\bar\alpha}=d\,\vec\alpha\rangle^\bullet\approx\langle\bar{\bar\beta}=d\,\vec\beta\rangle^\circ\approx\tilde{\tilde\tau}\\ \wedge\,(\bigwedge_{d.i\in\{+,-\}}\vec\alpha_{|i}\approx\vec\beta_{|i})\\ \wedge\,(\bigwedge_{d.i=\pm}\vec\alpha_{|i}=\vec\beta_{|i})\end{pmatrix} & \textsc{spu-dec-}\approx\\
\langle\bar{\bar\alpha}=d\,\vec\alpha=d\,\vec\beta\rangle^\iota\approx\tilde{\tilde\tau} & \rightarrow_u & \langle\bar{\bar\alpha}=d\,\vec\alpha\rangle^\iota\approx\tilde{\tilde\tau}\wedge(\bigwedge_{i\in[1,n]}\vec\alpha_{|i}=\vec\beta_{|i}) & \textsc{spu-dec-}=\\
\tilde{\tilde\tau} & \rightarrow_u & \mathsf{false} & \textsc{spu-clash}\\
 & & \textit{si } c\,\vec\tau \textit{ et } c'\,\vec\tau' \textit{ dans } \tilde{\tilde\tau} \textit{ avec } c\neq c' & \\
\tilde{\tilde\tau}\approx\langle\bar{\bar\tau}=(\xi_1:\tau_1;\tau')\rangle^\iota & \rightarrow_u & \exists\alpha\alpha_2.\begin{pmatrix}\tilde{\tilde\tau}\approx\langle\bar{\bar\tau}=(\xi_2:\alpha_2;\xi_1:\tau_1;\alpha)\rangle^\bullet\\ \wedge\,\tau'=(\xi_2:\alpha_2;\alpha)\end{pmatrix} & \textsc{spu-mutate}\\
 & & \textit{si } \xi_2\prec\xi_1 \textit{ et } \bar{\bar\tau} \textit{ ou } \tilde{\tilde\tau} \textit{ contient un type de} & \\
 & & \textit{la forme } (\xi_2:\star;\star) \textit{ et } \alpha\alpha_2\#\mathrm{ftv}(\tilde{\tilde\tau},\bar{\bar\tau},\tau_1,\tau') & \\
\langle\bar{\bar\tau}=\ell=\ell\rangle^\iota\approx\tilde{\tilde\tau} & \rightarrow_u & \langle\bar{\bar\tau}=\ell\rangle^\iota\approx\tilde{\tilde\tau} & \textsc{spu-atom-eq}\\
\langle\bar{\bar\tau}=\ell_1=\ell_2\rangle^\iota\approx\tilde{\tilde\tau} & \rightarrow_u & \mathsf{false} & \textsc{spu-atom-neq}\\
 & & \textit{si } \ell_1\neq\ell_2 & \\
\mathbb{I}[\mathsf{false}] & \rightarrow_u & \mathsf{false} & \textsc{spu-fail}\\
 & & \textit{si } \mathbb{I}\neq[\,] &
\end{array}
$$

Figure 10.2 – Règles d'unification

10.2.1 Unification

La première étape de l'algorithme, décrite par les règles de la figure 10.2, réalise deux processus d'unification de manière simultanée. L'un opère sur les multi-squelettes et l'autre sur les multi-équations. Ils sont tous les deux dérivés de l'algorithme d'unification dans les théories équationnelles de Rémy [Rém92, PR03] ; cependant, du fait de leur superposition, certains détails techniques diffèrent. Le but de cet algorithme est d'obtenir soit la contrainte false, soit une contrainte unifiée vérifiant les six critères de la définition suivante.

Définition 10.10 (Contrainte unifiée) *Une contrainte I est* **unifiée** *si et seulement l'un de ses représentants nommés I' vérifie les propriétés suivantes :*

(i) *Tous les types apparaissant dans les multi-équations I' sont des petits types.*

(ii) *Si $\tau_1\leq\tau_2\in I'$ alors $\tau_1\approx\tau_2\in I'$.*

(iii) *Chaque variable de type est membre d'au plus une multi-équation de I'.*

(iv) *Chaque multi-équation de I' contient au plus un type qui n'est pas une variable.*

(v) *Si $d\,\tau_1\cdots\tau_n\approx d'\,\tau'_1\cdots\tau'_{n'}\in I'$ alors $d=d'$, $n=n'$ et, pour tout $i\in[1,n]$, $\tau_i\approx^{d.i}\tau'_i\in I'$.*

(vi) *I' comporte au plus un multi-squelette de chaque sorte $\mathsf{Row}_{\Xi_1\cdots\Xi_n}\,\mathsf{Atom}$ (où $n\geq 0$).*

Le point (i) assure que le partage a été rendu totalement explicite dans la contrainte en introduisant un nom (*i.e.* une variable de type) pour chaque nœud de type. Par le point (ii), toute

l'information de structure donnée par les inégalités est, dans une contrainte unifiée, traduite dans les multi-squelettes. Les points (iii) à (v) concernent les multi-squelettes. L'algorithme d'unification maintient, grâce aux multi-équations, des classes d'équivalences (c'est-à-dire des ensembles disjoints) de variables (point (iii)) ; et à chaque classe peut être associé un descripteur (point (iv)). De plus, les différents descripteurs apparaissant dans un multi-squelette doivent être compatibles et leur feuilles récursivement reliées par des contraintes de squelettes ou d'égalité (point (v)). Enfin, puisque la relation $\approx$ est toujours satisfaite entre des types bruts de sorte $\mathsf{Row}_{\Xi_1 \cdots \Xi_n}\ \mathsf{Atom}$, on peut toujours fusionner deux multi-squelettes de cette sorte, et ainsi exiger qu'une contrainte unifiée en contienne au plus un (point (vi)).

Je commente maintenant les règles de la figure 10.2, et explique comment elles permettent d'obtenir une contrainte unifiée. La règle SPU-EX-AND est une version dirigée de LOG-EX-AND, dont l'effet est de faire remonter toutes les quantifications existentielles à la racine de la contrainte. Les multi-squelettes, inégalités et gardes qu'elle contient sont alors membres d'une seule conjonction, ce qui rend un nombre maximal d'instances des règles suivantes applicables. La condition d'application de SPU-EX-AND prévient toute capture de nom de variable. La règle SPU-NAME introduit un nom α pour un sous-terme strict τ qui n'est pas une variable d'un type apparaissant dans un multi-squelette. Une contrainte normale pour cette règle ne contient que des petits types, *i.e.* vérifie le point (i) de la définition 10.10. Cette règle permet également de restreindre les règles suivantes à des multi-squelettes mentionnant des petits types, et ainsi de prévenir toute duplication de structure lors de leur application.

La règle SPU-LEQ vise à satisfaire le point (ii). Elle reflète l'inclusion de $\leq$ dans $\approx$ en générant une contrainte de squelette à partir de chaque inégalité marquée $\bullet$. Comme effet de bord, elle modifie la marque portée par l'inégalité, de manière à empêcher des applications répétées de la règle avec la même prémisse. La règle SPU-FUSE-ATOM fusionne deux multi-squelettes de la même sorte $\mathsf{Row}_{\Xi_1 \cdots \Xi_n}\ \mathsf{Atom}$, comme demandé par le point (vi). Les règles SPU-FUSE-$=$ et SPU-FUSE-$\approx$ identifient respectivement deux multi-équations (et par là même, pour SPU-FUSE-$\approx$, deux multi-squelettes) qui ont une variable de type commune. La marque de la nouvelle multi-équation est $\iota_1 \vee \iota_2$, qui vaut $\circ$ si ι_1 et ι_2 sont égales à $\circ$, et $\bullet$ sinon. Ces deux règles permettent ainsi d'obtenir la propriété (iii) de la définition 10.10. Cependant, la nouvelle contrainte ne vérifie plus nécessairement les points (iv) et (v), même si cela était le cas de la contrainte originale. Le but des règles de SPU-CLASH à SPU-ATOM-NEQ est de traiter ces situations.

Tout d'abord, la règle SPU-DEC-$=$ décompose une équation entre deux types dont les symboles de têtes sont identiques. Elle produit une conjonction d'égalités entre leurs sous-termes. Seul un des deux types est conservé dans la multi-équation originelle, qui peut ainsi, lorsqu'elle est normale pour cette règle, satisfaire le point (iv). La règle SPU-DEC-$\approx$ effectue un travail similaire au niveau des multi-squelettes, en décomposant une contrainte $\approx$ entre deux types ayant le même symbole de tête. Elle produit une conjonction de contraintes de squelette (respectivement d'égalités) entre leurs sous-termes situées en positions covariantes ou contravariantes (respectivement invariantes). À la différence de la règle SPU-DEC-$=$, les deux types sont conservés dans le multi-squelette, la règle ne serait sinon pas correcte puisqu'elle pourrait perdre des relations d'égalités dans l'une ou l'autre des multi-équations. Cependant, de manière à empêcher toute application répétée de SPU-DEC-$\approx$ sur la même prémisse, l'algorithme enregistre l'opération de décomposition en modifiant la marque portée par l'une des multi-équations concernées. Enfin, les règles SPU-CLASH et SPU-MUTATE complètent SPU-DEC-$=$ et SPU-DEC-$\approx$, en s'appliquant dans les situations où un multi-squelette contient deux types ayant des symboles de tête différents. La règle SPU-CLASH traite le cas d'un multi-squelette de sorte Type : deux types ayant des constructeurs différents étant systématiquement incompatibles, une erreur est signalée. Le cas des multi-squelettes de rangées est différent, puisque deux termes de rangées ayant des étiquettes de tête différentes peuvent néanmoins avoir des interprétation égales. Il est traité par la règle SPU-MUTATE qui permute les deux étiquettes dans un des deux termes, de manière à faire apparaître la plus petite pour $\preceq$ en tête. SPU-ATOM-EQ et SPU-ATOM-

NEQ s'appliquent aux multi-équations contenant deux constantes atomiques, respectivement égales et différentes. Enfin, la règle SPU-FAIL fait remonter une erreur à travers un contexte (non vide) arbitraire.

Je m'intéresse dans la suite de cette section à la preuve du processus d'unification, qui est énoncée dans le théorème 10.15 (page 183). Pour obtenir ce résultat, je donne une série de quatre lemmes : les deux premiers sont des résultats de stabilité, qui montrent que chaque étape de réduction pour $\rightarrowtriangle_u$ préserve le caractère bien marqué et la sémantique des contraintes. Le troisième montre la terminaison de l'algorithme, quelle que soit la stratégie choisie, et enfin, le dernier caractérise la forme des résultats de l'algorithme d'unification, *i.e.* des formes normales pour $\rightarrowtriangle_u$.

Lemme 10.11 (Marques) *La réduction $\rightarrowtriangle_u$ préserve le caractère bien marqué des contraintes.* □

Lemme 10.12 (Correction) *La réduction $\rightarrowtriangle_u$ préserve la sémantique des contraintes.* □

⌜ *Preuve.* J'examine successivement les différentes règles.

- *Cas* SPU-EX-AND. Par LOG-EX-AND.
- *Cas* SPU-NAME. Par LOG-NAME-EQ.
- *Cas* SPU-LEQ. Par l'inclusion de $\leq$ dans $\approx$.
- *Cas* SPU-FUSE-ATOM. Par la propriété 1.4 (page 23).
- *Cas* SPU-FUSE-$\approx$. Par la transitivité de $\approx$ (propriété 1.3, page 23).
- *Cas* SPU-FUSE-$=$. Par la transitivité de $=$.
- *Cas* SPU-DEC-$\approx$. Par les règles SSK-TYPE et SSK-ROW (figure 1.3, page 24).
- *Cas* SPU-DEC-$=$, SPU-CLASH, SPU-ATOM-EQ *et* SPU-ATOM-NEQ. Par la définition de l'égalité dans un modèle syntaxique libre.
- *Cas* SPU-MUTATE. On veut montrer que $\tilde{\tilde{\tau}} \approx \langle \bar{\bar{\tau}} = (\xi_1 : \tau_1; \tau') \rangle^\iota \equiv \tilde{\tilde{\tau}} \approx \langle \bar{\bar{\tau}} = (\xi_2 : \alpha_2; \xi_1 : \tau_1; \alpha) \rangle^\bullet \wedge \tau' = (\xi_2 : \alpha_2; \alpha)$ où $\alpha\alpha_2 \# \mathrm{ftv}(\tilde{\tilde{\tau}}, \bar{\bar{\tau}}, \tau_1, \tau')$. Grâce à cette hypothèse de fraîcheur, puisque $\tau' = (\xi_2 : \alpha_2; \alpha)$ détermine α et α_2, il suffit de vérifier que, dans le modèle des types bruts, $t' = (\xi_2 : t_2; t)$ **(1)** implique $(\xi_1 : t_1; t') = (\xi_2 : t_2; \xi_1 : t_1; t)$ **(2)**. Par (1), on a $(\xi_1 : t_1; t') = (\xi_1 : t_1; \xi_2 : t_2; t)$. Par commutativité des étiquettes dans les termes de rangée, le membre droit de cette égalité est égal à $(\xi_2 : t_2; \xi_1 : t_1; t)$. On en déduit le but (2).
- *Cas* SPU-FAIL. Par LOG-FALSE. ⌟

Lemme 10.13 (Terminaison) *Le système de réécriture $\rightarrowtriangle_u$ est fortement normalisant.* □

⌜ *Preuve.* Une étiquette est dite ***apparente*** dans une contrainte si et seulement si elle apparaît dans au moins l'un de ses types. On vérifie, par inspection, que toutes les règles de la figure 10.2 (page 179) préservent ou réduisent l'ensemble des étiquettes apparentes de la contrainte manipulée. Je mesure un terme de rangée $(\xi : \tau_1; \tau_2)$ par la paire (Ξ, ξ) où Ξ est le codomaine de τ_2 (*i.e.* τ_2 est de sorte $\mathsf{Row}_\Xi\, \kappa$ pour une certaine κ). Ces poids sont ordonnés par le produit lexicographique des ordres $\supseteq$ et $\preceq$, lequel n'admet pas de chaîne strictement décroissante infinie sous l'hypothèse d'un ensemble fini d'étiquettes apparentes.

Je mesure une contrainte par les quantités suivantes, ordonnées lexicographiquement :

(1) Le nombre d'inégalités portant la marque $\bullet$.

(2) Le multi-ensemble des poids des termes de rangée.

(3) Le multi-ensemble des hauteurs des types, en incluant leurs sous-termes.

(4) Le nombre de multi-équations portant la marque $\bullet$.

(5) Le nombre de multi-équations.

(6) Le nombre de multi-squelettes.

(7) Le nombre de quantificateurs existentiels et de conjonctions.

(8) Le multi-ensemble des profondeurs des quantificateurs existentiels.

Je montre que chaque règle de la figure 10.2 (page 179) diminue strictement cette mesure. La règle SPU-EX-AND permute une quantification existentielle et une conjonction : elle réduit donc (8) et laisse (7) inchangé. De plus elle ne modifie pas les prédicats apparaissant dans le contrainte, préservant donc les quantités (1) à (6). La règle SPU-NAME remplace un sous-terme strict τ d'un type par une variable α et introduit une équation $\alpha = \tau$. On en déduit qu'elle préserve (1) et (2) et diminue (3), puisque le type τ, qui n'est pas de hauteur nulle, est compté une fois dans la contrainte obtenue, au lieu de deux dans la contrainte initiale. La règle SPU-LEQ change la marque d'une inégalité de $\bullet$ en $\circ$, elle diminue donc (1). Toutes les autres règles sauf SPU-FAIL n'affectent pas les inégalités, elles préservent donc cette quantité. Les règles SPU-FUSE-ATOM, SPU-FUSE-$\approx$ et SPU-FUSE-$=$ ne modifient aucune occurrence d'un type non variable dans la contrainte, elles préservent donc (2) et (3). SPU-FUSE-ATOM ne modifie pas les multi-équations de la contrainte, laissant donc (4) et (5) inchangées ; mais elle diminue (6) en fusionnant deux multi-squelettes. SPU-FUSE-$\approx$ et SPU-FUSE-$=$ diminuent (4), si au moins une des multi-équations fusionnées est marquée $\bullet$, et dans tous les cas diminuent (5). La règle SPU-DEC-$\approx$ introduit des multi-équations entre variables seulement, elle n'affecte donc pas les quantités (2) et (3). De plus, elle change la marque d'une multi-équation de $\bullet$ vers $\circ$, réduisant ainsi (4). La règle SPU-DEC-$=$ élimine un terme de rangée ou un type construit. Dans le premier cas, elle diminue (2), dans le second (3). La règle SPU-MUTATE réduit (2) : si le terme de rangée $(\xi_1 : \tau_1; \tau')$ a le poids (Ξ, ξ_1) alors les termes de rangées $(\xi_2 : \alpha_2; \xi_1 : \tau_1; \alpha)$, $(\xi_1 : \tau_1; \alpha)$ et $(\xi_2 : \tau_2; \alpha)$ ont pour poids respectifs (Ξ, ξ_2), $(\xi_2.\Xi, \xi_1)$ et $(\xi_1.\Xi, \xi_2)$, lesquels sont strictement inférieurs au précédent grâce à la condition $\xi_2 \prec \xi_1$. La règle SPU-ATOM-EQ élimine une occurrence d'un atome, elle laisse donc (2) inchangé, mais diminue (3). La règle SPU-ATOM-NEQ élimine un multi-squelette de sorte Atom contenant une constante atomique, diminuant ainsi (3) sans affecter (2). Enfin, la règle SPU-FAIL élimine un contexte non vide, faisant décroître chaque quantité de (1) à (6) au sens large et (7) au sens strict.

Puisque l'ordre sur ces mesures n'admet aucune chaîne strictement décroissante infinie sous l'hypothèse d'un ensemble finiment borné d'étiquettes apparentes (comme produit lexicographique d'ordres vérifiant la même propriété), et puisque toutes les règles de la figure 10.2 (page 179) préservent ou réduisent l'ensemble des étiquettes apparentes, on en déduit qu'elles n'admettent pas de réduction infinie. ⌟

Lemme 10.14 (Formes normales) *Si I est une contrainte bien marquée et normale pour les règles de la figure 10.2 (page 179) alors elle est soit unifiée soit égale à* false. □

⌜ *Preuve.* Supposons que I est une forme normale pour les règles de la figure 10.2 (page 179) qui n'est pas false. Puisque la contrainte I ne peut être réduite par SPU-EX-AND, elle est de la forme $\mathbb{X}[I']$, où I' est une contrainte sans quantificateur existentiel et normale pour $\rightarrowtriangle_u$. De plus I' ne contient pas la constante false, car I n'est pas false et ne peut être réduite par SPU-FAIL. On en déduit que I' vérifie les propriétés (i), (ii) et (iii) des contraintes bien marquées. Montrons maintenant qu'elle vérifie les six propriétés d'une contrainte unifiée :

(i) Puisque I' ne peut être réduite par la règle SPU-NAME, les types qui apparaissent dans ses multi-équations sont petits.

(ii) Puisque I' ne peut être réduite par la règle SPU-LEQ, toutes ses inégalités sont marquées $\circ$. Or, I' étant bien marquée, on en déduit que si $\alpha_1 \leq \alpha_2 \in I'$ alors $\alpha_1 \approx \alpha_2 \in I'$.

(iii) La contrainte I' étant normale pour SPU-FUSE-$\approx$, chaque variable ne peut être membre que d'un multi-squelette. De plus, puisque SPU-FUSE-$=$ ne s'applique pas, une variable ne peut être membre de deux multi-équations différentes d'un même multi-squelette. On en déduit que chaque variable de type est dans au plus une multi-équation.

(iv) Je raisonne par l'absurde en supposant qu'une multi-équation de I' contienne deux types τ et τ' non variables. Je raisonne par cas suivant sa sorte. Si elle est de sorte Atom alors τ et τ' sont des constantes atomiques et l'une des règles SPU-ATOM-EQ et SPU-ATOM-NEQ s'applique. Si elle est de sorte Type alors τ et τ' sont respectivement de la forme $c\,\vec{\tau}$ et $c'\,\vec{\tau}'$. Si $c = c'$ alors la règle SPU-DEC-= s'applique et sinon SPU-CLASH s'applique. Enfin, si elle est de sorte $\mathsf{Row}_\Xi\,\kappa$ alors τ et τ' sont respectivement de la forme $(\xi : \tau_1; \tau_2)$ et $(\xi' : \tau'_1; \tau'_2)$. Si $\xi = \xi'$ alors I' peut être réduite par SPU-DEC-=. Sinon, on a soit $\xi \prec \xi'$ soit $\xi' \prec \xi$ et SPU-MUTATE peut s'appliquer. Tous les cas aboutissant à une contradiction de l'irréductibilité de I', on en déduit que l'hypothèse initiale est fausse, et donc que chaque multi-équation de I' contient au plus un type non variable.

(v) Je suppose $d\,\alpha_1 \cdots \alpha_n \approx d'\,\alpha'_1 \cdots \alpha'_{n'} \in I'$. Puisque I' ne peut être réduite ni par SPU-CLASH ni par SPU-MUTATE, on a $d = d'$ et, par là même, $n = n'$. Puisque la contrainte I est bien marquée, il existe des types $d\,\beta_1 \cdots \beta_n$ et $d\,\beta'_1 \cdots \beta'_n$ dans des multi-équations marquées $\bullet$ tels que pour tout $i \in [1, n]$, $\alpha_i \approx^{d.i} \beta_i \in^* I'$ et $\alpha'_i \approx^{d.i} \beta'_i \in^* I'$. Puisque la règle SPU-DEC-$\approx$ n'est pas applicable, on en déduit que les deux types $d\,\beta_1 \cdots \beta_n$ et $d\,\beta'_1 \cdots \beta'_n$ sont dans la même multi-équation. Par le point (iv), on en déduit qu'ils sont égaux. Par transitivité, il en résulte que, pour tout $i \in [1, n]$, $\alpha_i \approx^{d.i} \alpha'_i \in^* I'$. La contrainte I' étant normale pour SPU-FUSE-$\approx$ et SPU-FUSE-=, cela implique $\alpha_i \approx^{d.i} \alpha'_i \in I'$.

(vi) Puisque I' ne peut être réduite par SPU-FUSE-ATOM, elle ne peut contenir deux multi-squelettes de sorte $\mathsf{Row}_{\Xi_1 \cdots \Xi_n}\,\mathsf{Atom}$. ⌟

Ces quatre lemmes permettent d'obtenir le théorème suivant, qui établit que la relation $\rightarrowtriangle_u$ donne un algorithme réécrivant une contrainte bien marquée en une contrainte unifiée ou trivialement fausse.

Théorème 10.15 (Unification) *Soit I une contrainte bien marquée. Alors $\rightarrowtriangle_u$ termine sur I et, si $I \rightarrowtriangle_u^{**} I'$, alors I' est soit* false *soit bien marquée et unifiée.* □

Dans une implémentation du solveur, il est possible de réaliser l'unification directement lors de la construction des contraintes primaires. Cela permet tout d'abord de détecter les erreurs d'unification immédiatement, et ainsi de les signaler en utilisant les même techniques que celles employées dans les systèmes à base d'unification pure. De plus, les structures de données utilisées peuvent être restreintes à la représentation de contraintes *unifiées*. Dans notre formalisation, une contrainte unifiée représente alors un état de l'algorithme de résolution, tandis qu'une contrainte non unifiée peut être vue comme la superposition d'une part d'un état et d'autre part d'information de contrôle. En quelques mots, chaque multi-équation et chaque multi-squelette peut être représenté en mémoire par une structure classique de recherche-union [Tar75]. Chaque multi-équation doit porter un pointeur sur son multi-squelette de manière à pouvoir y accéder en temps constant, par exemple pour effectuer SPU-FUSE-=, ainsi que des pointeurs sur les fils de son descripteur, le cas échéant. Nous verrons lors de l'étape suivante qu'il est également utile de stocker dans chaque multi-squelette une liste de pointeurs vers ses multi-équations. Enfin, les inégalités et les gardes peuvent être représentés par des pointeurs bi-directionnels entre multi-équations.

10.2.2 Test d'occurrence

Le modèle dans lequel les types sont interprétés est syntaxique : il ne comporte pas de types récursifs. Ainsi, une contrainte dont les multi-squelettes décrivent une structure cyclique n'est pas satisfiable. L'absence de tels cycles peut être détectée par une procédure usuellement appelée *test d'occurrence* qui consiste à effectuer le tri topologique d'un graphe dont les nœuds sont les multi-squelettes et les arcs formés par leurs descripteurs, comme précisé par les définitions suivantes.

Définition 10.16 (Domination) *Soit I une contrainte sans quantificateur existentiel. β est* **dominé** *par α dans I (noté : $\beta \prec_I \alpha$) si et seulement si il existe $d\,\vec{\beta}$ tel que $\alpha \approx d\,\vec{\beta} \in I$ et $\beta \in \vec{\beta}$.* □

Définition 10.17 (Test d'occurrence) *La contrainte I est* **cyclique** *si et seulement si, étant donné un de ses représentants nommés I', le graphe de $\prec_{I'}$ comporte un cycle. (Cette propriété ne dépend pas du choix du représentant I'.) Une contrainte non cyclique est* **acyclique** *; on dit qu'elle vérifie le test d'occurrence.* □

Je montre maintenant qu'une contrainte qui ne vérifie pas le test d'occurrence n'est pas satisfiable. Cette preuve repose sur la notion de hauteur d'un type brut définie à la section 1.2 (page 21).

Lemme 10.18 *Soit I une contrainte sans quantificateur existentiel. Si $\alpha_1 \prec_I \alpha_2$ alors, pour toute solution φ de I, $\mathrm{h}(\varphi(\alpha_1)) < \mathrm{h}(\varphi(\alpha_2))$ ou bien α_1 et α_2 sont respectivement de sortes $\mathsf{Row}_{\Xi_1}\,\kappa$ et $\mathsf{Row}_{\Xi_2}\,\kappa$ avec $\Xi_1 \subset \Xi_2$, et $\mathrm{h}(\varphi(\alpha_1)) = \mathrm{h}(\varphi(\alpha_2))$.* □

⌜ *Preuve.* Supposons $\alpha_1 \prec_I \alpha_2$. Trois cas sont envisageables.

∘ *Cas $\alpha_2 \approx c\,\vec{\alpha} \in I$ avec $\alpha_1 \in \bar{\alpha}$.* Puisque φ est une solution de I, on a $\varphi(\alpha_2) \approx c\,\varphi(\vec{\alpha})$. Par la propriété 1.5 (page 23), on en déduit que $\mathrm{h}(\varphi(\alpha_2)) = \mathrm{h}(c\,\varphi(\vec{\alpha}))$. Or, par définition, $\mathrm{h}(c\,\varphi(\vec{\alpha})) = 1+\max_{\alpha\in\bar{\alpha}} \mathrm{h}(\varphi(\alpha))$. Puisque $\alpha_1 \in \bar{\alpha}$, ceci implique $h(\varphi(\alpha_1)) < h(\varphi(\alpha_2))$, ce qui permet de conclure.

∘ *Cas $\alpha_2 \approx (\xi : \alpha_1; \tau) \in I$.* Puisque φ est une solution de I, on a $\varphi(\alpha_2) \approx (\xi : \varphi(\alpha_1); \varphi(\tau))$. Par la propriété 1.5 (page 23), on en déduit que $\mathrm{h}(\varphi(\alpha_2)) = \mathrm{h}(\xi : \varphi(\alpha_1); \varphi(\tau))$. Or, par définition, $\mathrm{h}(\xi : \varphi(\alpha_1); \varphi(\tau)) = \max\{\mathrm{h}(1 + \varphi(\alpha_1)); \mathrm{h}(\varphi(\tau))\}$. On en déduit donc que $\mathrm{h}(\varphi(\alpha_1)) < h(\varphi(\alpha_2))$, ce qui permet de conclure.

∘ *Cas $\alpha_2 \approx (\xi : \tau; \alpha_1) \in I$.* Puisque φ est une solution de I, on a $\varphi(\alpha_2) \approx (\xi : \varphi(\tau); \varphi(\alpha_1))$. Par propriété 1.5 (page 23), on en déduit que $\mathrm{h}(\varphi(\alpha_2)) = \mathrm{h}(\xi : \varphi(\tau); \varphi(\alpha_1))$. Par définition, $\mathrm{h}(\xi : \varphi(\tau); \varphi(\alpha_1)) = \max\{\mathrm{h}(\varphi(\tau)) + 1; \mathrm{h}(\varphi(\alpha_1))\}$, ce qui implique $\mathrm{h}(\varphi(\alpha_1)) \leq \mathrm{h}(\varphi(\alpha_2))$. Si $\mathrm{h}(\varphi(\alpha_1)) < \mathrm{h}(\varphi(\alpha_2))$, alors on peut conclure immédiatement. Sinon, on a $\mathrm{h}(\varphi(\alpha_1)) = \mathrm{h}(\varphi(\alpha_2))$. Or, les types α_2 et $(\xi : \tau; \varphi_1)$ sont membres du même multi-squelette. Ils ont donc la même sorte. On en déduit que α_1 et α_2 sont respectivement de sortes $\mathsf{Row}_{\Xi}\,\kappa$ et $\mathsf{Row}_{\xi.\Xi}\,\kappa$, pour des certains Ξ et κ, ce qui permet de conclure. ⌟

Théorème 10.19 (Test d'occurrence) *Une contrainte cyclique n'est pas satisfiable.* □

⌜ *Preuve.* Soit I une contrainte cyclique et I' l'un de ses représentants nommés tel que $\prec_{I'}$ soit cyclique. Supposons I satisfiable ; par la propriété 10.6 (page 177), I' est également satisfiable. Soit φ l'une de ses solutions. En appliquant le lemme 10.18 à un cycle de $\prec_{I'}$, on obtient, pour chaque variable α du cycle, $\mathrm{h}(\varphi(\alpha)) < \mathrm{h}(\varphi(\alpha))$ ou bien $\varphi(\alpha)$ de sortes $\mathsf{Row}_{\xi_1}\,\kappa$ et $\mathsf{Row}_{\xi_2}\,\kappa$ avec $\xi_1 \subset \xi_2$. Dans les deux cas, il s'agit d'une contradiction. On en déduit que I n'est pas satisfiable. ⌟

En pratique, le test d'occurrence peut être effectué en temps linéaire grâce à un tri topologique du graphe défini par les multi-squelettes [Knu68]. Il faut noter que, puisque l'unification termine même en présence de structures cycliques, il n'est pas nécessaire d'effectuer de tests d'occurrence lors de son déroulement. Il est ainsi suffisant de réaliser *une seule fois* cette vérification, avant la procédure d'expansion décrite dans la prochaine section, qui est quant à elle susceptible de ne pas terminer en présence de cycles.

10.2.3 Expansion et décomposition

La troisième étape de l'algorithme de résolution consiste à expliciter totalement la structure des types induite par les multi-squelettes, de manière à pouvoir décomposer les inégalités et les gardes

Expansion

$$(\exists\bar\alpha.I_1)\wedge I_2 \;\to_e\; \exists\bar\alpha.(I_1\wedge I_2) \quad \text{si } \bar\alpha \# \mathrm{ftv}(I_2) \qquad \text{SPE-EX-AND}$$

$$\langle\bar{\bar\alpha}\rangle^\bullet \approx \langle\bar{\bar\tau} = d\,\vec\beta\rangle^\bullet \approx \tilde{\tilde\tau}\wedge I \;\to_e\; \exists\bar\alpha.\begin{pmatrix}\langle\bar{\bar\alpha} = d\,\vec\alpha\rangle^\circ \approx \langle\bar{\bar\tau} = d\,\vec\beta\rangle^\bullet \approx \tilde{\tilde\tau}\\ \wedge\,(I \oslash \vec\beta_{|1} \approx^{d.1} \vec\alpha_{|1} \oslash\cdots\oslash \vec\beta_{|n} \approx^{d.n} \vec\alpha_{|n})\end{pmatrix} \quad \text{SPE-EXPAND}$$
si les variables de $\vec\alpha$ *sont distinctes*
et non dans $\mathrm{ftv}(\bar{\bar\alpha},\bar{\bar\tau},\vec\beta,\tilde{\tilde\tau},I)$

Décomposition

$$\tilde{\tilde\tau}\wedge\alpha\le^\circ\beta \;\to_e\; \tilde{\tilde\tau}\wedge(\textstyle\bigwedge_{d.j=+}\alpha_j\le^\circ\beta_j)\wedge(\bigwedge_{d.j=-}\beta_j\le^\circ\alpha_j) \quad \text{SPE-DEC-}\le$$
si $\alpha = d\,\alpha_1\cdots\alpha_n \in \tilde{\tilde\tau}$ *et* $\beta = d\,\beta_1\cdots\beta_n \in \tilde{\tilde\tau}$

$$\tilde{\tilde\tau}\wedge\alpha\lessdot\beta \;\to_e\; \tilde{\tilde\tau}\wedge(\textstyle\bigwedge_{j\in\text{guarded-l}(d)}\alpha_j\lessdot\beta) \quad \text{SPE-DEC-L-}\lessdot$$
si $\alpha = d\,\alpha_1\cdots\alpha_n \in \tilde{\tilde\tau}$

$$\tilde{\tilde\tau}\wedge\alpha\lessdot\beta \;\to_e\; \tilde{\tilde\tau}\wedge(\textstyle\bigwedge_{j\in\text{guarded-r}(d)}\alpha\lessdot\beta_j) \quad \text{SPE-DEC-R-}\lessdot$$
si $\beta = d\,\beta_1\cdots\beta_n \in \tilde{\tilde\tau}$

Figure 10.3 – Règles d'expansion et de décomposition

jusqu'à ce qu'elles atteignent des variables sans structure connue, dites terminales. Illustrons tout d'abord ce procédé par un exemple élémentaire de contrainte bien marquée et unifiée :

$$\langle\alpha\rangle^\bullet \approx \langle\beta = c\,\beta_1\cdots\beta_n\rangle^\bullet \wedge (\bigwedge_{i\in[1,n]}\langle\beta_i\rangle^\bullet)\wedge\alpha\le\beta.$$

Toute solution du premier multi-squelette interprète la variable α en un type dont le constructeur de tête est c. On peut donc *expanser* α et réécrire la contrainte précédente sous la forme équivalente

$$\exists\alpha_1\cdots\alpha_n.\big(\langle\alpha = c\,\alpha_1\cdots\alpha_n\rangle^\circ \approx \langle\beta = c\,\beta_1\cdots\beta_n\rangle^\bullet \wedge (\bigwedge_{i\in[1,n]}\alpha_i\approx^{d.i}\beta_i)\wedge\alpha\le\beta\big)$$

dans laquelle les sous-termes du type α sont explicitement nommés $\alpha_1,\ldots,\alpha_n$. Ces nouvelles variables de type sont également introduites dans les multi-squelettes des sous-termes de β, de manière à préserver le caractère unifié de la contrainte. Grâce à cette expansion, il est maintenant possible de *décomposer* l'inégalité $\alpha\le\beta$ selon les variances du constructeur c comme suit :

$$\exists\alpha_1\cdots\alpha_n.\big(\langle\alpha = c\,\alpha_1\cdots\alpha_n\rangle^\circ \approx \langle\beta = c\,\beta_1\cdots\beta_n\rangle^\bullet \wedge (\bigwedge_{i\in[1,n]}\alpha_i\approx^{d.i}\beta_i)\wedge(\bigwedge_{d.j=+}\alpha_j\le\beta_j)\wedge(\bigwedge_{d.j=-}\beta_j\le\alpha_j)\big)$$

Des inégalités sont générées entre les sous-termes de α et β apparaissant en positions covariantes et contravariantes. Il n'est pas nécessaire de considérer, lors de la décomposition d'une inégalité, les arguments en positions invariantes, car ils sont déjà unifiés par le fragment $(\bigwedge_{i\in[1,n]}\alpha_i\approx^{d.i}\beta_i)$ produit ici par l'expansion ou, dans d'autres exemples, l'unification. L'intérêt de ce processus vient du fait que, abstraction faite des quantificateurs de tête, la contrainte obtenue peut être séparée en deux parties :

$$\langle\alpha = c\,\alpha_1\cdots\alpha_n\rangle^\circ \approx \langle\beta = c\,\beta_1\cdots\beta_n\rangle^\bullet \qquad\text{et}\qquad (\bigwedge_{i\in[1,n]}\alpha_i\approx^{d.i}\beta_i)\wedge(\bigwedge_{d.j=+}\alpha_j\le\beta_j)\wedge(\bigwedge_{d.j=-}\beta_j\le\alpha_j)$$

La première, dite *structurelle*, décrit la structure arborescente des types induite par la contrainte ; la seconde, dite *atomique*, relie les feuilles de cette structure par des contraintes de squelette, des égalités, des inégalités et des gardes.

Les procédures d'expansion et de décomposition sont formalisées par les règles de réécriture données figure 10.3. La règle SPE-EX-AND est identique à SPU-EX-AND : son but est de faire remonter les quantificateurs existentiels introduits par l'expansion vers la racine de la contrainte. La règle

SPE-EXPAND effectue l'expansion d'une multi-équation $\bar{\bar{\alpha}}$ apparaissant dans le même multi-squelette qu'un descripteur $d\,\vec{\beta}$. Des variables de type fraîches $\vec{\alpha}$ sont introduites comme sous-termes pour $\bar{\bar{\alpha}}$. De manière à préserver le caractère bien unifié de la contrainte, elles doivent simultanément être placées dans les multi-squelettes des anciennes variables $\vec{\beta}$, comme précisé par la définition suivante.

Définition 10.20 *Soient I une contrainte et α, β deux variables de types telles que β soit membre d'exactement une multi-équation de I. Notons $I = \mathbb{I}[\langle\beta = \bar{\bar{\tau}}\rangle^\iota \approx \tilde{\tilde{\tau}}]$ avec $\alpha\beta \# \mathrm{dpv}(I)$. Alors $I \oslash \beta = \alpha$ et $I \oslash \beta \approx \alpha$ dénotent respectivement les contraintes $\mathbb{I}[\langle\beta = \alpha = \bar{\bar{\tau}}\rangle^\iota \approx \tilde{\tilde{\tau}}]$ et $\mathbb{I}[\langle\beta = \bar{\bar{\tau}}\rangle^\iota \approx \langle\alpha\rangle^\bullet \approx \tilde{\tilde{\tau}}]$.* □

Propriété 10.21 *Quand elle est définie, la contrainte $I \oslash \beta \approx^\nu \alpha$ est équivalente à $I \wedge \beta \approx^\nu \alpha$.* □

Les règles du deuxième groupe de la figure 10.3 décrivent le processus de décomposition des inégalités et des gardes. La règle SPE-DEC-$\leq$ décompose une inégalité entre deux types construits ou termes de rangées en générant des inégalités entre leurs sous-termes en positions covariantes ou contravariantes. Comme expliqué précédemment, il n'est pas nécessaire de considérer les sous-termes en position invariante : ceux-ci ont déjà été identifiés par l'unification, grâce aux règles SPU-LEQ et SPU-DEC-$\approx$, ou par l'expansion. Les règles SPE-DEC-L-$\lessdot$ et SPE-DEC-R-$\lessdot$ effectuent la décomposition des gardes lorsque leurs membres gauches et droits, respectivement, sont des types construits ou des termes de rangée.

La terminaison du processus d'expansion repose sur l'absence de cycle dans la contrainte manipulée, c'est pourquoi le test d'occurrence doit être effectué de manière préalable. Une stratégie d'application efficace des règles d'expansion et de décomposition consiste à traiter les multi-squelettes dans l'ordre topologique $\prec_I$ exhibé par le test d'occurrence, en parcourant la structure des types de haut en bas ; de telle sorte que les variables de type introduites par la règle SPE-EXPAND sont systématiquement générées dans des multi-squelettes qui n'ont pas encore été considérés.

▶ **Exemple** Avant de poursuivre le développement avec la preuve de cette algorithme, j'illustre son déroulement sur un exemple concret, en considérant une contrainte primaire issue du typage dans le système MLIF($\mathcal{X}$) (chapitre 7, page 135) du fragment de code Core ML suivant :

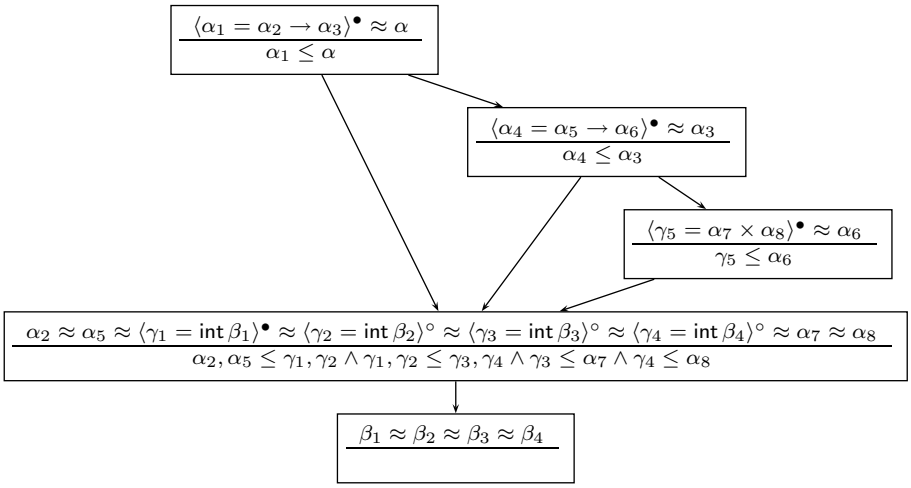
$$\begin{array}{l}\mathsf{let}\ f = \lambda x.\lambda y\big(\mathsf{bind}\ s = x + y\ \mathsf{in}\\ \qquad\qquad\qquad\quad \mathsf{bind}\ p = x \times y\ \mathsf{in}\\ \qquad\qquad\qquad\quad (p + s, p - s)\big)\\ \mathsf{in}\ \ldots\end{array}$$

Pour simplifier les choses, j'omets ici les trois annotations portées par les types flèches, qui n'interviennent pas puisque la fonction f ne produit ni effet de bord ni exception. Le problème produit par le générateur de contraintes du système MLIF($\mathcal{X}$) lors de l'analyse de cette expression comporte deux liaisons let monomorphes pour les variables s et p. Comme je l'expliquerai au chapitre 11 (page 217), celles-ci peuvent être éliminées par la strate secondaire du solveur, de telle sorte que la contrainte à résoudre devient :

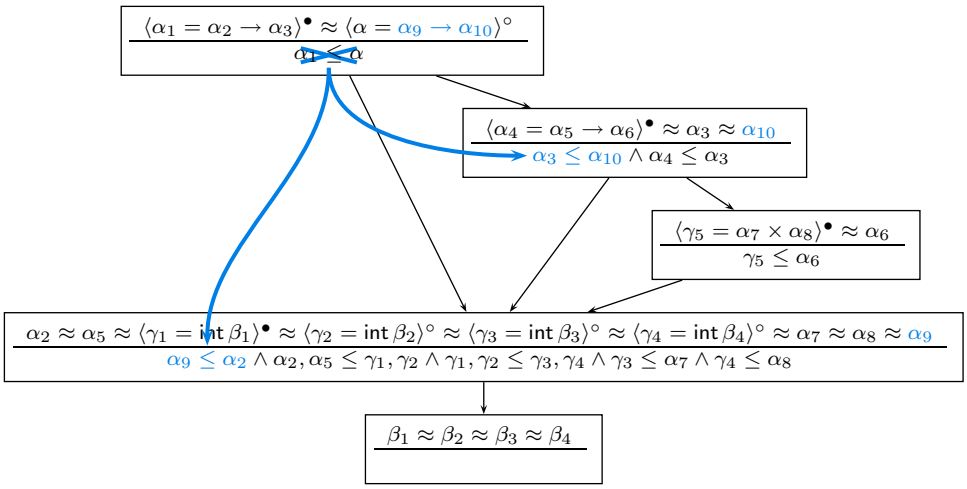
$$\begin{array}{l}\mathsf{let}\ f : \forall\alpha\left[\begin{array}{r}\exists\alpha_1\alpha_2\alpha_3.(\alpha_1 = \alpha_2 \to \alpha_3 \wedge \alpha_1 \leq \alpha\\ \wedge\ \exists\alpha_4\alpha_5\alpha_6.(\alpha_4 = \alpha_5 \to \alpha_6 \wedge \alpha_4 \leq \alpha_3\\ \wedge\ \exists\beta_1\gamma_1.(\gamma_1 = \mathsf{int}\,\beta_1 \wedge \alpha_2 \leq \gamma_1 \wedge \alpha_5 \leq \gamma_1\\ \wedge\ \exists\beta_2\gamma_2.(\gamma_2 = \mathsf{int}\,\beta_2 \wedge \alpha_2 \leq \gamma_2 \wedge \alpha_5 \leq \gamma_2\\ \wedge\ \exists\beta_3\gamma_3.(\gamma_3 = \mathsf{int}\,\beta_3 \wedge \gamma_1 \leq \gamma_3 \wedge \gamma_2 \leq \gamma_3\\ \wedge\ \exists\beta_4\gamma_4.(\gamma_4 = \mathsf{int}\,\beta_4 \wedge \gamma_1 \leq \gamma_4 \wedge \gamma_2 \leq \gamma_4\\ \wedge\ \exists\alpha_7\alpha_8\gamma_5.(\gamma_3 \leq \alpha_7 \wedge \gamma_4 \leq \alpha_8\\ \wedge\gamma_5 = \alpha_7 \times \alpha_8 \wedge \gamma_5 \leq \alpha_6)\cdots)\end{array}\right].\alpha\\ \mathsf{in}\ \ldots\end{array}$$

Dans cette contrainte, α représente le type de f, α_2 et α_5 correspondent aux types de ses arguments x et y, γ_1 et γ_2 à ceux des variables s et p. Le type du résultat de la fonction est γ_5. La propriété 10.8

10.4-a Unification et test d'occurence



10.4-b Expansion et décomposition du premier multi-squelette



10.4-c Expansion et décomposition du deuxième multi-squelette

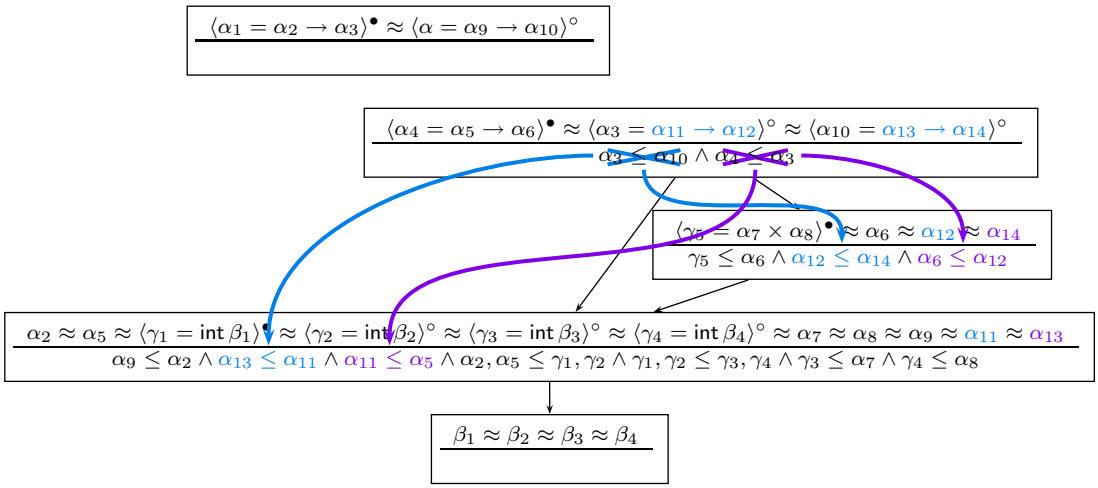


Figure 10.4 – Exemple (1/5)

(page 178) permet de mettre cette contrainte en forme bien marquée en introduisant un multi-squelette trivial pour chaque variable puis en marquant chaque multi-équation ou inéquation par •. On peut ensuite appliquer l'algorithme d'unification et le test d'occurrence. Le résultat obtenu est illustré par la figure 10.4-a. Les quantificateurs existentiels, qui apparaissent maintenant au sommet de la contrainte, sont omis dans cette représentation. Chaque boîte contient un multi-squelette, ainsi que les inégalités qui font intervenir ses membres (la ligne de séparation horizontale doit être lue comme une conjonction). Les flèches qui relient les boîtes décrivent l'ordre hiérarchique trouvé par le test d'occurrence. Les multi-squelettes peuvent être considérés dans ce même ordre par l'algorithme d'expansion et de décomposition. Sa première étape est donnée par la figure 10.4-b : un descripteur $\alpha_9 \to \alpha_{10}$ est introduit pour la variable α (la quantification existentielle de ces deux variables, qui peut être remontée au sommet de la contrainte, est à nouveau omise). Cela permet de décomposer l'inégalité $\alpha_1 \leq \alpha$. Les deux inéquations obtenues sont directement placées dans les multi-squelettes fils. La figure 10.4-c décrit l'effet de l'algorithme d'expansion et de décomposition sur le deuxième multi-squelette : deux variables, α_3 et α_{10}, sont expansées, et deux inégalités décomposées. Le processus continue de même avec les deux multi-squelettes suivants, de manière à décomposer toutes les inégalités jusqu'à ce qu'elles atteignent le dernier. On observe cependant que la taille du problème augmente considérablement d'étape en étape : au terme de l'expansion et de la décomposition, nous aurions 41 variables distinctes (contre 16 dans le problème initial) et 19 inégalités (contre 13). C'est la raison pour laquelle je poursuivrai l'étude de cet exemple une fois les techniques de simplification introduites.

La suite de cette section est dévolue à la preuve de l'algorithme d'expansion et de décomposition. J'établis tout d'abord que chaque pas de réécriture par $\rightarrowtriangle_e$ préserve les propriétés obtenues par les étapes précédentes — unification et test d'occurrence — de l'algorithme.

Lemme 10.22 *Les règles de la figure 10.3 (page 185) préservent les caractères bien marqué, unifié et acyclique des contraintes.* □

J'omets la preuve de ce lemme qui ne présente pas de difficulté particulière, mais nécessite un développement relativement long, consistant à examiner successivement les membres de chacune des règles de la figure 10.3 (page 185) dans un contexte arbitraire. Je poursuis la preuve de l'algorithme en montrant sa correction vis-à-vis de la sémantique des contraintes et sa terminaison.

Lemme 10.23 (Correction) *La réduction $\rightarrowtriangle_e$ préserve la sémantique des contraintes.* □

⌜ *Preuve.* J'examine successivement les différentes règles.

∘ *Cas* SPE-EX-AND. Par LOG-EX-AND.

∘ *Cas* SPE-EXPAND. Par la propriété 10.21 (page 186), il suffit de vérifier que les contraintes $\langle\bar\alpha\rangle^\bullet \approx \langle\bar\tau = d\,\beta_1\cdots\beta_n\rangle^\bullet \approx \tilde{\bar\tau}$ **(1)** et $\exists\alpha_1\cdots\alpha_n.(\langle\bar\alpha = d\,\alpha_1\cdots\alpha_n\rangle^\circ \approx \langle\bar\tau = d\,\beta_1\cdots\beta_n\rangle^\bullet \approx \tilde{\bar\tau} \wedge \alpha_1 \approx^{d.1} \beta_1 \approx \cdots \approx \alpha_n \approx^{d.n} \beta_n)$ **(2)**, sont équivalentes où $\alpha_1, \ldots, \alpha_n$ sont distinctes **(3)** et non dans $\mathrm{ftv}(\bar\alpha, \bar\tau, \beta_1\cdots\beta_n, \tilde{\bar\tau})$ **(4)**. Puisque, par SSK-TYPE et SSK-ROW, $d\,\alpha_1\cdots\alpha_m \approx d\,\beta_1\cdots\beta_n \Vdash \alpha_1 \approx^{d.1} \beta_1 \wedge\cdots\wedge \alpha_n \approx^{d.n} \beta_n$, la contrainte (2) est équivalente à $\exists\alpha_1\cdots\alpha_n.(\langle\bar\alpha = d\,\alpha_1\cdots\alpha_n\rangle^\circ \approx \langle\bar\tau = d\,\beta_1\cdots\beta_n\rangle^\bullet \approx \tilde{\bar\tau})$. Par LOG-EX-AND, (4) et la transitivité de $=$, cette dernière est équivalente à $\langle\bar\alpha\rangle^\bullet \approx \langle\bar\tau = d\,\beta_1\cdots\beta_n\rangle^\bullet \approx \tilde{\bar\tau} \wedge \exists\alpha_1\cdots\alpha_n.\langle\bar\alpha = d\,\alpha_1\cdots\alpha_n\rangle^\circ$ **(5)**. Or, grâce à (3) et (4), $\langle\bar\alpha\rangle^\bullet \approx \langle\bar\tau = d\,\beta_1\cdots\beta_n\rangle^\bullet \approx \tilde{\bar\tau} \Vdash \exists\alpha_1\cdots\alpha_n.\langle\bar\alpha = d\,\alpha_1\cdots\alpha_n\rangle^\circ$. On en déduit donc, par LOG-DUP, que (5) et (1), puis (2) et (1) sont équivalentes.

∘ *Cas* SPE-DEC-$\leq$. Il suffit de montrer que les contraintes $\tilde\tau \wedge \alpha \leq^\circ \beta$ **(1)** et $\tilde\tau \wedge (\bigwedge_{d.j=+} \alpha_j \leq^\circ \beta_j) \wedge (\bigwedge_{d.j=-} \beta_j \leq^\circ \alpha_j)$ **(2)** sont équivalentes, sous les hypothèses $\alpha = d\,\alpha_1\cdots\alpha_n \in \tilde\tau$ **(3)** et $\beta = d\,\beta_1\cdots\beta_n \in \tilde\tau$ **(4)**. Par (3) et (4), on a $d\,\alpha_1\cdots\alpha_n \approx d\,\beta_1\cdots\beta_n \in \tilde\tau$. On en déduit que $\tilde\tau \Vdash \bigwedge_{d.i=\pm} \alpha_i = \beta_i$ **(5)**. Par définition de l'ordre de sous-typage, la contrainte (1) est équivalente à $\tilde\tau \wedge (\bigwedge_{d.j=+} \alpha_j \leq^\circ \beta_j) \wedge (\bigwedge_{d.j=-} \beta_j \leq^\circ \alpha_j) \wedge (\bigwedge_{d.i=\pm} \alpha_i = \beta_i)$. Or, par LOG-DUP et (5), cette dernière est elle-même équivalente à (2), ce qui permet de conclure.

∘ *Cas* SPE-DEC-L-⋖ *et* SPE-DEC-R-⋖. Par la propriété 10.1 (page 172). ⌟

Lemme 10.24 (Terminaison) *Le système de réécriture $\twoheadrightarrow_e$ est fortement normalisant sur les contraintes acycliques.* □

⌜ *Preuve.* Soit I' une contrainte sans quantificateur existentiel. La hauteur d'une variable de type α dans I' est le maximum des longueurs des chemins d'extrémité α dans le graphe $\prec_{I'}$, *i.e.* $\max\{\, n \mid \exists \alpha_1 \cdots \alpha_n \alpha_n \cdots \prec_{I'} \alpha_1 \prec_{I'} \alpha \,\}$. Toutes les variables d'un multi-squelette de I' ayant la même hauteur, on peut parler de hauteur d'une multi-équation dans I'. De plus, on définit la hauteur d'une inégalité $\alpha_1 \leq \alpha_2$ ou d'une garde $\alpha_1 \lessdot \alpha_2$ dans I' comme la somme des hauteurs de α_1 et α_2 dans I'.

Soit I une contrainte acyclique et I' un de ses représentants nommés. On mesure la contrainte I par le produit lexicographique des quantités suivantes :

(1) Le multi-ensemble des hauteurs des multi-équations dans I' ne contenant que des variables.

(2) Le multi-ensemble des hauteurs des inégalités et des gardes dans I'.

(3) Le multi-ensemble des profondeurs des quantificateurs existentiels.

On peut vérifier que cette définition ne dépend pas du choix du représentant I'.

La règle SPE-EXPAND insère un type non variable dans une multi-équation ne contenant que des variables, et génère des multi-équations de hauteur strictement inférieure, puisque la contrainte est acyclique. Elle diminue donc (1). La règle SPE-DEC-$\leq$ remplace une inégalité par plusieurs de hauteur strictement inférieure, réduisant ainsi (2) sans affecter (1). De même, les règles SPE-DEC-L-⋖ et SPE-DEC-R-⋖ remplacent une garde par plusieurs de hauteur strictement inférieure, diminuant également (2) sans modifier (1). Enfin, la règle SPE-EX-AND diminue (3) en remontant un quantificateur existentiel. Puisqu'elle n'affecte pas les représentants nommés de la contrainte réduite, elle préserve (1) et (2).

L'ordre sur ces mesures n'admet pas de chaîne strictement décroissante infinie. On en déduit que $\twoheadrightarrow_e$ termine sur toute contrainte acyclique. ⌟

Pour terminer cette étude de l'algorithme d'expansion et de décomposition, je dois préciser la forme de ses sorties. Pour énoncer ce résultat (lemme 10.28), je définis deux classes particulières de contraintes dites structurelles et atomiques.

Définition 10.25 (Contrainte structurelle) *Une contrainte est* **structurelle** *si elle est une conjonction de multi-squelettes dont chaque multi-équation contient un descripteur de la forme $d\,\vec{\alpha}$. Son support est l'ensemble des variables de type membres d'une de ses multi-équations.* □

Définition 10.26 (Contrainte atomique) *Une contrainte est* **atomique** *si elle est une conjonction d'inégalités, de gardes et de multi-squelettes dont les éléments sont des variables de type ou des constantes atomiques.* □

Définition 10.27 (Contrainte réduite) *Une contrainte I est* **réduite** *si et seulement si elle est de la forme $\mathbb{X}[I_s \wedge I_a]$ où*

(i) *I_s est une contrainte structurelle,*

(ii) *I_a est une contrainte atomique dont les variables libres ne sont pas dans le support de I_s.*

Une contrainte réduite se décompose en deux parties. La première, I_s, est composée de multi-squelettes décrivant des structures d'arbres (dans le cas où la contrainte est acyclique), dont les nœuds sont les variables de type apparaissant dans ses multi-équations. La contrainte I_a porte quant-à-elle exclusivement sur les feuilles de ces arbres. Elle peut comporter des contraintes de squelette ou des égalités entre variables et/ou constantes atomiques, ainsi que des inégalités et des

gardes. L'intérêt de cette décomposition apparaît grâce au théorème 10.33 (page 192) qui montre qu'une contrainte réduite est satisfiable si et seulement si sa partie atomique l'est.

Le lemme suivant précise la forme des résultats produits par l'algorithme d'expansion lorsque son entrée est une contrainte issue de l'algorithme d'unification puis du test d'occurrence.

Lemme 10.28 (Formes normales) *Si I est une contrainte bien marquée, unifiée, acyclique et normale pour les règles de la figure 10.3 (page 185) alors elle est réduite.* □

⌜ *Preuve.* Soit I une contrainte bien marquée, unifiée et normale pour les règles de la figure 10.3 (page 185). Puisque I ne peut être réduite par SPE-EX-AND, elle est de la forme $\mathbb{X}[I']$ où I' est sans quantificateur existentiel. Posons $I' = I_s \wedge I_a$ où I_s est la conjonction des multi-squelettes de I' contenant au moins un descripteur de la forme $d\,\vec{\alpha}$. Pour conclure, il suffit de montrer que I_s et I_a vérifient respectivement les points (i) et (ii) de la définition 10.27.

Je montre tout d'abord que I_s vérifie le point (i). Les multi-squelettes de I_s sont nécessairement de sorte Type ou $\mathsf{Row}_\Xi\,\kappa$. De plus, puisque I est unifiée, ses multi-squelettes ne peuvent contenir que des petits types. On en déduit qu'une multi-équation de I_s sans type de la forme $d\,\vec{\alpha}$ ne contient que des variables. Supposons ainsi que I_s contienne un multi-squelette $\langle\bar{\alpha}\rangle^\iota \approx \tilde{\bar{\tau}}$. De par la définition de I_s, $\tilde{\bar{\tau}}$ contient un type $d\,\alpha_1 \cdots \alpha_n$. De plus, I étant bien marquée, on peut supposer que ce type est dans une multi-équation marquée $\bullet$. Écrivons ainsi $\tilde{\bar{\tau}}$ sous la forme $\langle\bar{\tau}' = d\,\alpha_1 \cdots \alpha_n\rangle^\bullet \approx \tilde{\bar{\tau}}'$ puis I' comme $\langle\bar{\tau}\rangle^\iota \approx \langle\bar{\tau}' = d\,\alpha_1 \cdots \alpha_n\rangle^\bullet \approx \tilde{\bar{\tau}}' \wedge I''$. Puisque I' est bien marquée, chaque variable parmi $\alpha_1, \ldots, \alpha_n$ apparaît dans un multi-squelette de I'. De plus, I' étant acyclique, ces multi-squelettes sont tous distincts de $\langle\bar{\tau}\rangle^\iota \approx \langle\bar{\tau}' = d\,\alpha_1 \cdots \alpha_n\rangle^\bullet \approx \tilde{\bar{\tau}}'$ et sont donc dans I''. On en déduit que I' peut être réduit par SPE-EXPAND, ce qui contredit l'hypothèse de normalité. La contrainte I_s vérifie donc la propriété (i).

Je considère maintenant la contrainte I_a. Puisque I' est unifiée, ses multi-squelettes ne contiennent que des petits types. Puisque tout multi-squelette contenant un type qui n'est pas une variable ou une constante atomique est dans I_s, on en déduit que I_a est atomique. Soit $\alpha \in \mathrm{ftv}(I_a)$. Trois cas sont envisageables.

· *Si α apparaît dans un multi-squelette de I_a*, α ne peut être membre d'une multi-équation de I_s car I' est unifiée.

· *Si α apparaît dans une inégalité de I_a,* notons β l'autre membre de cette inégalité. Puisque I' est unifiée, α et β apparaissent dans le même multi-squelette $\tilde{\bar{\tau}}$ de I'. De plus, si ce multi-squelette était dans I_s, par le point (i) et puisque I' est unifiée, il existerait $d\,\alpha_1 \cdots \alpha_n$ et $d\,\beta_1 \cdots \beta_n$ tels que $\alpha = d\,\alpha_1 \cdots \alpha_n \in \tilde{\bar{\tau}}$ et $\beta = d\,\beta_1 \cdots \beta_n \in \tilde{\bar{\tau}}$. La règle SPE-DEC-$\leq$ serait applicable. On en déduit que le multi-squelette de α et β est dans I_a.

· *Si α apparaît dans une garde de I_a,* elle ne peut être membre d'un multi-squelette $\tilde{\bar{\tau}}$ de I_s, car, par le point (i), il existerait $d\,\alpha_1 \cdots \alpha_n$ tel que $\alpha = d\,\alpha_1 \cdots \alpha_n \in I_s$ et SPE-DEC-L-$\lessdot$ ou SPE-DEC-R-$\lessdot$ serait applicable.

On en déduit finalement que toute variable libre dans $\mathrm{ftv}(I_a)$ n'est pas dans le support de I_s, ce qui termine la preuve du point (ii). ⌟

Ces quatre lemmes permettent d'obtenir le théorème suivant, qui établit que la relation $\rightarrowtriangle_e$ donne un algorithme réécrivant une contrainte issue de l'unification et du test d'occurrence en une contrainte réduite.

Théorème 10.29 (Expansion et décomposition) *Soit I une contrainte bien marquée, unifiée et acyclique différente de* false. *Alors $\rightarrowtriangle_e$ termine sur I et, si $I \rightarrowtriangle_e^{**} I'$, alors I' est bien marquée, unifiée, acyclique et réduite.* □

Je donne maintenant le théorème qui permet de ramener la résolution d'une contrainte réduite à celle d'une contrainte atomique. Je montre en fait un résultat légèrement plus général, en considérant une contrainte de la forme $I_s \wedge I$ où I_s est structurelle de support disjoint de $\mathrm{ftv}(I)$. Je ne

suppose pas explicitement que la contrainte résiduelle I est atomique : cette hypothèse n'est en effet pas nécessaire.

Théorème 10.30 *Soit I_s une contrainte structurelle de support $\bar{\alpha}$ et acyclique. Soit I une contrainte telle que $\bar{\alpha} \# I$ et $I_s \wedge I$ soit unifiée. Si I est satisfiable alors $I_s \wedge I$ est satisfiable.* □

⌜ *Preuve.* La contrainte I étant satisfiable, considérons l'une de ses solutions φ, *i.e.* $\varphi \vdash I$ **(1)**. Puisque I_s est acyclique, $\prec_{I_s}$ définit un ordre bien fondé sur $\bar{\alpha}$ **(2)**. Soit $\alpha \in \bar{\alpha}$. Puisque $I_s \wedge I$ est unifiée, α apparaît dans exactement une multi-équation de I_s et il existe un unique type non variable $d\,\alpha_1 \cdots \alpha_n$ tel que $\alpha = d\,\alpha_1 \cdots \alpha_n \in I_s$ **(3)**. On a de plus, par définition de $\prec_{I_s}$, pour tout $i \in [1,n]$, $\alpha_i \prec_{I_s} \alpha$ **(4)**. Grâce à (2), (3) et (4), on peut définir une affectation φ' par :

$$\varphi'(\alpha) = \begin{cases} \varphi(\alpha) & \text{si } \alpha \notin \bar{\alpha} \\ d\,\varphi'(\alpha_1) \cdots \varphi'(\alpha_n) & \text{si } \alpha \in \bar{\alpha} \text{ et } \alpha = d\,\alpha_1 \cdots \alpha_n \in I_s \end{cases} \qquad \textbf{(5)}$$

Puisque $\bar{\alpha} \# \mathrm{ftv}(I)$, φ' et φ coïncident sur $\mathrm{ftv}(I)$. On en déduit, grâce à (1), $\varphi' \vdash I$. Pour conclure, il me suffit donc de montrer que $\varphi' \vdash I_s$. Puisque I_s est structurelle et $I_s \wedge I$ est unifiée, il est suffisant d'établir les propriétés suivantes :

∘ $\tau = \tau' \in I_s$ *implique* $\varphi'(\tau) = \varphi'(\tau')$ **($\mathbf{P_1}$)**. Soit τ et τ' deux types tels que $\tau = \tau' \in I_s$ **($\mathbf{H_1}$)**, je veux montrer que $\varphi'(\tau) = \varphi'(\tau')$ **($\mathbf{C_1}$)**. Puisque $I_s \wedge I$ est unifiée et I_s structurelle, τ et τ' sont nécessairement des variables ou des descripteurs de la forme $d\,\vec{\alpha}$.

· *Si τ est β et τ' est β'* alors $\beta \in \bar{\alpha}$ **(6)** et $\beta' \in \bar{\alpha}$ **(7)**. Puisque I_s est structurelle, il existe $d\,\vec{\beta}$ et $d\,\vec{\beta'}$ tels que $\beta = d\,\vec{\beta} \in I_s$ **(8)** et $\beta' = d\,\vec{\beta'} \in I_s$ **(9)**. Puisque $I \wedge I_s$ est unifiée, ($\mathrm{H_1}$), (8) et (9) impliquent $d\,\vec{\beta} = d\,\vec{\beta'} \in I_s \wedge I$ puis $d\,\vec{\beta} = d\,\vec{\beta'}$. Or, par (6), (7) et (5), $\varphi'(\beta) = \varphi'(d\,\vec{\beta})$ et $\varphi'(\beta') = \varphi'(d\,\vec{\beta'})$. On en déduit l'égalité $\varphi'(\beta) = \varphi'(\beta')$, qui est le but ($\mathrm{C_1}$).

· *Si τ et τ' sont deux descripteurs*, alors, puisque $I_s \wedge I$ est unifiée, on a nécessairement $\tau = \tau'$, ce qui donne ($\mathrm{C_1}$).

· *Si τ est β et τ' est $d\,\vec{\beta}$, ou, symétriquement, si τ est $d\,\vec{\beta}$ et τ' est β*, on a $\beta \in \bar{\alpha}$. On déduit de ($\mathrm{H_1}$) et (5) que $\varphi'(\beta) = \varphi'(d\,\vec{\beta})$, ce qui donne le but ($\mathrm{C_1}$) : $\varphi'(\tau) = \varphi'(\tau')$.

∘ $\tau \approx \tau' \in I_s$ *implique* $\varphi'(\tau) \approx \varphi'(\tau')$ **($\mathbf{P_2}$)**. Soient τ et τ' deux types tels que $\tau \approx \tau' \in I_s$ **($\mathbf{H_2}$)**, je veux montrer que $\varphi'(\tau) \approx \varphi'(\tau')$ **($\mathbf{C_2}$)**. Puisque $I_s \wedge I$ est unifiée, τ et τ' sont nécessairement des variables ou des descripteurs de la forme $d\,\vec{\alpha}$.

· *Je m'intéresse tout d'abord au cas où τ et τ' sont des variables β et β'.* Je procède pour cela par induction sur l'ordre $\prec_{I_s}$. Puisque I_s est structurelle de support $\bar{\alpha}$, par ($\mathrm{H_2}$), on a $\beta, \beta' \in \bar{\alpha}$ **(10)**, et il existe deux descripteurs $d\,\beta_1 \cdots \beta_n$ et $d'\,\beta'_1 \cdots \beta'_{n'}$ tels que $\beta = d\,\beta_1 \cdots \beta_n \in I_s$ **(11)** et $\beta' = d'\,\beta'_1 \cdots \beta'_{n'} \in I_s$ **(12)**, avec, pour tout $i \in [1,n]$, $\beta_i \prec_{I_s} \beta$ **(13)** et, pour tout $i \in [1,n']$, $\beta'_i \prec_{I_s} \beta'$ **(14)**. Puisque $I_s \wedge I$ est unifiée, (11), (12) et ($\mathrm{H_2}$) impliquent $d = d'$ et $n = n'$ et $\forall i \in [1,n]$ $\beta_i \approx^{d.i} \beta'_i \in$ **(15)**. Par (10), (11) et (12) donnent respectivement $\varphi'(\beta) = d\,\varphi'(\beta_1) \cdots \varphi'(\beta_n)$ **(16)** et $\varphi'(\beta') = d\,\varphi'(\beta'_1) \cdots \varphi'(\beta'_n)$ **(17)**. Soit $i \in [1,n]$ **(18)**. Si $\beta_i \approx^{d.i} \beta'_i \in I$ alors β_i et β'_i ne sont pas dans $\bar{\alpha}$ et $\varphi'(\beta_i) = \varphi(\beta_i)$ et $\varphi'(\beta'_i) = \varphi(\beta'_i)$; ce dont on déduit, par (1) et (15), $\varphi'(\beta_i) \approx^{d.i} \varphi'(\beta'_i)$. Sinon, par (15), on a $\beta_i \approx^{d.i} \beta'_i \in I_s$ **(19)** : soit $d.i = \pm$, et ($\mathrm{P_1}$) donne $\varphi'(\beta_i) = \varphi(\beta'_i)$; ou soit $d.i \in \{+,-\}$, et grâce à (13) et (14), on peut appliquer l'hypothèse d'induction à (19) pour obtenir $\varphi'(\beta_i) \approx \varphi'(\beta'_i)$. Finalement, déchargeant (18), on conclut que, pour tout $i \in [1,n]$, $\varphi'(\beta_i) \approx^{d.i} \varphi'(\beta'_i)$. Par (16) et (17), on en conclut que $\varphi'(\beta) \approx \varphi'(\beta')$.

· *Si τ et τ' sont respectivement $d\,\beta_1 \cdots \beta_n$ et $d\,\beta'_1 \cdots \beta'_n$.* Puisque $I_s \wedge I$ est unifiée, on a, pour tout $i \in [1,n]$, $\beta_i \approx^{d.i} \beta'_i \in I_s \wedge I$. En utilisant le cas précédent (si $\beta_i \approx^{d.i} \beta'_i \in I_s$) ou la coïncidence de φ' et φ sur $\mathcal{V}\backslash\bar{\alpha}$ (si $\beta_i \approx^{d.i} \beta'_i \in I$), on en déduit que, pour tout $i \in [1,n]$, $\varphi'(\beta_i) \approx^{d.i} \varphi'(\beta'_i)$. Cela implique $\varphi'(\tau) \approx \varphi'(\tau')$.

· *Si l'un des types τ ou τ' est une variable β, et l'autre n'est pas une variable*, on procède de même en considérant le type $d\,\beta_1 \cdots \beta_n$ tel que $\beta = d\,\beta_1 \cdots \beta_n \in I_s$. ⌟

$$\text{ATM-EQ}\ \frac{\alpha = \beta \in I}{\alpha \leq \beta \in^* I} \qquad \text{ATM-LEQ}\ \frac{\alpha \leq \beta \in I}{\alpha \leq \beta \in^* I} \qquad \text{ATM-GD}\ \frac{\alpha \lessdot \beta \in I}{\alpha \lessdot \beta \in^* I} \qquad \text{ATM-ATOM-GD}\ \frac{\alpha \lessdot \beta \in^* I \qquad \alpha, \beta \in \mathcal{V}_{\mathsf{Atom}}}{\alpha \leq \beta \in^* I}$$

$$\text{ATM-LEQ-LEQ}\ \frac{\alpha_1 \leq \alpha_2 \in^* I \qquad \alpha_2 \leq \alpha_3 \in^* I}{\alpha_1 \leq \alpha_3 \in^* I} \qquad \text{ATM-GD-GD}\ \frac{\alpha_1 \lessdot \alpha_2 \in^* I \qquad \alpha_2 \lessdot \alpha_3 \in^* I}{\alpha_1 \lessdot \alpha_3 \in^* I}$$

$$\text{ATM-LEQ-GD}\ \frac{\alpha_1 \leq \alpha_2 \in^* I \qquad \alpha_2 \lessdot \alpha_3 \in^* I}{\alpha_1 \lessdot \alpha_3 \in^* I} \qquad \text{ATM-GD-LEQ}\ \frac{\alpha_1 \lessdot \alpha_2 \in^* I \qquad \alpha_2 \leq \alpha_3 \in^* I}{\alpha_1 \lessdot \alpha_3 \in^* I}$$

Figure 10.5 – Résolution des contraintes atomiques

Corollaire 10.31 *Soit $I = \mathbb{X}[I_s \wedge I_a]$ une contrainte unifiée, acyclique et réduite, écrite sous la forme donnée par la définition 10.27 (page 189). Alors I est satisfiable si et seulement si I_a l'est.* □

Associé aux résultats précédents (théorèmes 10.15, 10.17 et 10.29), cet énoncé montre que les procédures d'unification, puis d'expansion et de décomposition permettent de ramener la satisfiabilité d'une contrainte arbitraire à celle d'une contrainte atomique. En d'autres termes, il me suffit maintenant de donner un algorithme décidant la satisfiabilité des contraintes atomiques, ce que je fais dans la prochaine sous-section.

10.2.4 Résolution des contraintes atomiques

La résolution des contraintes atomiques est un problème bien connu : en 1992, Tiuryn [Tiu92] a proposé un algorithme linéaire pour déterminer la satisfiabilité d'inégalités dans le cas relativement général où l'ensemble des atomes forme une union disjointe de treillis, comme l'ensemble des types bruts $\mathcal{T}$. La procédure que je présente ici en est directement dérivée ; elle est cependant légèrement étendue de manière à traiter les gardes en plus des inégalités.

Considérons une contrainte atomique I. Sa satisfiabilité peut être déterminée en considérant le graphe défini sur les variables de type par les inégalités et les gardes, en vérifiant que tout chemin dans ce graphe entre deux (variables unifiées à des) constantes atomiques est valide pour l'ordre du treillis atomique $\leq_{\mathcal{L}}$. Rappelons que toutes les variables d'une contrainte atomique ne sont pas nécessairement de sorte Atom : les chemins qui peuvent comporter des arcs $\leq$ et $\lessdot$ sont formalisés par les règles de la figure 10.5. Elles définissent des jugements $\alpha \leq \beta \in^* I$ et $\alpha \lessdot \beta \in^* I$, dont l'interprétation sémantique est donnée par le lemme suivant.

Lemme 10.32 *Soit I une contrainte atomique. Si $\alpha \leq \beta \in^* I$ alors $I \Vdash \alpha \leq \beta$. Si $\alpha \lessdot \beta \in^* I$ alors $I \Vdash \alpha \lessdot \beta$.* □

⌜ *Preuve.* Par induction sur la dérivation du jugement donné en hypothèse. ⌟

Théorème 10.33 *Soit I une contrainte atomique, unifiée et bien marquée. I est satisfiable si et seulement si, pour tous atomes ℓ_1 et ℓ_2 et toutes variables α_1 et α_2 de sorte* Atom*, si $\alpha_1 = \ell_1 \in I$, $\alpha_2 = \ell_2 \in I$ et $\alpha_1 \leq \alpha_2 \in^* I$ alors $\ell_1 \leq_{\mathcal{L}} \ell_2$.* □

⌜ *Preuve.* Soit I une contrainte atomique et unifiée. On veut montrer l'équivalence entre les propriétés I est satisfiable **($\mathbf{P_1}$)** et pour tous atomes ℓ_1 et ℓ_2 et toutes variables α_1 et α_2 de sorte Atom, si $\alpha_1 = \ell_1 \in I$, $\alpha_2 = \ell_2 \in I$ et $\alpha_1 \leq \alpha_2 \in I$ alors $\ell_1 \leq_{\mathcal{L}} \ell_2$ **($\mathbf{P_2}$)**.

Supposons tout d'abord la propriété (P_1) vérifiée : soit φ une solution de I **(1)**. Soient ℓ_1, ℓ_2 deux atomes, et α_1, α_2 deux variables de sorte Atom tels que $\alpha_1 = \ell_1 \in I$ **(2)**, $\alpha_2 = \ell_2 \in I$ **(3)**

et $\alpha_1 \leq \alpha_2 \in^* I$ **(4)**. Grâce à l'hypothèse (1), (4) implique, par le lemme 10.32, $\varphi(\alpha_1) \leq \varphi(\alpha_2)$ **(5)** ; et (2) et (3) donnent respectivement $\varphi(\alpha_1) = \ell_1$ et $\varphi(\alpha_2) = \ell_2$. On en déduit $\ell_1 \leq \ell_2$ **(6)**. En déchargeant les hypothèses (2), (3) et (4) sur (6), on obtient (P_2).

Inversement, supposons maintenant que I vérifie la propriété (P_2). Étant donnée une variable de type α, je définis sa borne inférieure dans I (notée : $\mathrm{lb}_I(\alpha)$) par :

$$\begin{aligned} \mathrm{lbs}_I(\alpha) &= \{\, \ell \mid \exists \beta \;\; \beta = \ell \in I \text{ et } (\beta \leq \alpha \in^* I \text{ ou } \beta \lessdot \alpha \in^* I) \,\} \\ \mathrm{lb}_I(\alpha) &= \sqcup \mathrm{lbs}_I(\alpha) \end{aligned}$$

Montrons tout d'abord que si $\alpha \leq \alpha' \in^* I$ ou $\alpha \lessdot \alpha' \in^* I$ alors $\mathrm{lb}_I(\alpha) \leq \mathrm{lb}_I(\alpha')$ **(P_3)**. Soient α et α' deux variables telles que $\alpha \leq \alpha' \in^* I$ **(7)** ou $\alpha \lessdot \alpha' \in^* I$ **(8)**. Soit $\ell \in \mathrm{lbs}_I(\alpha)$ **(9)** ; il existe β telle que $\beta = \ell \in I$ **(10)** et soit $\beta \leq \alpha \in^* I$ **(11)** soit $\beta \lessdot \alpha \in^* I$ **(12)**. En utilisant l'une des règles ATM-LEQ, ATM-GD, ATM-LEQ-GD ou ATM-GD-LEQ, on déduit de (7) ou (8) et (11) ou (12) $\beta \leq \alpha' \in^* I$ ou $\beta \lessdot \alpha' \in^* I$. Grâce à (10), $\ell \in \mathrm{lbs}_I(\alpha')$ s'ensuit. En déchargeant (9), on en déduit que $\mathrm{lbs}_I(\alpha) \subseteq \mathrm{lbs}_I(\alpha')$, puis $\mathrm{lb}_I(\alpha) \leq \mathrm{lb}_I(\alpha')$. En déchargeant (7) ou (8), on obtient la propriété (P_3).

Je définis l'affectation φ_I comme suit : si α est une variable de sorte κ alors $\varphi_I(\alpha) = \Lambda_\kappa(\mathrm{lb}_I(\alpha))$ (propriété 10.3, page 173). Pour conclure, il me suffit de montrer que φ_I est une solution de I. Puisque cette contrainte est atomique, il me suffit de vérifier les propriétés suivantes :

◦ *Si* $\alpha \leq \alpha' \in I$ *alors* $\varphi_I(\alpha) \leq \varphi_I(\alpha')$ **(P_4)**. Considérons deux variables α et α' de sorte κ, telles que $\alpha \leq \alpha' \in I$ **(1)**. Par ATM-LEQ, on a $\alpha \leq \alpha' \in^* I$. En utilisant ($P_3$), on en déduit que $\mathrm{lb}_I(\alpha) \leq \mathrm{lb}_I(\alpha')$. Grâce à la propriété 10.3 (page 173), on en conclut $\Lambda_\kappa(\mathrm{lb}_I(\alpha)) \leq \Lambda_\kappa(\mathrm{lb}_I(\alpha'))$, *i.e.* $\varphi_I(\alpha) \leq \varphi_I(\alpha')$.

◦ *Si* $\alpha \lessdot \alpha' \in I$ *alors* $\varphi_I(\alpha) \leq \varphi_I(\alpha')$ **(P_5)**. Considérons deux variables α et α' de sortes respectives κ et κ', telles que $\alpha \lessdot \alpha' \in I$ **(1)**. Par ATM-GD, on a $\alpha \lessdot \alpha' \in^* I$. En utilisant ($P_3$), on en déduit que $\mathrm{lb}_I(\alpha) \leq \mathrm{lb}_I(\alpha')$. Grâce à la propriété 10.3 (page 173), on en conclut que $\Lambda_\kappa(\mathrm{lb}_I(\alpha)) \lessdot \Lambda_{\kappa'}(\mathrm{lb}_I(\alpha'))$, *i.e.* $\varphi_I(\alpha) \lessdot \varphi_I(\alpha')$.

◦ *Si* $\alpha = \alpha' \in I$ *alors* $\varphi_I(\alpha) = \varphi_I(\alpha')$ **(P_6)**. Considérons deux variables α et α' de sorte κ, telles que $\alpha = \alpha' \in I$. Par ATM-EQ, on a $\alpha \leq \alpha' \in^* I$ et $\alpha' \leq \alpha \in^* I$. En utilisant ($P_3$), on en déduit que $\mathrm{lb}_I(\alpha) = \mathrm{lb}_I(\alpha')$, ce qui donne $\Lambda_\kappa(\mathrm{lb}_I(\alpha)) = \Lambda_\kappa(\mathrm{lb}_I(\alpha'))$, puis $\varphi_I(\alpha) = \varphi_I(\alpha')$.

◦ *Si* $\alpha = \ell \in I$ *alors* $\varphi_I(\alpha) = \ell$ **(P_7)**. Considérons une variable α de sorte Atom, telle que $\alpha = \ell \in I$ **(1)**. Puisque I est bien marquée, on a $\alpha = \alpha \in I$ donc, par ATM-EQ, $\alpha \leq \alpha \in^* I$ **(2)**. On déduit de (1) et (2) que $\ell \in \mathrm{lbs}_I(\alpha)$, et donc que $\ell \leq \mathrm{lb}_I(\alpha)$ **(3)**. Inversement, soit $\ell' \in \mathrm{lbs}_I(\alpha)$ **(4)**. Il existe une variable β de sorte Atom telle que $\beta = \ell' \in I$ **(5)** et $\beta \leq \alpha \in^* I$ **(6)** ou $\beta \lessdot \alpha \in^* I$ **(7)**. Puisque α et β sont de sortes Atom, par ATM-ATOM-GD, (7) implique (6). En appliquant l'hypothèse (P_2) à (5), (6) et (1), on obtient $\ell' \leq_{\mathcal{L}} \ell$. En déchargeant l'hypothèse (4), on conclut que $\forall \ell' \in \mathrm{lbs}_I(\alpha)\;\; \ell' \leq \ell$, d'où $\mathrm{lb}_I(\alpha) \leq \ell$ **(8)**. En combinant (3) et (8), on obtient finalement $\mathrm{lb}_I(\alpha) = \ell$, c'est-à-dire, puisque α est de sorte Atom, $\varphi_I(\alpha) = \ell$. ⌟

La démonstration de ce théorème permet d'obtenir un algorithme pour déterminer la satisfiabilité d'une contrainte atomique, en temps linéaire vis à vis sa taille. Il suffit pour cela de faire deux observations à propos de la solution φ_I construite pour la preuve. Tout d'abord, puisque nous avons montré que si I vérifie l'hypothèse (P_2) alors φ_I est une solution de I, il est immédiat que la contrainte I est satisfiable *si et seulement si* elle est satisfaite par φ_I, *i.e.* si et seulement si φ_I vérifie les propriétés (P_4) à (P_7). De plus, seule la preuve de (P_7) utilise l'hypothèse (P_2), autrement dit les propriétés (P_4), (P_5) et (P_6) sont vraies par construction de φ_I. En conclusion, pour déterminer si la contrainte I est satisfiable, il suffit de construire l'affectation φ_I et de déterminer si elle vérifie la propriété (P_7). Ce dernier test étant immédiat, il me reste à expliquer comment calculer, en temps linéaire, la valeur de φ_I pour chaque variable de type libre dans I.

Pour cela, on considère le graphe G dont les nœuds sont les variables de type libres dans I, et les arcs donnés par les inégalités et gardes dans I (*i.e.* un arc de α vers β existe dans G si et

seulement si $\alpha \leq \beta \in I$ ou $\alpha \lessdot \beta \in I$). On détermine ensuite les composantes fortement connexes du graphe G, puis on construit le graphe quotient G', dont chaque nœud $\bar{\alpha}$ est une composante fortement connexe de G. Étant donné deux composantes $\bar{\alpha}$ et $\bar{\beta}$, un arc de $\bar{\alpha}$ vers $\bar{\beta}$ est présent dans G' si et seulement si il existe $\alpha \in \bar{\alpha}$ et $\beta \in \bar{\beta}$ tels qu'il y ait un arc de α vers β dans G. Le graphe G' est acyclique, on peut donc effectuer son tri topologique. Cela permet de calculer pour chaque composante $\bar{\alpha}$ la borne inférieure commune à tous ses membres en utilisant l'égalité suivante :

$$\mathrm{lb}_I(\bar{\alpha}) = \bigsqcup \big(\{\, \ell \mid \exists \alpha \in \bar{\alpha} \;\; \alpha = \ell \in I \,\} \cup \{\, \mathrm{lb}_I(\bar{\beta}) \mid \bar{\beta} \text{ prédécesseur de } \bar{\alpha} \text{ dans } G' \,\} \big)$$

dans un parcours du graphe G' dans l'ordre exhibé par le tri topologique. Ce calcul peut être réalisée en temps linéaire, à condition que les opérations d'union et de comparaison de deux atomes s'effectuent en temps constant.

10.2.5 Étude de la complexité

Il est intéressant de faire un parallèle entre la stratégie de résolution des contraintes que j'ai adoptée, et l'algorithme de Heintze et McAllester pour l'analyse de flots de contrôle (CFA) [HM97]. Tandis que, dans l'approche standard pour l'analyse de flots, un clôture transitive est dynamiquement entrelacée avec la génération des contraintes, Heintze et McAllester ont proposé une procédure qui commence par construire un certain graphe, puis effectue dans un deuxième temps une clôture transitive « à la demande ». Cela donne un algorithme en temps *linéaire* sous l'hypothèse de types de taille bornée [McA96]. De la même manière, l'algorithme que j'ai décrit ci-avant pour le sous-typage structurel retarde la clôture autant que possible en commençant par expanser la structure des types et en décomposant les inégalités jusqu'à obtenir un problème atomique. L'hypothèse des types de taille bornée peut être discutée [SHO98], mais elle capture l'intuition que, dans les exemples de programme concrets, les fonctions ont généralement un nombre d'arguments et un ordre restreint. Cette observation a été particulièrement utile pour comprendre le comportement linéaire observé pour l'inférence de type de ML [Hen93, McA03].

Je montre maintenant de manière informelle que la complexité en temps du corps de l'algorithme de résolution est *quasi-linéaire*. Pour simplifier les choses, j'exclus les rangées de mon propos : leur analyse est en effet assez délicate [Pot03]. L'entrée de l'algorithme, une contrainte I, est mesurée par sa taille n qui est la somme des tailles de tous ses types. Je note I' un résultat produit par l'algorithme pour l'entrée I. Comme je l'ai expliqué, je fais l'hypothèse que la taille des types est bornée : soit h la hauteur de la plus haute structure exhibée par le test d'occurrence, *i.e.* la longueur du plus long chemin dans le graphe défini par $\prec_{I'}$. Je note également a l'arité maximale des constructeurs de type.

La première étape de l'algorithme est la combinaison de deux algorithmes d'unification, qui peuvent être mis en œuvre séparément en considérant d'abord les multi-squelettes puis les multi-équations. Ainsi, en utilisant un algorithme de « recherche-union » (*union-find* en anglais), leur coût est un $O(n\alpha(n))$, où α est une fonction reliée à l'inverse de la fonction d'Ackermann [Tar75]. Le test d'occurrence est équivalent à un tri topologique du graphe induit par les descripteurs sur les multi-squelettes, de telle sorte qu'il peut être réalisé en temps linéaire vis-à-vis de leur nombre, qui est un $O(n)$. Ensuite, sous l'hypothèse de types de hauteur bornée, l'expansion génère au plus a^h nouvelles variables de type pour chaque multi-équation présente dans le problème initial. La décomposition d'une inégalité ou d'une garde est bornée de manière similaire. On en déduit que le coût de ces deux étapes est un $O(\mathsf{a}^\mathsf{h} n)$. Enfin, comme je l'ai expliqué à la section 10.2.4 (page 192), le test de satisfiabilité d'une contrainte atomique peut être réalisé en temps linéaire en la taille de cette contrainte. Or, après l'expansion et la décomposition, la taille de la partie atomique de la contrainte manipulée par l'algorithme est également un $O(\mathsf{a}^\mathsf{h} n)$. On en déduit finalement que la coût de la procédure de résolution est un $O(\mathsf{a}^\mathsf{h} n)$, c'est-à-dire $O(n)$ sous l'hypothèse de types de hauteur bornée.

$\otimes$	$\varnothing$	$+$	$-$	$\pm$
$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
$+$	$\varnothing$	$+$	$-$	$\pm$
$-$	$\varnothing$	$-$	$+$	$\pm$
$\pm$	$\varnothing$	$\pm$	$\pm$	$\pm$

Figure 10.6 – Définition de la composition de polarités

10.3 Heuristiques de simplification des contraintes

La description du corps de l'algorithme de résolution est à présent achevée. Dans les paragraphes suivants, j'introduis au sein de ce processus un certain nombre d'heuristiques dont le but principal est de réduire la taille du problème au cours de son traitement, et ainsi d'en améliorer l'efficacité. La simplification de contraintes de sous-typage est un problème subtil : elle doit être sémantiquement correcte et complète (*i.e.* préserver la sémantique des contraintes), et le coût de sa mise en œuvre, aussi bien en temps qu'en espace, doit rester aussi faible que possible pour se montrer rentable, c'est-à-dire être largement compensé par le gain obtenu au niveau du processus de résolution. Poursuivant mon approche pragmatique, je ne m'intéresse pas ici à la définition d'un critère de minimalité relatif à la taille des contraintes obtenues après simplification : outre les difficultés théoriques afférentes, l'obtention d'une telle propriété serait probablement coûteuse en pratique. Au contraire, je présente une série d'heuristiques dont le comportement se révèle très efficace : finement combinées avec le corps de l'algorithme présenté à la section précédente, elles permettent d'accroître notablement son efficacité.

La partie la plus sensible du processus de résolution est la phase d'expansion, qui est théoriquement susceptible de créer un nombre exponentiel de variables de type. Comme je l'ai expliqué à la section 10.2.3 (page 184), l'expansion et la décomposition peuvent être réalisées successivement dans chaque multi-squelette, en considérant ceux-ci dans l'ordre exhibé par le test d'occurrence. C'est pourquoi, dans le but de limiter le nombre de variables de types introduites, j'applique les heuristiques de simplification qui sont *locales* (section 10.3.2, page 199) à un multi-squelette juste avant son expansion. Ces phases de résolution et de simplification doivent donc s'exécuter de manière entrelacée. Un deuxième groupe de simplifications (section 10.3.3, page 209) nécessitent quant à elles de considérer le graphe formé par les inégalités et les gardes dans son ensemble. Par conséquent, elles sont réalisées seulement au terme de la résolution, de manière à réduire la taille du résultat final, ce qui est essentiel pour deux raisons. Tout d'abord, cela permet de présenter des informations de type concises et compréhensibles à l'utilisateur. D'autre part, comme nous le verrons au chapitre 11 (page 217), le traitement des formes d'introduction et d'instanciation de schémas de types dans le solveur complet nécessite d'effectuer des copies des contraintes primaires. Ces dernières doivent donc préalablement être rendues aussi compactes que possible.

En quelques mots, la taille d'une contrainte peut être réduite de deux manières. La première, mise en œuvre dans les heuristiques d'élimination des cycles et des chaînes (section 10.3.2, page 199), la minimisation (section 10.3.3, deuxième paragraphe, page 212) et le *hash-consing* (section 10.3.3, troisième paragraphe, page 215) consiste à identifier des variables de type. Cela revient en général à remplacer des inégalités par des équations, ce qui est effectif car le traitement des secondes est beaucoup plus économique — en temps comme en espace — que celui des premières. La seconde méthode — réalisée par le dépoussiérage (section 10.3.2, troisième paragraphe, page 202) et la clôture polarisée (section 10.3.3, premier paragraphe, page 209) — élimine des variables de type intermédiaires, et les fragments de contraintes associés, qui ne sont pas utiles pour le résultat final, par exemple car elles sont inaccessibles.

$$\text{POL-EMPTY}\ \frac{\forall \alpha \in \mathcal{V}\ \ \Pi(\alpha) = \pm}{\Pi \vdash [\,]} \qquad \text{POL-AND}\ \frac{\Pi \vdash \mathbb{C} \qquad \forall \alpha \in \mathrm{ftv}(C)\ \ \Pi(\alpha) = \pm}{\Pi \vdash \mathbb{C}[[\,] \wedge C]} \qquad \text{POL-EXISTS}\ \frac{\Pi[\vec{\alpha} \mapsto \vec{\pi}] \vdash \mathbb{C}}{\Pi \vdash \mathbb{C}[\exists \bar{\alpha}.[\,]]}$$

$$\text{POL-LET}\ \frac{\begin{array}{c}\forall \alpha \in \mathrm{ftv}^+(\tau)\ \ + \in \Pi(\alpha) \qquad \forall \alpha \in \mathrm{ftv}^-(\tau)\ \ - \in \Pi(\alpha) \\ \Pi[\vec{\alpha} \mapsto \vec{\pi}] \vdash \mathbb{C} \qquad \forall \alpha \in \mathrm{ftv}(C)\ \ \Pi[\vec{\alpha} \mapsto \vec{\pi}](\alpha) = \pm\end{array}}{\Pi \vdash \mathbb{C}[\mathsf{let}\, x\ :\ \forall \bar{\alpha}[[\,]].\tau\ \mathsf{in}\ C]} \qquad \text{POL-IN}\ \frac{\Pi \vdash \mathbb{C}' \qquad \forall \alpha \in \mathrm{ftv}(\sigma)\ \ \Pi(\alpha) = \pm}{\Pi \vdash \mathbb{C}'[\mathsf{let}\, x\ :\ \sigma\ \mathsf{in}\ [\,]]}$$

Figure 10.7 – Polarités des variables de type dans les contextes

10.3.1 Polarités

Une ***polarité*** (notée : π) est une partie (possiblement vide) de $\{-, +\}$. (Les variances forment donc un sous-ensemble des polarités.) Les quatre polarités sont notées $\varnothing$ (lire : *apolaire*), $+$ (*positive*), $-$ (*négative*) et $\pm$ (*bipolaire*). Une ***polarisation*** (notée : Π) est une fonction totale des variables de type vers les polarités. Par abus de notation, j'écris $\Pi(\bar{\alpha})$ pour $\bigcup_{\alpha \in \bar{\alpha}} \Pi(\alpha)$. L'opérateur de composition des variances $\otimes$ est étendu aux polarités comme indiqué par la figure 10.6. Étant donnés deux types bruts t_1 et t_2 de même sorte et une polarité π, je définis $t_1 \leq^\pi t_2$ par :

$$t_1 \leq^\pi t_2 \Leftrightarrow \begin{cases} + \in \pi \ \Rightarrow\ t_1 \leq t_2 \\ - \in \pi \ \Rightarrow\ t_2 \leq t_1 \end{cases}$$

En d'autres termes, $\leq^\varnothing$ désigne la relation toujours vraie, $\leq^+$ correspond à $\leq$, $\leq^-$ à $\geq$ et $\leq^\pm$ à $=$. J'étends cette notation aux affectations et aux polarisations en définissant $\varphi_1 \leq^\Pi \varphi_2$ comme équivalent à $\forall \alpha \in \mathcal{V}\ \ \varphi_1(\alpha) \leq^{\Pi(\alpha)} \varphi_2(\alpha)$.

La plupart des techniques de simplification présentées dans les sections suivantes nécessitent de prendre en compte le contexte $\mathbb{C}$ dans lequel apparaît la contrainte traitée I. En effet, la connaissance de ce dernier permet d'utiliser des stratégies de simplification plus puissantes : pour que la simplification de I en I' soit logiquement valide, il suffit de respecter $\mathbb{C}[I] \equiv \mathbb{C}[I']$, et non plus nécessairement $I \equiv I'$. De manière à la rendre exploitable pour guider les heuristiques de simplification, l'information portée par le contexte est approximée en introduisant une polarité pour chaque variable de type, grâce à une polarisation Π. Une variable de type α telle que $+ \in \Pi(\alpha)$ est dite *positive*. Intuitivement, il s'agit d'une variable sur laquelle le contexte peut poser des bornes *supérieures* (comme, par exemple, une inégalité $\alpha \leq \beta$). On est donc intéressé par sa borne *inférieure* à l'intérieur du contexte. Inversement, si $+ \notin \Pi(\alpha)$, on dit que α est *non-positive* : dans ce cas, le contexte courant ne peut donner de borne *supérieure* à α, de telle sorte que les contraintes portant sur sa borne *inférieure* à l'intérieur du contexte peuvent être éliminées. De manière symétrique, une variable de type α telle que $- \in \Pi(\alpha)$ est dite négative et ne peut recevoir de contrainte sur sa borne inférieure dans le contexte ; une variable α telle que $- \notin$ est *non-négative*. On peut voir les polarités des variables de types dans un contexte comme un raffinement orienté de la notion d'accessibilité : une variable apparaissant à l'intérieur d'un contexte est accessible depuis l'extérieur si et seulement si elle n'est pas apolaire.

Les polarisations permettent d'introduire des formes relaxées des notions d'implication et d'équivalence logique entre contraintes.

Définition 10.34 (Implication et équivalence modulo) *Soient C_1 et C_2 deux contraintes, Π une polarisation. C_1* **implique** *C_2* **modulo** *Π (noté : $C_1 \Vdash C_2 \mod \Pi$) si et seulement si, pour tous φ_1 et X tels que $\varphi_1, X \vdash C_1$, il existe φ_2 tel que $\varphi_2, X \vdash C_2$ et $\varphi_2 \leq^\Pi \varphi_1$. On dit que C_1 et C_2 sont* **équivalentes modulo** *Π (noté : $C_1 \equiv C_2 \mod \Pi$) si et seulement $C_1 \Vdash C_2 \mod \Pi$ et $C_2 \Vdash C_1 \mod \Pi$.* □

Intuitivement, la contrainte C_1 implique C_2 modulo Π si et seulement si, pour toute solution

de la première, on peut construire une solution de la seconde, en augmentant (resp. diminuant) éventuellement l'affectation des variables de type non-positives (resp. non-négatives).

Les polarités des variables de types dans un contexte peuvent être déterminées grâce aux règles de la figure 10.7 : on dit que la polarisation Π est valide dans le contexte $\mathbb{C}$ si et seulement si le jugement $\Pi \vdash \mathbb{C}$ est dérivable. Je commente maintenant ces règles, en donnant pour chacune d'elles un lemme qui formule sa correction en termes d'implication de contraintes modulo. La première règle, POL-EMPTY, exprime le fait que, dans un contexte vide, toutes les variables de types sont accessibles par leurs deux bornes. Elles doivent donc être considérées comme bipolaires.

Lemme 10.35 *Supposons* $\forall \alpha \in \mathcal{V}\ \Pi(\alpha) = \pm$. *Si* $C_1 \Vdash C_2 \mod \Pi$ *alors* $C_1 \Vdash C_2$. □

Les autres règles examinent les contextes non vides en commençant par leur extrémité (*i.e.* leur trou). La règle POL-AND s'applique à un contexte se terminant par une conjonction : la contrainte C étant arbitraire, ses variables libres doivent être considérées comme bipolaires. La polarité des autres variables de type doit être fixée d'une manière valide pour la partie supérieure du contexte, $\mathbb{C}$.

Lemme 10.36 *Supposons* $\forall \alpha \in \mathrm{ftv}(C)\ \Pi(\alpha) = \pm$. *Si* $C_1 \Vdash C_2 \mod \Pi$ *alors* $C_1 \wedge C \Vdash C_2 \wedge C \mod \Pi$. □

⌜ *Preuve.* Supposons $C_1 \Vdash C_2 \mod \Pi$ **(H₁)** et $\forall \alpha \in \mathrm{ftv}(C)\ \Pi(\alpha) = \pm$ **(H₂)**. Soient φ_1 et X tels que $\varphi_1, X \vdash C_1 \wedge C$ **(1)**. On a $\varphi_1, X \vdash C_1$ donc, par (H_1), il existe φ_2 tel que $\varphi_2, X \vdash C_2$ **(2)** et $\varphi_2 \leq^{\Pi} \varphi_1$ **(3)**. On déduit de (H_2) et (3) que, pour tout $\alpha \in \mathrm{ftv}(C)$, $\varphi_1(\alpha) = \varphi_2(\alpha)$. Par (1), cela implique $\varphi_2, X \vdash C$ **(4)**. En combinant (2) et (4), on obtient $\varphi_2, X \vdash C_2 \wedge C$ **(5)**. En déchargeant l'hypothèse (1) sur (5) et (3), on en déduit le but : $C_1 \wedge C \Vdash C_2 \wedge C \mod \Pi$. ⌟

La règle POL-EXISTS concerne un contexte se terminant par un quantificateur existentiel : les variables $\bar{\alpha}$ étant locales au trou du contexte, leur polarité peut être fixée indépendamment du contexte supérieur $\mathbb{C}$ pour lequel elles sont inaccessibles.

Lemme 10.37 *Si* $C_1 \Vdash C_2 \mod \Pi$ *alors* $\exists \bar{\alpha}.C_1 \Vdash \exists \bar{\alpha}.C_2 \mod \Pi[\vec{\alpha} \mapsto \vec{\pi}]$. □

⌜ *Preuve.* Supposons $C_1 \Vdash C_2 \mod \Pi$ **(H₁)**. Soient φ_1 et X tels que $\varphi_1, X \vdash \exists \bar{\alpha}.C_1$ **(1)**. Il existe φ_1' tel que $\varphi_1 = \varphi_1'[\vec{\alpha} \mapsto \vec{t}]$ et $\varphi_1', X \vdash C_1$. Par (H_1), on en déduit qu'il existe φ_2' tel que $\varphi_2', X \vdash C_2$ **(2)** et $\varphi_2' \leq^{\Pi} \varphi_1'$ **(3)**. Posons $\varphi_2 = \varphi_2'[\vec{\alpha} \mapsto \vec{t}]$. Par (2), on a $\varphi_2, X \vdash \exists \bar{\alpha}.C_2$ **(4)**. De plus, puisque φ_1 et φ_2 coïncident sur $\bar{\alpha}$, (3) implique $\varphi_2 \leq^{\Pi[\vec{\alpha} \mapsto \vec{\pi}]} \varphi_1$ **(5)**. En déchargeant l'hypothèse (1) sur (4) et (5), on en déduit le but : $\exists \bar{\alpha}.C_1 \Vdash \exists \bar{\alpha}.C_2 \mod \Pi[\vec{\alpha} \mapsto \vec{\pi}]$. ⌟

Les deux premières prémisses de la règle POL-LET concernent le type τ : sa borne supérieure peut être contrainte par d'éventuelles formes d'instanciation portant sur x dans C. Les variables de type libres dans τ doivent donc recevoir des polarités conformément aux signes de leurs occurrences. Les deux dernières prémisses de POL-LET concernent la partie supérieure du contexte et la contrainte C. Elles sont comparables à celles de POL-AND, étant entendu que les variables $\bar{\alpha}$ sont locales au schéma.

Lemme 10.38 *Supposons* $\forall \alpha \in \mathrm{ftv}^+(\tau)\ + \in \Pi(\alpha)$, $\forall \alpha \in \mathrm{ftv}^-(\tau)\ - \in \Pi(\alpha)$ *et* $\forall \alpha \in \mathrm{ftv}(C)\ \Pi[\vec{\alpha} \mapsto \vec{\pi}](\alpha) = \pm$. *Si* $C_1 \Vdash C_2 \mod \Pi$, *alors* $\mathsf{let}\, x : \forall \bar{\alpha}[C_1].\tau\ \mathsf{in}\ C \Vdash \mathsf{let}\, x : \forall \bar{\alpha}[C_2].\tau\ \mathsf{in}\ C \mod \Pi[\vec{\alpha} \mapsto \vec{\pi}]$. □

⌜ *Preuve.* Supposons $C_1 \Vdash C_2 \mod \Pi$ **(H₁)**, $\forall \alpha \in \mathrm{ftv}^+(\tau)\ + \in \Pi(\alpha)$ **(H₂)**, $\forall \alpha \in \mathrm{ftv}^-(\tau)\ - \in \Pi(\alpha)$ **(H₃)** et $\forall \alpha \in \mathrm{ftv}(C)\ \Pi[\vec{\alpha} \mapsto \vec{\pi}](\alpha) = \pm$ **(H₄)**. Soient φ_1 et X tels que $\varphi_1, X \vdash \mathsf{let}\, x : \forall \bar{\alpha}[C_1].\tau\ \mathsf{in}\ C$ **(1)**. On a $\varphi_1, X \vdash \forall \bar{\alpha}[C_1].\tau \preceq s$ **(2)** et $\varphi_1, X[x \mapsto s] \vdash C$ **(3)**. Soit $t \in s$ **(4)**. Par (2), il existe φ_1' tel que $\varphi_1 = \varphi_1'[\vec{\alpha} \mapsto \vec{t}]$ **(5)**, $\varphi_1', X \vdash C_1$ **(6)** et $\varphi_1'(\tau) \leq t$ **(7)**. Par (H_1) et (6), il existe φ_2' tel que $\varphi_2', X \vdash C_2$ **(8)** et $\varphi_2' \leq^{\Pi} \varphi_1'$ **(9)**. Par (H_2), (H_3) et la propriété 1.7 (page 25), (7) implique

$\varphi'_2(\tau) \leq \varphi'_1(\tau)$. Par (7), on en déduit que $\varphi'_2(\tau) \leq t$ **(10)**. Soit $\varphi_2 = \varphi'_2[\vec{\alpha} \mapsto \vec{t}]$ **(11)**. Par (8) et (10), on a $\varphi_2, X \vdash \forall\bar{\alpha}[C_2].\tau \preceq t$. En déchargeant l'hypothèse (4), on en déduit que $\varphi_2, X \vdash \forall\bar{\alpha}[C_2].\tau \preceq s$ **(12)**. Par (5) et (11), φ_1 et φ_2 coïncident sur $\bar{\alpha}$ et sont respectivement identiques à φ'_1 et φ'_2 sur $\mathcal{V}\backslash\bar{\alpha}$. Cela donne, grâce à (9), $\varphi_2 \leq^{\Pi[\vec{\alpha}\mapsto\vec{\pi}]} \varphi_1$ **(13)**. Par (H_4), on en déduit que φ_1 et φ_2 coïncident sur $\mathrm{ftv}(C)$, de telle sorte que (3) implique $\varphi_2, X[x \mapsto s] \vdash C$ **(14)**. Par une instance de C-LET, les jugements (12) et (14) donnent $\varphi_2, X \vdash \mathsf{let}\, x : \forall\bar{\alpha}[C_2].\tau \;\mathsf{in}\; C$ **(15)**. En déchargeant l'hypothèse (1) sur (15) et (13), on obtient le but : $\mathsf{let}\, x : \forall\bar{\alpha}[C_1].\tau \;\mathsf{in}\; C \Vdash \mathsf{let}\, x : \forall\bar{\alpha}[C_2].\tau \;\mathsf{in}\; C \mod \Pi[\vec{\alpha} \mapsto \vec{\pi}]$. ⌟

Enfin, la règle POL-IN traite les contextes se terminant par une forme $\mathsf{let}\, x : \sigma \;\mathsf{in}\; [\,]$, de manière similaire à POL-AND.

Lemme 10.39 *Si* $C_1 \Vdash C_2 \mod \Pi$ *et* $\forall\alpha \in \mathrm{ftv}(\sigma)\ \Pi(\alpha) = \pm$ *alors* $\mathsf{let}\, x : \sigma \;\mathsf{in}\; C_1 \Vdash \mathsf{let}\, x : \sigma \;\mathsf{in}\; C_2 \mod \Pi$. □

L'affaiblissement d'une polarisation préserve sa validité pour un contexte : si $\Pi_1 \vdash \mathbb{C}$ et, pour tout $\alpha \in \mathcal{V}$, $\Pi_1(\alpha) \subseteq \Pi_2(\alpha)$ alors $\Pi_2 \vdash \mathbb{C}$ est également dérivable. De plus, la polarisation qui considère toutes les variables de type comme bipolaires est valide pour tous les contextes. On en déduit que chaque contexte $\mathbb{C}$ admet une polarisation valide *minimale* (au sens de l'inclusion point à point) ; laquelle peut être déterminée par la résolution d'une formule booléenne dont les variables logiques correspondent aux polarités des variables de type définies par $\mathbb{C}$. Par ailleurs, la deuxième prémisse de la règle POL-AND, la quatrième prémisse de POL-LET et la première prémisse de POL-IN sont relativement approximatives : elles considèrent toute variable libre de la contrainte C (ou du schéma σ) comme bipolaire, quelle que soit la façon dont elle est employée. Il serait possible de raffiner ce critère de manière à capturer des informations sur la contrainte C. Cependant, il est généralement préférable en pratique de simplifier autant que possible le calcul des polarités, en particulier de manière à ce qu'il n'induise pas de surcoût à l'exécution. Comme cela apparaîtra au chapitre 11 (page 217), un compromis intéressant consiste à arrêter l'exploration du contexte à la première forme $\mathsf{let}\, x : \forall\bar{\alpha}[[\,]].\tau \;\mathsf{in}\; C$ rencontrée, en considérant toutes les variables non dans $\bar{\alpha}$ comme bipolaires.

Le théorème suivant montre, que sous un contexte $\mathbb{C}$ de polarisation Π, il est possible de considérer l'implication ou l'équivalence modulo Π, pour obtenir extérieurement l'implication ou l'équivalence au sens habituel.

Théorème 10.40 (Polarités d'un contexte) *Si* $\Pi \vdash \mathbb{C}$ *et* $C_1 \Vdash C_2 \mod \Pi$ *alors* $\mathbb{C}[C_1] \Vdash \mathbb{C}[C_2]$. □

⌜ *Preuve.* Par induction sur la dérivation de $\Pi \vdash \mathbb{C}$, en utilisant les lemmes 10.35, 10.36, 10.37, 10.38 et 10.39. ⌟

La notion de polarité comme raffinement de l'accessibilité a été introduite par les travaux de Fuh et Mishra [FM89], puis reprise par Trifonov et Smith [TS96], et Pottier [Pot01b]. Ces derniers ne s'intéressent cependant qu'à des schémas de type clos, où toutes les variables de type sont universellement quantifiées. Cela donne lieu à une définition légèrement moins générale que celle présentée ici, qui rattache les polarités aux *contextes* du langage de contraintes.

Les définitions précédentes indiquent comment les polarités des variables de types peuvent être extraites du contexte dans lequel la contrainte à simplifier est placée. Je montre maintenant comment ces polarités doivent être propagées à travers la structure des types exhibée par l'expansion. Comme je l'ai expliqué à la section 10.2.3 (page 184), on souhaite appliquer certaines heuristiques de simplification tout au long de la phase d'expansion et de décomposition, sur chaque multi-squelette *avant* de lui appliquer les règles de la figure 10.3 (page 185). En faisant abstraction des quantificateurs existentiels (lesquels peuvent être considérés comme faisant partie du contexte $\mathbb{C}$, puisqu'ils sont remontés au sommet de la contrainte par SPE-EX-AND lors de l'expansion), à chaque étape de la procédure d'expansion et de décomposition, la contrainte résolue peut être écrite sous la forme $I_s \wedge I$ où I_s est une contrainte structurelle, correspondant à la partie déjà expansée et

décomposée, et I est la partie restant à traiter, dont les variables libres ne sont pas dans le support de I_s. Comme énoncé par le lemme 10.28 (page 190), lorsque l'expansion et la décomposition sont terminées, I est une contrainte atomique. La définition et le théorème qui suivent montrent comment propager les polarités extraites du contexte le long de la structure des types donnée par I_s, de manière à pouvoir simplifier directement I.

Définition 10.41 *Étant données une polarisation* Π *et une contrainte structurelle* I_s, $\Pi \otimes I_s$ *est la plus petite polarisation* Π' *telle que pour tout* $\alpha \in \mathcal{V}$, $\Pi(\alpha) \subseteq \Pi'(\alpha)$, *et pour tous* $\beta = d\,\beta_1 \cdots \beta_n \in I_s$, *et* $i \in [1, n]$, $\Pi'(\beta) \otimes d.i \subseteq \Pi'(\beta_i)$. □

Si une variable β a pour descripteur $d\,\beta_1 \cdots \beta_n$ dans I_s alors toute nouvelle borne (inférieure ou supérieure) sur β est susceptible de se répercuter, par décomposition, sur les sous-termes $\beta_1, \ldots, \beta_n$. Ainsi, si β est positive pour Π alors chaque sous-terme doit être positif pour $\Pi \otimes I_s$ s'il est à une position covariante ou invariante de d, et négatif s'il est en position contravariante ou invariante. Inversement, si β est négative alors un sous-terme devient positif s'il est en position contravariante ou invariante, et négatif s'il est en position covariante ou invariante.

Théorème 10.42 (Polarités et structure) *Soient* I_s *une contrainte structurelle acyclique de support* $\bar{\alpha}$ *et* Π *une polarisation. Soient* I_1 *et* I_2 *deux contraintes telles que* $\bar{\alpha} \# \mathrm{ftv}(I_1, I_2)$ *et* $I_s \wedge I_2$ *soit unifiée. Si* $I_1 \Vdash I_2 \mod (\Pi \otimes I_s)$ *alors* $I_s \wedge I_1 \Vdash I_s \wedge I_2 \mod \Pi$. □

⌜ *Preuve.* Soient I_1 et I_2 deux contraintes telles que $\bar{\alpha} \# \mathrm{ftv}(I_1, I_2)$, et $I_s \wedge I_2$ soit unifiée. Soit $\Pi' = \Pi \otimes I_s$; on a, pour tout $\alpha \in \mathcal{V}$, $\Pi(\alpha) \subseteq \Pi'(\alpha)$ **(1)**, et pour tous $\beta = d\,\beta_1 \cdots \beta_n \in I_s$, et $i \in [1, n]$, $\Pi'(\beta) \otimes d.i \subseteq \Pi'(\beta_i)$ **(2)**. Supposons $I_1 \Vdash I_2 \mod \Pi'$ **(3)**.

Soit φ_1 une affectation telle que $\varphi_1 \vdash I_s \wedge I_1$ **($\mathbf{H_1}$)** ; je dois montrer qu'il existe φ_2' telle que $\varphi_2' \vdash I_s \wedge I_2$ **($\mathbf{C_1}$)** et $\varphi_2' \leq^{\Pi} \varphi_1$ **($\mathbf{C_2}$)**. L'hypothèse (H_1) implique $\varphi_1 \vdash I_1$; par (3), on en déduit qu'il existe φ_2 telle que $\varphi_2 \vdash I_2$ **(4)** et $\varphi_2 \leq^{\Pi'} \varphi_1$ **(5)**. La contrainte $I_s \wedge I_2$ étant unifiée, j'utilise le même procédé que dans la preuve du théorème 10.30 (page 191) pour construire, à partir de φ_2, une solution φ_2' de $I_s \wedge I_2$ telle que :

$$\varphi_2'(\alpha) = \begin{cases} \varphi_2(\alpha) & \text{si } \alpha \notin \bar{\alpha} \\ d\,\varphi_2'(\alpha_1) \cdots \varphi_2'(\alpha_n) & \text{si } \alpha \in \bar{\alpha} \text{ et } \alpha = d\,\alpha_1 \cdots \alpha_n \in I_s \end{cases}$$

Le but (C_1) s'obtient de la même manière qu'au théorème 10.30 (page 191). Il me reste à montrer (C_2). Je considère pour cela une variable α. Je prouve par induction sur l'ordre $\prec_{I_s}$, qui est bien fondé puisque I_s est supposée acyclique, que $\varphi_2'(\alpha) \leq^{\Pi'(\alpha)} \varphi_1(\alpha)$ **(6)**.

- *Si* $\alpha \notin \bar{\alpha}$, alors $\varphi_2'(\alpha) = \varphi_2(\alpha)$. On en déduit, par (5), que $\varphi_2'(\alpha) \leq^{\Pi'(\alpha)} \varphi_1(\alpha)$.
- *Si* $\alpha \in \bar{\alpha}$, puisque I_s est structurelle, il existe $d\,\alpha_1 \cdots \alpha_n$ telle que $\alpha = d\,\alpha_1 \cdots \alpha_n \in I_s$ **(7)**. Soit $i \in [1, n]$ **(8)**. Par (7), on a $\alpha_i \prec_{I_s} \alpha$, ce qui permet d'appliquer l'hypothèse d'induction à α_i et d'obtenir $\varphi_2'(\alpha_i) \leq^{\Pi'(\alpha_i)} \varphi_1(\alpha_i)$ **(9)**. Or (2) implique $\Pi'(\alpha) \otimes d.i \subseteq \Pi'(\alpha_i)$, de telle sorte que (9) implique $\varphi_2'(\alpha_i) \leq^{\Pi'(\alpha) \otimes d.i} \varphi_1(\alpha_i)$. En déchargeant (8), on en déduit que $d\,\varphi_2'(\alpha_1) \cdots \varphi_2'(\alpha_n) \leq^{\Pi'(\alpha)} d\,\varphi_1(\alpha_1) \cdots \varphi_1(\alpha_n)$. Puisque φ_1 et φ_2' satisfont I_s, par (7), les membres gauche et droit de cette inégalité sont respectivement égaux à $\varphi_2'(\alpha)$ et $\varphi_1(\alpha)$, ce qui permet de conclure : $\varphi_2'(\alpha) \leq^{\Pi'(\alpha)} \varphi_1(\alpha)$.

Par (6) et (1), on obtient le but (C_2). ⌟

10.3.2 Simplifications réalisées lors de l'expansion et de la décomposition

Cette section présente les techniques de simplification qui doivent être appliquées de manière conjuguée avec les processus d'expansion et de décomposition des contraintes. Elles sont introduites en définissant une nouvelle relation de réécriture sur les contraintes, notée $-\Pi\rightarrowtriangle_e$, qui étend $\rightarrowtriangle_e$. Cette relation est paramétrée par une polarisation décrivant le contexte dans lequel apparaît la

contrainte traitée. Elle est définie par quatre groupes de règles que j'introduis et explique dans les paragraphes suivants. Voici le premier d'entre eux :

$$I \ -\Pi\!\!\rightarrow_e \ I' \quad \textit{si } I \rightarrow_e I' \qquad \text{PE'-BASE}$$

$$\exists\bar{\alpha}.I \ -\Pi\!\!\rightarrow_e \ \exists\bar{\alpha}.I' \quad \textit{si } I \ -\Pi[\vec{\alpha}\mapsto\varnothing]\!\!\rightarrow_e I' \qquad \text{PE'-EXISTS}$$

$$I_s \wedge I \ -\Pi\!\!\rightarrow_e \ I_s \wedge I' \quad \textit{si } I_s \textit{ structurelle de support disjoint de } \mathrm{ftv}(I) \textit{ dépoussiérée pour } \Pi \textit{ et } I \ -\Pi\otimes I_s\!\!\rightarrow_e I' \qquad \text{PE'-STRUCT}$$

La règle PE'-BASE exprime l'inclusion de $\rightarrow_e$ dans $-\Pi\!\!\rightarrow_e$, de manière indépendante de la polarisation Π. Comme je l'ai expliqué précédemment, lorsque la règle SPE-EX-AND est appliquée avidement, la contrainte traitée par le processus d'expansion et de décomposition est de la forme $\mathbb{X}[I_s \wedge I]$ où I_s est une contrainte structurelle de support disjoint de $\mathrm{ftv}(I)$, qui correspond à la partie du problème déjà traitée. Les règles PE'-EXISTS et PE'-STRUCT sont des règles de contexte qui permettent de considérer le fragment I et de lui appliquer une des autres règles de simplification, avec une polarisation tenant compte du contexte $\mathbb{X}[I_s \wedge [\,]]$. La définition et le rôle de la condition « I_s *dépoussiérée pour* Π » apparaîtront ci-après avec le dépoussiérage, elles peuvent pour l'instant être laissées de côté. La règle PE'-EXISTS donne les mêmes polarités aux variables $\bar{\alpha}$ que POL-EXISTS. Dans la règle PE'-STRUCT, la partie I_s peut être mise de côté à condition de propager les polarités le long de la structure des types, comme indiqué par la définition 10.41 et le théorème 10.42.

▶ **Unification des cycles** La première technique de simplification est la seule qui n'utilise pas les polarités des variables de type : elle consiste simplement à éliminer les cycles d'inégalités. Dans une contrainte unifiée, toutes les variables de tels cycles appartiennent nécessairement au même multi-squelette ; ils peuvent donc être recherchés de manière indépendante dans chaque multi-squelette, juste avant son expansion, puis éliminés grâce à la règle suivante :

$$\begin{pmatrix} \langle\bar{\bar{\tau}}_1\rangle^{\iota_1} \approx \cdots \approx \langle\bar{\bar{\tau}}_n\rangle^{\iota_n} \approx \bar{\bar{\tau}} \wedge \\ \alpha_1 \leq \beta_2 \wedge \cdots \wedge \alpha_{n-1} \leq \beta_n \wedge \alpha_n \leq \beta_1 \end{pmatrix} \wedge I \ -\Pi\!\!\rightarrow_e \ \langle\bar{\bar{\tau}}_1 = \cdots = \bar{\bar{\tau}}_n\rangle^{\iota_1\vee\cdots\vee\iota_n} \approx \bar{\bar{\tau}} \wedge I \quad \textit{si } \forall i \in [1,n] \ \ \alpha_i, \beta_i \in \bar{\bar{\tau}}_i$$

Cette simplification identifie plusieurs multi-équations du multi-squelette considéré, réduisant ainsi le nombre potentiel de variables concernées par la phase d'expansion. Par là même, elle élimine une série d'inégalités, économisant ainsi leur décomposition à venir. Si elle est sémantiquement correcte, la règle donnée ci-avant ne respecte pas le caractère unifié de la contrainte réduite, dans le cas où plusieurs des multi-équations parmi $\bar{\bar{\tau}}_1, \ldots, \bar{\bar{\tau}}_n$ comportent un descripteur. Il est donc nécessaire de réaliser immédiatement une phase d'unification sur la contrainte obtenue après avoir éliminé le cycle, ce qui nécessite de corriger la règle précédente comme suit :

$$\begin{pmatrix} \langle\bar{\bar{\tau}}_1\rangle^{\iota_1} \approx \cdots \approx \langle\bar{\bar{\tau}}_n\rangle^{\iota_n} \approx \bar{\bar{\tau}} \wedge \\ \alpha_1 \leq \beta_2 \wedge \cdots \wedge \alpha_{n-1} \leq \beta_n \wedge \alpha_n \leq \beta_1 \end{pmatrix} \wedge I \ -\Pi\!\!\rightarrow_e \ I' \quad \textit{si } \forall i \in [1,n] \ \ \alpha_i, \beta_i \in \bar{\bar{\tau}}_i \textit{ et } \langle\bar{\bar{\tau}}_1 = \cdots = \bar{\bar{\tau}}_n\rangle^{\iota_1\vee\cdots\vee\iota_n} \approx \bar{\bar{\tau}} \wedge I \rightarrow_u^{**} I' \qquad \text{PE'-CYCLE}$$

Il faut noter que cet appel à la procédure d'unification se limite à fusionner des multi-équations appartenant déjà aux mêmes multi-squelettes, de telle sorte qu'il ne modifie pas la structure induite par ces derniers ni l'ordre $\prec_I$ exhibé par le test d'occurrence. Il ne peut en particulier produire d'erreur.

Dans leurs travaux sur les contraintes d'inclusion, Fähndrich *et al.* ont proposé un algorithme partiel dynamique de détection des cycles, qui permet d'éliminer les cycles d'un graphe de contraintes de manière incrémentale, au fur et à mesure de la génération d'arcs par l'algorithme de clôture. Il n'est cependant pas utile de mettre en œuvre un tel mécanisme dans une implémentation du solveur présenté dans ce chapitre : en effet, puisque les cycles sont internes aux multi-squelettes, ils

$$
\begin{aligned}
\mathrm{lhs}(\mathsf{true}) = \mathrm{lhs}(\mathsf{false}) &= \varnothing \\
\mathrm{lhs}(\tilde{\bar{\tau}}) &= \mathrm{ftv}(\tilde{\bar{\tau}}) \\
\mathrm{lhs}(\alpha \leq \beta) &= \{\alpha\} \\
\mathrm{lhs}(\alpha \lessdot \beta) &= \{\alpha\} \\
\mathrm{lhs}(I_1 \wedge I_2) &= \mathrm{lhs}(I_1) \cup \mathrm{lhs}(I_2) \\
\mathrm{lhs}(\exists \bar{\alpha}.I) &= \mathrm{lhs}(I) \backslash \bar{\alpha}
\end{aligned}
\qquad
\begin{aligned}
\mathrm{rhs}(\mathsf{true}) = \mathrm{rhs}(\mathsf{false}) &= \varnothing \\
\mathrm{rhs}(\tilde{\bar{\tau}}) &= \mathrm{ftv}(\tilde{\bar{\tau}}) \\
\mathrm{rhs}(\alpha \leq \beta) &= \{\beta\} \\
\mathrm{rhs}(\alpha \lessdot \beta) &= \{\beta\} \\
\mathrm{rhs}(I_1 \wedge I_2) &= \mathrm{rhs}(I_1) \cup \mathrm{rhs}(I_2) \\
\mathrm{rhs}(\exists \bar{\alpha}.I) &= \mathrm{rhs}(I) \backslash \bar{\alpha}
\end{aligned}
$$

Figure 10.8 – Variables libres dans un membre gauche ou droit

$$
\begin{array}{rcll}
\langle \alpha = \bar{\alpha} \rangle^{\bullet} \approx \langle \beta = \bar{\tau} \rangle^{\iota} \approx \tilde{\bar{\tau}} \wedge \alpha \leq \beta \wedge I & -\Pi\!\!\rightarrow_e & \langle \alpha = \bar{\alpha} = \beta = \bar{\tau} \rangle^{\iota} \approx \tilde{\bar{\tau}} \wedge I & \textsc{pe'-chain-left} \\
 & & \textit{si } \alpha\bar{\alpha} \# \mathrm{ftv}(\beta, \bar{\tau}, \tilde{\bar{\tau}}) \cup \mathrm{lhs}(I) & \\
 & & \textit{et } \Pi(\alpha\bar{\alpha}) \subseteq - & \\
\langle \alpha = \bar{\tau} \rangle^{\iota} \approx \langle \beta = \bar{\beta} \rangle^{\bullet} \approx \tilde{\bar{\tau}} \wedge \alpha \leq \beta \wedge I & -\Pi\!\!\rightarrow_e & \langle \alpha = \bar{\tau} = \beta = \bar{\beta} \rangle^{\iota} \approx \tilde{\bar{\tau}} \wedge I & \textsc{pe'-chain-right} \\
 & & \textit{si } \beta\bar{\beta} \# \mathrm{ftv}(\alpha, \bar{\tau}, \tilde{\bar{\tau}}) \cup \mathrm{rhs}(I) & \\
 & & \textit{et } \Pi(\beta\bar{\beta}) \subseteq + & \\
\alpha_1 \leq \beta_1 \wedge \alpha_2 \leq \beta_2 \wedge I & -\Pi\!\!\rightarrow_e & \alpha_1 \leq \beta_1 \wedge I & \textsc{pe'-stutter-}\leq \\
 & & \textit{si } \alpha_1 = \alpha_2 \in I \textit{ et } \beta_1 = \beta_2 \in I & \\
\alpha_1 \lessdot \beta_1 \wedge \alpha_2 \lessdot \beta_2 \wedge I & -\Pi\!\!\rightarrow_e & \alpha_1 \lessdot \beta_1 \wedge I & \textsc{pe'-stutter-}\lessdot \\
 & & \textit{si } \alpha_1 = \alpha_2 \in I \textit{ et } \beta_1 = \beta_2 \in I & \\
\alpha \lessdot \beta & -\Pi\!\!\rightarrow_e & \alpha \leq \beta & \textsc{pe'-atom-}\lessdot \\
 & & \textit{si } \alpha \textit{ et } \beta \textit{ de sorte } \mathsf{Atom} & \\
\alpha \leq \beta \wedge I & -\Pi\!\!\rightarrow_e & I & \textsc{pe'-reflex-}\leq \\
 & & \textit{si } \alpha = \beta \in I &
\end{array}
$$

Figure 10.9 – Réduction des chaînes

peuvent être recherchés de manière indépendante dans chacun d'entre eux, juste avant son expansion et la décomposition de ses inégalités et gardes. Il est ainsi suffisant d'utiliser des procédures habituelles de décomposition d'un graphe en composantes fortement connexes, comme l'algorithme de Tarjan [Tar72], qui sont de complexité linéaire.

▶ **Réduction des chaînes** Les systèmes de type à base de contraintes de sous-typage génèrent souvent un grand nombre de chaînes d'inégalités : puisque le sous-typage est autorisé de manière implicite en tous les points du programme, l'algorithme de génération de contraintes — comme celui que j'ai décrit pour MLIF($\mathcal{X}$) au chapitre 7 (page 135) — doit généralement produire une inégalité pour chacun d'entre-eux. Cependant, beaucoup de ces possibles coercions ne sont pas réellement utilisées par le programme, et peuvent en fait être remplacées par des égalités, grâce au processus de *réduction des chaînes*. Cette simplification se retrouve sous différentes formes dans la littérature [FM89, AW92, EST95].

Cette technique de simplification est spécifiée par les règles de la figure 10.9, et plus particulièrement PE'-CHAIN-LEFT et PE'-CHAIN-RIGHT, qui sont symétriques l'une de l'autre. La première considère une multi-équation $\alpha = \bar{\alpha}$ qui admet un *unique* majorant β : la condition $\alpha\bar{\alpha} \# \mathrm{lhs}(I)$ assure que les variables $\alpha\bar{\alpha}$ n'apparaissent que comme membres droits d'inégalités ou de gardes dans la contrainte I (l'ensemble $\mathrm{lhs}(I)$ est formellement défini figure 10.8). Les variables $\alpha\bar{\alpha}$ sont supposées non-positives, de telle sorte qu'elles ne peuvent recevoir, dans le contexte où la contrainte est placée, de nouveau majorant. On en déduit qu'il est possible, modulo la polarité Π, de fusionner la

multi-équation $\alpha = \bar{\bar{\alpha}}$ avec celle de son unique majorant $\beta = \bar{\bar{\tau}}$. La règle PE'-CHAIN-RIGHT procède de manière symétrique en unifiant une multi-équation de variables non-négatives avec leur unique minorant. Les autres règles, PE'-STUTTER-$\leq$, PE'-STUTTER-$\lessdot$, PE'-ATOM-$\lessdot$ et PE'-REFLEX-$\leq$ permettent d'éliminer des inégalités ou des gardes inutiles. Outre le fait qu'elles réduisent le nombre de contraintes à décomposer, elles sont susceptibles de faire apparaître des majorants ou minorants unique, et par là même des opportunités d'appliquer PE'-CHAIN-LEFT ou PE'-CHAIN-RIGHT.

Lemme 10.43 *Supposons $\varphi \vdash I$. Si $\alpha \notin \mathrm{lhs}(I)$ alors pour tout type brut t, $\varphi(\alpha) \leq t$ implique $\varphi[\alpha \mapsto t] \vdash I$. Si $\alpha \notin \mathrm{rhs}(I)$ alors pour tout type brut t, $t \leq \varphi(\alpha)$ implique $\varphi[\alpha \mapsto t] \vdash I$.* □

⌜ *Preuve.* Supposons $\varphi \vdash I$ **($\mathbf{H_1}$)**. Je considère le cas où $\alpha \notin \mathrm{lhs}(I)$ **($\mathbf{H_2}$)**, l'autre cas étant symétrique. Soit t tel que $\varphi(\alpha) \leq t$ **($\mathbf{H_3}$)**. Je montre par induction sur la contrainte I que $\varphi[\alpha \mapsto t] \vdash I$ **($\mathbf{C_1}$)**.

∘ *Cas I est* true. Le but (C_1) est une tautologie.

∘ *Cas I est* false. Ce cas ne peut intervenir, car il est incompatible avec l'hypothèse (H_1).

∘ *Cas I est $\bar{\bar{\tau}}$.* On a $\mathrm{lhs}(\bar{\bar{\tau}}) = \mathrm{ftv}(\bar{\bar{\tau}})$. Par ($\mathrm{H}_2$), α n'est pas dans $\mathrm{ftv}(\bar{\bar{\tau}})$. On en déduit que φ et $\varphi[\alpha \mapsto t]$ coïncident sur $\mathrm{ftv}(\bar{\bar{\tau}})$. Par ($\mathrm{H}_1$), on obtient le but ($\mathrm{C}_1$) : $\varphi[\alpha \mapsto t] \vdash \bar{\bar{\tau}}$.

∘ *Cas I est $\beta_1 \leq \beta_2$.* On a $\mathrm{lhs}(\beta_1 \leq \beta_2) = \{\beta_1\}$. Par ($\mathrm{H}_2$), α n'est pas β_1, donc $\varphi[\alpha \mapsto t](\beta_1) = \varphi(\beta_1)$ **(1)**, et par (H_3), on a $\varphi(\beta_2) \leq \varphi[\alpha \mapsto t](\beta_2)$ **(2)**. Or (H_1) donne $\varphi(\beta_1) \leq \varphi(\beta_2)$ **(3)**. Par transitivité, on déduit de (1), (3) et (2) que $\varphi[\alpha \mapsto t](\beta_1) \leq \varphi[\alpha \mapsto t](\beta_2)$, ce qui donne le but ($\mathrm{C}_1$) : $\varphi[\alpha \mapsto t] \vdash \beta_1 \leq \beta_2$.

∘ *Cas I est $\beta_1 \lessdot \beta_2$.* On a $\mathrm{lhs}(\beta_1 \leq \beta_2) = \{\beta_1\}$. Par ($\mathrm{H}_2$), α n'est pas β_1, donc $\varphi[\alpha \mapsto t](\beta_1) = \varphi(\beta_1)$ **(1)**, et par (H_3), on a $\varphi(\beta_2) \leq \varphi[\alpha \mapsto t](\beta_2)$ **(2)**. Or (H_1) donne $\varphi(\beta_1) \lessdot \varphi(\beta_2)$ **(3)**. Par la propriété 10.2 (page 173), on déduit de (1), (3) et (2) que $\varphi[\alpha \mapsto t](\beta_1) \lessdot \varphi[\alpha \mapsto t](\beta_2)$, ce qui donne le but ($\mathrm{C}_1$) : $\varphi[\alpha \mapsto t] \vdash \beta_1 \lessdot \beta_2$.

∘ *Cas I est $I_1 \wedge I_2$.* On a $\mathrm{lhs}(I_1 \wedge I_2) = \mathrm{lhs}(I_1) \cup \mathrm{lhs}(I_2)$, de telle sorte que ($\mathrm{H}_2$) donne $\alpha \notin \mathrm{lhs}(I_1)$ **(1)** et $\alpha \notin \mathrm{lhs}(I_2)$ **(2)**. De même, (H_1) implique $\varphi \vdash I_1$ **(3)** et $\varphi \vdash I_2$ **(4)**. En appliquant l'hypothèse d'induction à (3), (1) et (H_3), puis à (4), (2) et (H_3), on obtient $\varphi[\alpha \mapsto t] \vdash I_1$ et $\varphi[\alpha \mapsto t] \vdash I_2$, ce qui donne le but ($\mathrm{C}_1$) : $\varphi[\alpha \mapsto t] \vdash I_1 \wedge I_2$.

∘ *Cas I est $\exists \bar{\alpha}.I'$.* Quitte à renommer les variables $\bar{\alpha}$ dans I', on peut supposer $\alpha \notin \bar{\alpha}$. Puisque $\mathrm{lhs}(I) = \mathrm{lhs}(I') \backslash \bar{\alpha}$, on a $\alpha \notin \mathrm{lhs}(I')$ **(1)**. De plus, par (H_1), il existe $\vec{t}$ tel que $\varphi[\vec{\alpha} \mapsto \vec{t}] \vdash I'$ **(2)**. En appliquant l'hypothèse d'induction à (2), (1) et (H_3), on a $\varphi[\vec{\alpha} \mapsto \vec{t}][\alpha \mapsto t] \vdash I'$ **(3)**. Puisque $\alpha \notin \bar{\alpha}$, $\varphi[\vec{\alpha} \mapsto \vec{t}][\alpha \mapsto t]$ égale $\varphi[\alpha \mapsto t][\vec{\alpha} \mapsto \vec{t}]$, de telle sorte que (3) implique le but (C_1) : $\varphi[\alpha \mapsto t] \vdash \exists \bar{\alpha}.I'$. ⌟

▶ **Dépoussiérage (élimination des variables apolaires)** Une fois expansées et les inégalités ou gardes qu'elles portent décomposées, les variables apolaires peuvent être éliminées des multi-squelettes d'une contrainte : elles n'ont plus d'incidence sur sa sémantique. C'est la finalité du quatrième et dernier groupe de règles définissant la relation $-\Pi\twoheadrightarrow_e$, donné figure 10.10. La règle PE'-GC-VAR élimine une variable apolaire d'une multi-équation. La condition d'application assure que toutes les inégalités ou gardes portant sur cette dernière ont préalablement été décomposées. De plus, grâce à la condition « I_s dépoussiérée pour Π » de la règle PE'-STRUCT, qui est précisée par l'énoncé suivant, on est assuré que la variable éliminée n'est pas mentionnée par le contexte.

Définition 10.44 *Soit I_s une contrainte structurelle. On dit que I_s est* **dépoussiérée** *pour Π si et seulement si, pour tout $\alpha \in \mathrm{ftv}(I_s)$, $(\Pi \otimes I_s)(\alpha) \neq \varnothing$.* □

Les règles PE'-GC-TYPE-1 et PE'-GC-TYPE-2 éliminent une multi-équation ne comportant plus qu'un descripteur : la première s'applique lorsqu'il reste au moins deux multi-équations dans le

$$\langle \alpha = \bar{\bar{\alpha}} = d\,\vec{\alpha}\rangle^{\iota} \approx \bar{\bar{\tau}} \wedge I \;\; -\Pi\!\!\rightarrow_e \;\; \langle \bar{\bar{\alpha}} = d\,\vec{\alpha}\rangle^{\iota} \approx \bar{\bar{\tau}} \wedge I \quad \textit{si } \Pi(\alpha) = \varnothing \textit{ et } \alpha \notin \mathrm{ftv}(\bar{\bar{\alpha}}, \vec{\alpha}, \bar{\bar{\tau}}, I) \qquad \text{PE'-GC-VAR}$$

$$\langle \bar{\bar{\alpha}} = d\,\vec{\alpha}\rangle^{\iota_1} \approx \langle d\,\vec{\beta}\rangle^{\iota_2} \approx \bar{\bar{\tau}} \wedge I \;\; -\Pi\!\!\rightarrow_e \;\; \langle \bar{\bar{\alpha}} = d\,\vec{\alpha}\rangle^{\iota_1 \vee \iota_2} \approx \bar{\bar{\tau}} \wedge I \quad \textit{si } \forall i \in [1,n] \;\; \vec{\alpha}_{|i} \approx^{d.i} \vec{\beta}_{|i} \in I \qquad \text{PE'-GC-TYPE-1}$$

$$\langle d\,\vec{\alpha}\rangle^{\bullet} \wedge I \;\; -\Pi\!\!\rightarrow_e \;\; I \qquad \text{PE'-GC-TYPE-2}$$

Figure 10.10 – Élimination des variables apolaires

multi-squelette. La marque ι_2 de la multi-équation éliminée est répercutée sur une autre multi-équation, de manière à préserver le caractère bien marqué de la contrainte. La règle PE'-GC-TYPE-2 traite quant à elle le cas où le multi-squelette ne contient plus qu'une multi-équation.

La même simplification est généralement présente dans les solveurs de contraintes d'unification (ou, de manière générale, les synthétiseurs de types pour les systèmes à base d'unification). Elle est cependant souvent omise de la spécification de l'algorithme : si ce dernier est implémenté dans un langage à gestion automatique de la mémoire, elle est réalisée directement par le glaneur de cellules, car en l'absence de multi-squelettes, une multi-équation ne contenant que des variables apolaires n'est *a priori* mentionnée par aucune autre structure de donnée.

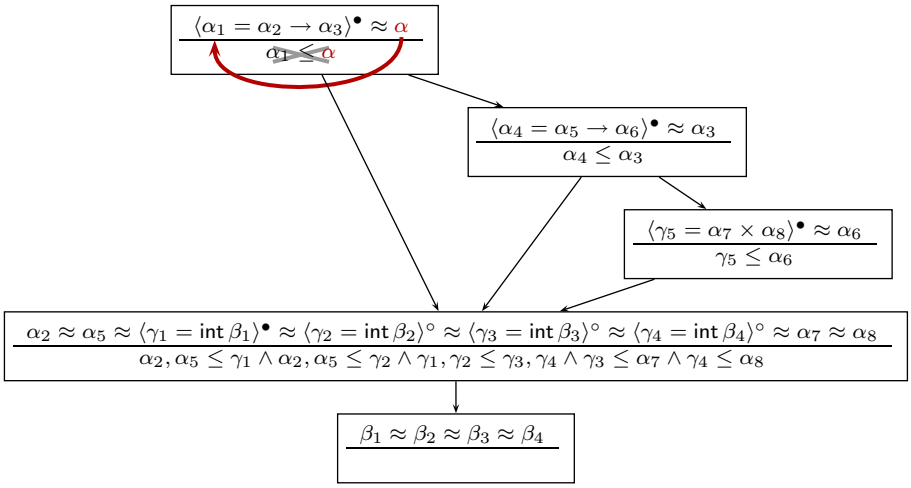
▶ **Exemple** Avant de donner la preuve de ces techniques de simplification, observons leur effet sur l'exemple dont j'ai commencé la présentation à la section 10.2.3 (page 184). Le contexte dans lequel est placé la contrainte primaire étudiée est $\mathsf{let}\, f \,:\, \forall\alpha[[\,]].\alpha\ \mathsf{in}\ \ldots$. La variable α est donc positive. Les autres variables de types que fait intervenir l'exemple sont quantifiées existentiellement au sommet de la contrainte primaire, elles sont donc initialement apolaires. Sur les figures, les variables positives sont imprimées en rouge, tandis que les négatives apparaissent en vert. Dans le premier multi-squelette, la variable α est positive et a une seule borne inférieure α_1 (figure 10.11-a) ; ces deux variables peuvent donc être unifiées, ce qui donne la contrainte décrite par la figure 10.11-b. La polarité de la variable α doit être propagée sur les fils de son descripteur, conformément au théorème 10.42 (page 199) : le constructeur $\rightarrow$ étant contravariant pour son premier paramètre et covariant pour son second, les variables α_2 et α_3 deviennent respectivement négative et positive. L'étape suivante consiste à dépoussiérer la multi-équation, en retirant la variable α_1 qui est apolaire (figure 10.11-c). Ce processus se répète exactement de la même manière dans le deuxième (figures 10.12-a et 10.12-b) puis le troisième (figures 10.12-c et 10.13-a) multi-squelettes considérés. Il faut remarquer que, grâce à la réduction des chaînes, aucune variable n'a jusqu'à présent été expansée. Le traitement du quatrième multi-squelette (figure 10.13-b, page 206) commence par la réduction de deux chaînes, qui permet d'éliminer les deux variables α_7 et α_8 qui sont positives et n'ont chacune qu'une seule borne inférieure. Aucune autre chaîne ne peut être réduite : α_2 et α_5 sont négatives et ont deux bornes supérieures tandis que γ_1 et γ_2 qui sont apolaires ont chacune deux bornes inférieures et deux bornes supérieures. Il faut donc procéder à l'expansion des deux variables α_2 et α_5, comme indiqué par la figure 10.13-c (page 206). Les polarités peuvent ensuite être propagées et les inégalités décomposées figure 10.14-a (page 207), ce qui termine le processus d'expansion.

Les lemmes suivants étendent respectivement les lemmes 10.22, 10.23, 10.24 et 10.28 au système de réécriture $-\Pi\!\!\rightarrow_e$ en prenant en considération les règles de simplification.

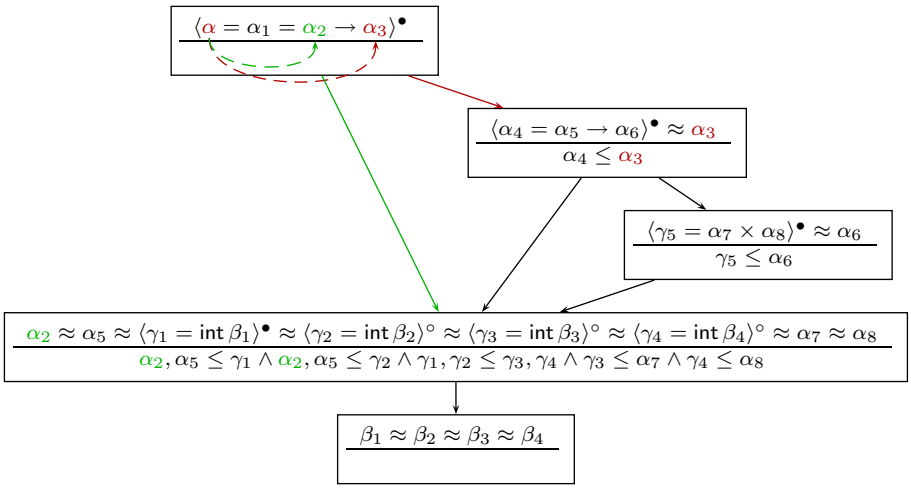
Lemme 10.45 *La réduction $-\Pi\!\!\rightarrow_e$ préserve le caractère bien marqué, unifié et acyclique des contraintes.* □

Lemme 10.46 (Correction) *La réduction $-\Pi\!\!\rightarrow_e$ préserve la sémantique des contraintes modulo la polarisation* Π *(*i.e. *si $I_1 \; -\Pi\!\!\rightarrow_e I_2$ alors $I_1 \equiv I_2 \mod \Pi$).* □

10.11-a Réduction d'une chaîne



10.11-b Propagation des polarités



10.11-c Élimination d'une variable apolaire

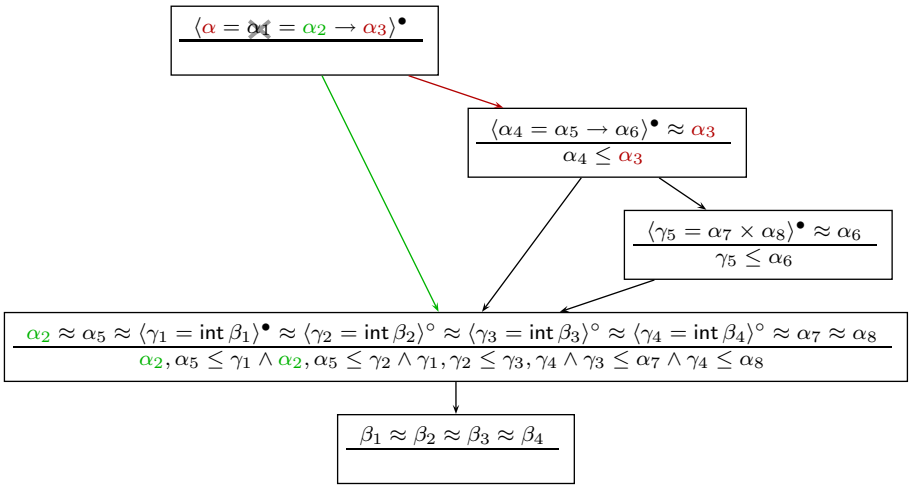
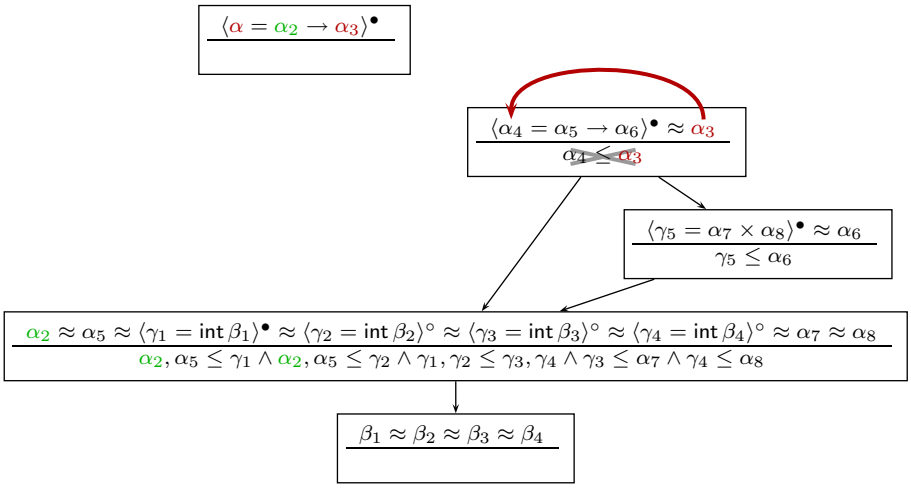
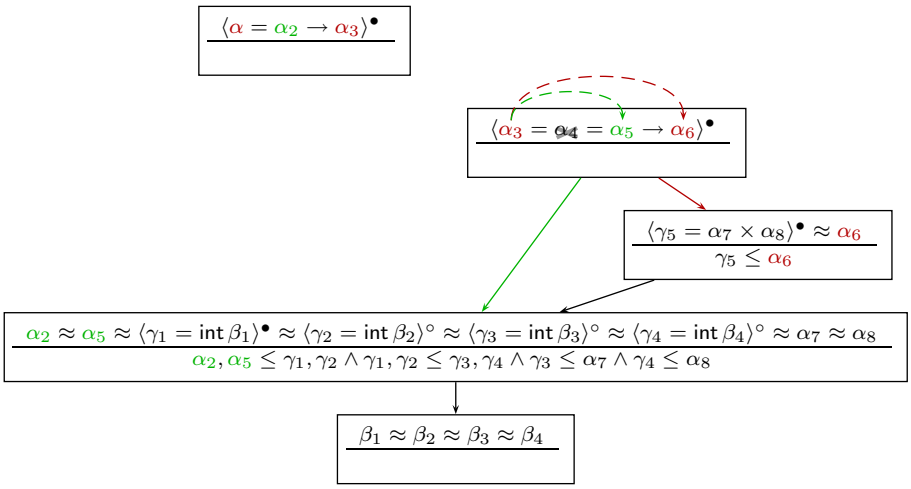


Figure 10.11 – Exemple (2/5)

10.12-a Réduction d'une chaîne



10.12-b Propagation des polarités et élimination d'une variable apolaire



10.12-c Réduction d'une chaîne

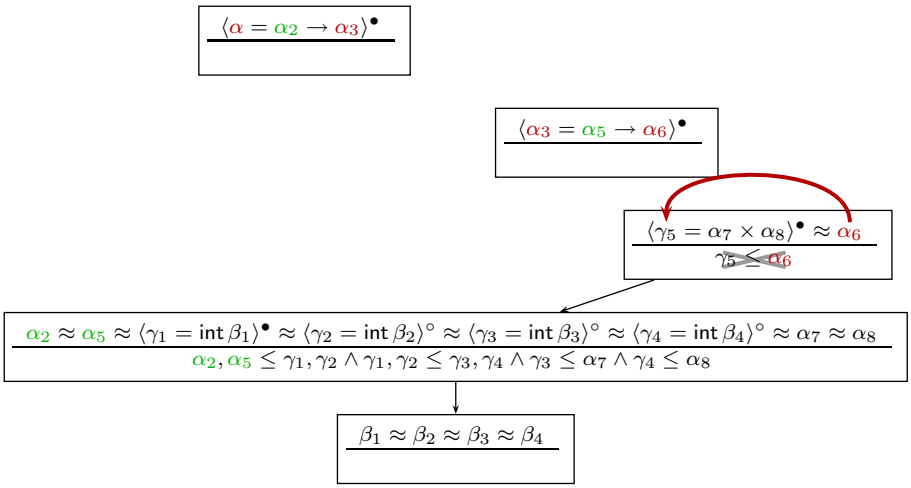
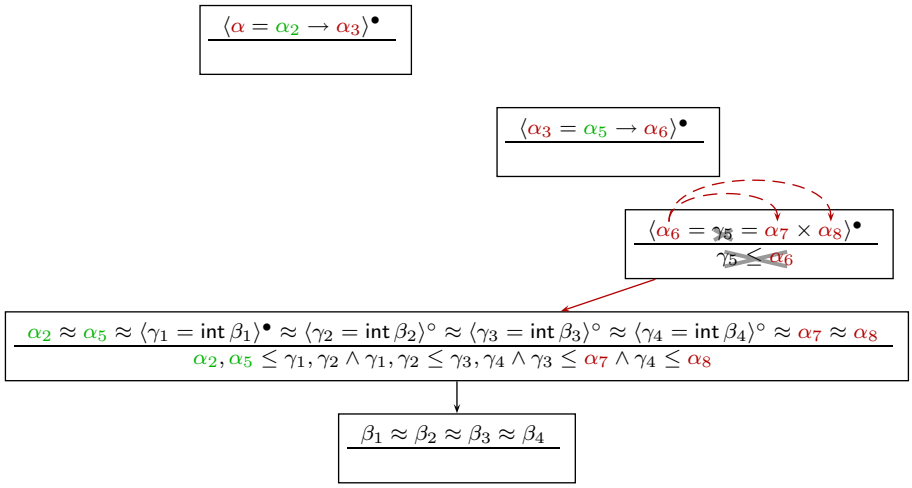
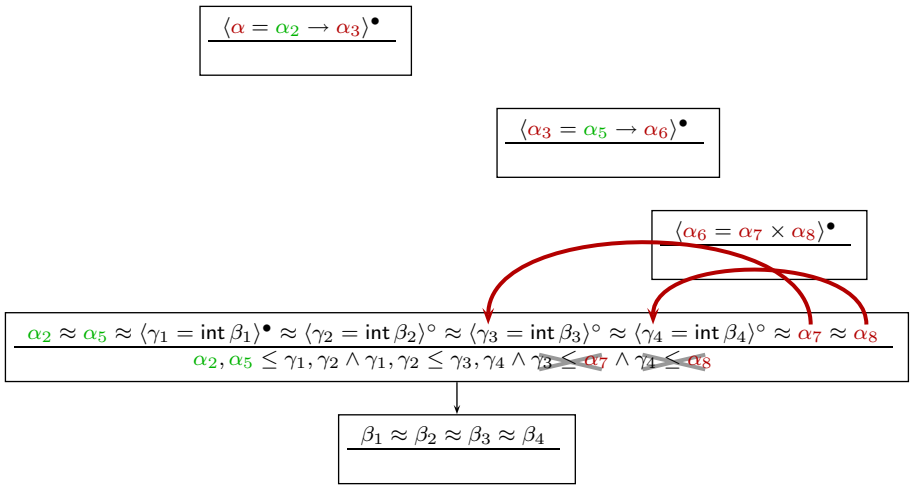


Figure 10.12 – Exemple (3/5)

10.13-a Propagation des polarités et élimination d'une variable apolaire



10.13-b Réduction de deux chaînes



10.13-c Expansion

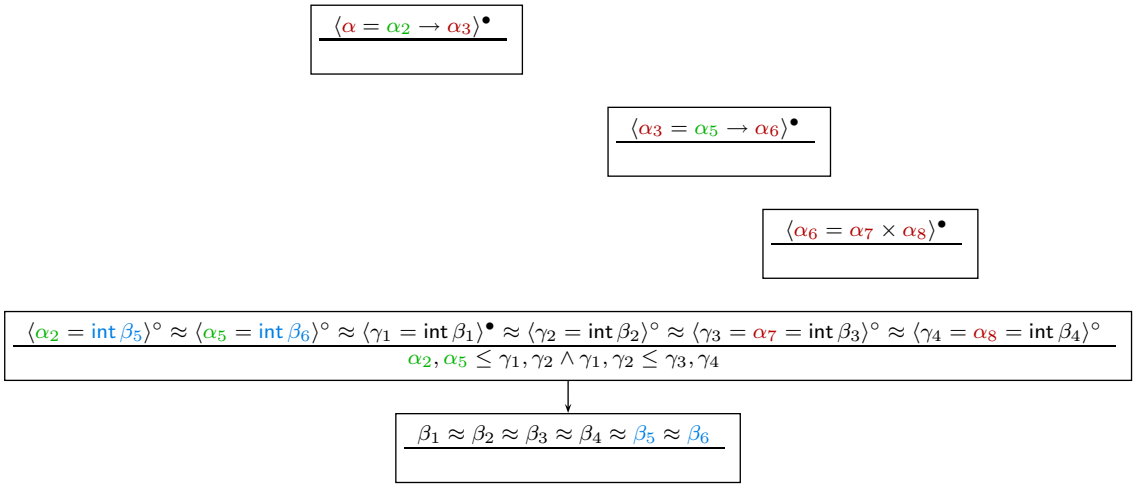
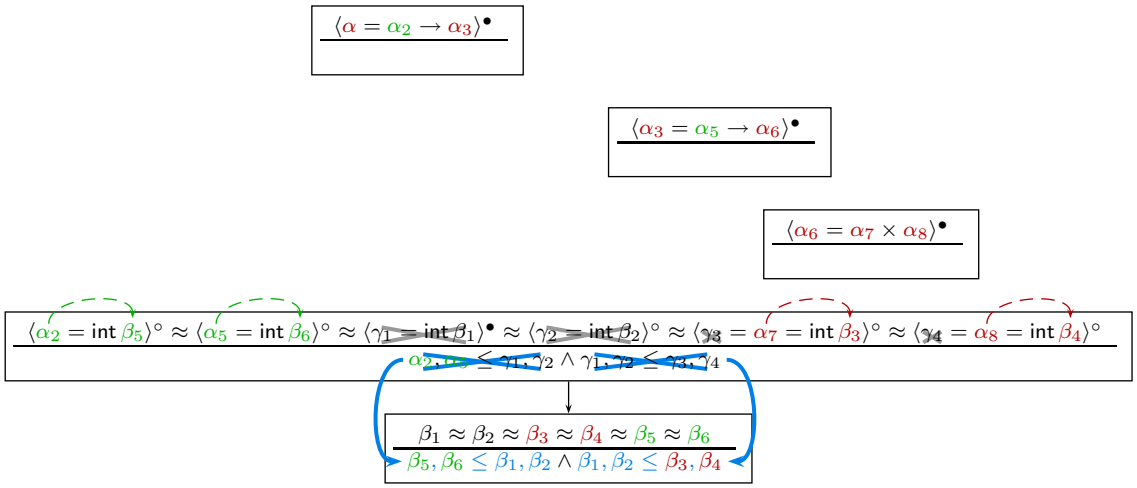
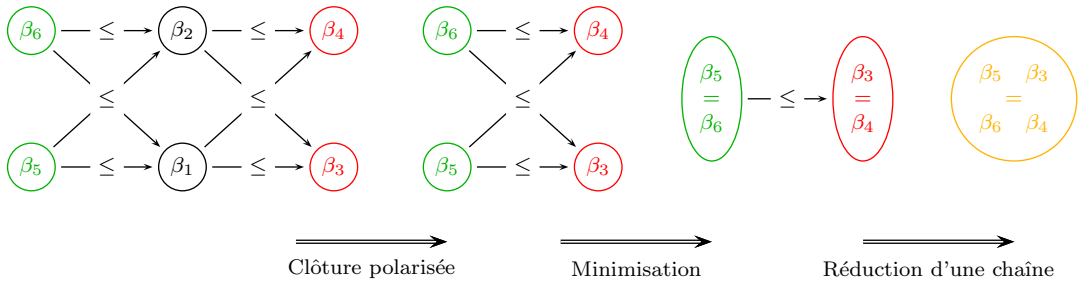


Figure 10.13 – Exemple (4/5)

10.14-a Propagation des polarités, décomposition et élimination des variables apolaires



10.14-b Simplification du graphe atomique



10.14-c Hash-consing

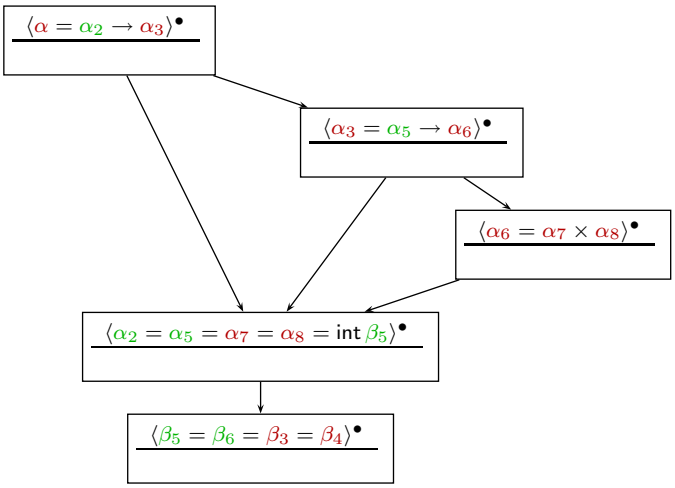


Figure 10.14 – Exemple (5/5)

⌜ *Preuve.* Je procède par induction sur la dérivation de la réduction donnée en hypothèse, et examine successivement les différentes règles.

◦ *Cas* PE'-BASE. Par le lemme 10.23 (page 188).

◦ *Cas* PE'-EXISTS. Grâce à l'hypothèse d'induction, il suffit de montrer que, si on a $I \equiv I' \bmod \Pi[\vec{\alpha} \mapsto \varnothing]$ alors $\exists \bar{\alpha}.I \equiv \exists \bar{\alpha}.I' \bmod \Pi$. Cette propriété a déjà été établie dans le cas relatif à la règle POL-EXISTS de la preuve du théorème 10.40 (page 198).

◦ *Cas* PE'-STRUCT. Par le théorème 10.42 (page 199).

◦ *Cas* PE'-CHAIN-LEFT. Je dois montrer $\langle \alpha = \bar{\alpha} \rangle^\bullet \approx \langle \beta = \bar{\tau} \rangle^\iota \approx \tilde{\tau} \wedge \alpha \leq \beta \wedge I \equiv \langle \alpha = \bar{\alpha} = \beta = \bar{\tau} \rangle^\iota \approx \tilde{\tau} \wedge I \bmod \Pi$ **(1)** sous les hypothèses $\alpha\bar{\alpha} \# \mathrm{ftv}(\beta, \bar{\tau}, \tilde{\tau}) \cup \mathrm{lhs}(I)$ **(2)** et $\Pi(\alpha\bar{\alpha}) \subseteq -$ **(3)**. On a tout d'abord $\langle \alpha = \bar{\alpha} = \beta = \bar{\tau} \rangle^\iota \Vdash \langle \alpha = \bar{\alpha} \rangle^\bullet \approx \langle \beta = \bar{\tau} \rangle^\iota \wedge \alpha \leq \beta$, ce dont on déduit $\langle \alpha = \bar{\alpha} = \beta = \bar{\tau} \rangle^\iota \approx \tilde{\tau} \wedge I \Vdash \langle \alpha = \bar{\alpha} \rangle^\bullet \approx \langle \beta = \bar{\tau} \rangle^\iota \approx \tilde{\tau} \wedge \alpha \leq \beta \wedge I$ **(4)**. Considérons maintenant une affectation φ telle que $\varphi \vdash \langle \alpha = \bar{\alpha} \rangle^\bullet \approx \langle \beta = \bar{\tau} \rangle^\iota \approx \tilde{\tau} \wedge \alpha \leq \beta \wedge I$ **(5)**. Définissons φ' par $\varphi'(\gamma) = \varphi(\beta)$ si $\gamma \in \alpha\bar{\alpha}$ **(6)** et $\varphi'(\gamma) = \varphi(\gamma)$ sinon **(7)**. Soit $\gamma \in \alpha\bar{\alpha}$ **(8)**. Par (5), on a $\varphi(\gamma) = \varphi(\alpha)$ et $\varphi(\alpha) \leq \varphi(\beta)$, ce dont on déduit, grâce à (6), $\varphi(\gamma) \leq \varphi'(\gamma)$. En déchargeant (8) puis en utilisant (3) et (7) on obtient $\varphi' \leq^\Pi \varphi$ **(9)**. Par (5), (7) et (2) on a $\varphi' \vdash \langle \beta = \bar{\tau} \rangle^\iota \approx \tilde{\tau}$, puis par (6), $\varphi' \vdash \langle \alpha = \bar{\alpha} = \beta = \bar{\tau} \rangle^\iota \approx \tilde{\tau}$ **(10)**. Par (5), (6), (7) et (3), le lemme 10.43 (page 202) donne $\varphi' \vdash I$ **(11)**. En combinant (10) et (11), puis en déchargeant (5), on obtient $\langle \alpha = \bar{\alpha} \rangle^\bullet \approx \langle \beta = \bar{\tau} \rangle^\iota \approx \tilde{\tau} \wedge \alpha \leq \beta \wedge I \Vdash \langle \alpha = \bar{\alpha} = \beta = \bar{\tau} \rangle^\iota \approx \tilde{\tau} \wedge I \bmod \Pi$ **(12)**. Les implications (4) et (12) donnent le but (1).

◦ *Cas* PE'-CHAIN-RIGHT. Ce cas est symétrique au précédent.

◦ *Cas* PE'-CYCLE, PE'-STUTTER-$\leq$, PE'-STUTTER-$\lessdot$, PE'-ATOM-$\lessdot$ *et* PE'-REFLEX-$\leq$. La correction de ces règles découle de manière immédiate de l'interprétation des prédicats d'égalité, de sous-typage et de garde.

◦ *Cas* PE'-GC-VAR. Je dois montrer $\langle \alpha = \bar{\alpha} = d\,\vec{\alpha} \rangle^\iota \approx \tilde{\tau} \wedge I \equiv \langle \bar{\alpha} = d\,\vec{\alpha} \rangle^\iota \approx \tilde{\tau} \wedge I \bmod \Pi$ **(1)** sous les hypothèses $\Pi(\alpha) = \varnothing$ **(2)** et $\alpha \notin \mathrm{ftv}(\bar{\alpha}, \vec{\alpha}, \tilde{\tau}, I)$ **(3)**. On a $\langle \alpha = \bar{\alpha} = d\,\vec{\alpha} \rangle^\iota \Vdash \langle \bar{\alpha} = d\,\vec{\alpha} \rangle^\iota$ donc $\langle \alpha = \bar{\alpha} = d\,\vec{\alpha} \rangle^\iota \approx \tilde{\tau} \wedge I \Vdash \langle \bar{\alpha} = d\,\vec{\alpha} \rangle^\iota \approx \tilde{\tau} \wedge I$ **(4)**. Considérons maintenant une affectation φ telle que $\varphi \vdash \langle \bar{\alpha} = d\,\vec{\alpha} \rangle^\iota \approx \tilde{\tau} \wedge I$ **(5)**. Posons $\varphi' = \varphi[\alpha \mapsto \varphi(d\,\vec{\alpha})]$. Par (2), on a $\varphi' \leq^\Pi \varphi$ **(6)** et, par (3), (5) implique $\varphi' \vdash \langle \bar{\alpha} = d\,\vec{\alpha} \rangle^\iota \approx \tilde{\tau} \wedge I$. Puisque $\varphi'(\alpha) = d\,\vec{\alpha}$, on en déduit $\varphi' \vdash \langle \alpha = \bar{\alpha} = d\,\vec{\alpha} \rangle^\iota \approx \tilde{\tau} \wedge I$ **(7)**. En déchargeant (5) sur (6) et (7), on obtient $\langle \bar{\alpha} = d\,\vec{\alpha} \rangle^\iota \approx \tilde{\tau} \wedge I \Vdash \langle \alpha = \bar{\alpha} = d\,\vec{\alpha} \rangle^\iota \approx \tilde{\tau} \wedge I \bmod \Pi$ **(8)**. Les implications (4) et (8) donnent l'équivalence (1).

◦ *Cas* PE'-GC-TYPE-1. Je dois montrer que $\langle \bar{\alpha} = d\,\alpha_1 \cdots \alpha_n \rangle^{\iota_1} \approx \langle d\,\beta_1 \cdots \beta_n \rangle^{\iota_2} \approx \tilde{\tau} \wedge I \equiv \langle \bar{\alpha} = d\,\alpha_1 \cdots \alpha_n \rangle^{\iota_1 \vee \iota_2} \approx \tilde{\tau} \wedge I \bmod \Pi$ **(1)** sous l'hypothèse $\forall i \in [1, n]\ \ \alpha_i \approx^{d.i} \beta_i \in I$ **(2)**. On a $d\,\alpha_1 \cdots \alpha_n \approx d\,\beta_1 \cdots \beta_n \equiv \bigwedge_{i \in [1,n]} \alpha_i \approx^{d.i} \beta_i$. Par (2) et LOG-DUP, on en déduit que $d\,\alpha_1 \cdots \alpha_n \approx d\,\beta_1 \cdots \beta_n \wedge I \equiv I$, puis $\langle \bar{\alpha} = d\,\alpha_1 \cdots \alpha_n \rangle^{\iota_1} \approx \langle d\,\beta_1 \cdots \beta_n \rangle^{\iota_2} \approx \tilde{\tau} \wedge I \equiv \langle \bar{\alpha} = d\,\alpha_1 \cdots \alpha_n \rangle^{\iota_1 \vee \iota_2} \approx \tilde{\tau} \wedge I$. Le but (1) s'ensuit.

◦ *Cas* PE'-GC-TYPE-2. Un multi-squelette à un seul élément est une tautologie. On en déduit que les contraintes $\langle d\,\vec{\alpha} \rangle^\bullet \wedge I$ et I sont équivalentes. ⌟

Lemme 10.47 (Terminaison) *Le système de réécriture* $-\Pi\twoheadrightarrow_e$ *est fortement normalisant sur les contraintes acycliques.* □

⌜ *Preuve (esquisse).* Soit I une contrainte acyclique et I' un de ses représentants nommés. On mesure la contrainte I par le produit lexicographique des quantités suivantes :

(1) Le multi-ensemble des hauteurs des multi-équations dans I' ne contenant que des variables.

(2) Le multi-ensemble des hauteurs des inégalités et des gardes dans I'.

(3) Le multi-ensemble des profondeurs des quantificateurs existentiels.

(4) Le nombre de gardes dans I'.

(5) Le nombre total d'éléments des multi-équations de I.

On vérifie par induction que cette mesure est diminuée par chacune des règles définissant la relation $-\Pi\twoheadrightarrow_e$. Puisque cette mesure étend celle utilisée dans la preuve du lemme 10.24 (page 189) par de nouvelles composantes, elle est diminuée par PE'-BASE. Le cas des règles PE'-EXISTS et PE'-STRUCT se traite par induction. La règle PE'-CYCLE et celles de la figure 10.9 (page 201) ne modifient pas la structure décrite par les multi-squelettes préservant la hauteur de toutes les variables. PE'-CYCLE fusionne des multi-équations et élimine des inégalités diminuant ainsi (1) (au sens large) et (2) au sens strict. PE'-CHAIN-LEFT et PE'-CHAIN-RIGHT éliminent une multi-équation ne contenant que des variables, réduisant (1). Puisque PE'-STUTTER-$\leq$, PE'-STUTTER-$\lessdot$ et PE'-REFLEX-$\leq$ éliminent une inégalité ou garde, sans affecter les multi-squelettes, elles diminuent (2) en laissant (1) inchangée. PE'-ATOM-$\lessdot$ remplace une garde par une inégalité, diminuant (4) sans affecter les points (1) à (3). Les règles PE'-GC-VAR, PE'-GC-TYPE-1 et PE'-GC-TYPE-2 sont susceptibles d'éliminer des arcs dans le graphe $\prec_{I'}$, et donc de diminuer la hauteur de certaines variables, réduisant alors les quantités (1) et (2). Elles n'affectent pas (3) et (4) et diminuent dans tous les cas (5). ⌟

Je caractérise enfin les formes normales du système de réécriture $-\Pi\twoheadrightarrow_e$. Puisqu'il inclut la relation $\twoheadrightarrow_e$, ses formes normales sont des contraintes réduites. De plus, grâce aux règles de dépoussiérage, elles ne comportent plus de variable apolaire dans leur partie structurelle, d'où la définition suivante.

Définition 10.48 (Contrainte réduite dépoussiérée) *Soit $I = \mathbb{X}[I_s \wedge I_a]$ une contrainte réduite écrite sous la forme donnée à la définition 10.27 (page 189). On dit que I est* **dépoussiérée** *pour Π si et seulement si I_s est dépoussiérée pour $\Pi[\mathrm{dtv}(\mathbb{X}) \mapsto \varnothing]$.* □

Lemme 10.49 (Formes normales) *Si I est une contrainte bien marquée, unifiée, acyclique et normale pour $-\Pi\twoheadrightarrow_e$ alors elle est réduite et dépoussiérée pour Π.* □

Ces quatre lemmes permettent d'obtenir le théorème suivant, qui établit que la relation $-\Pi\twoheadrightarrow_e$ donne un algorithme réécrivant une contrainte issue de l'unification et du test d'occurrence en une contrainte réduite dépoussiérée pour Π.

Théorème 10.50 (Expansion et décomposition avec simplification) *Soit I une contrainte bien marquée, unifiée et acyclique différente de* false. *Alors $\twoheadrightarrow_e$ termine sur I et, si $I \twoheadrightarrow_e^* I'$ et I' est une forme normale, alors I' est bien marquée, unifiée, acyclique, réduite et dépoussiérée pour Π.* □

10.3.3 Simplifications réalisées après l'expansion et la décomposition

Je présente maintenant deux techniques puissantes de simplification qui s'appliquent de manière successive à une contrainte atomique satisfiable et unifiée, sous une polarisation arbitraire. Grâce au théorème 10.42 (page 199), elles peuvent être utilisées au terme du processus d'expansion, sur le fragment atomique de la contrainte réduite. Comme les heuristiques précédentes, elles visent à réduire le nombre de variables distinctes : la première procède par élimination des variables inaccessibles tandis que la seconde identifie des variables équivalentes. Ces identifications peuvent ensuite être remontées sur la structure des types grâce au *hash-consing*.

▶ **Clôture polarisée** Cette première procédure permet d'éliminer d'une contrainte atomique toutes les variables apolaires : ces points intermédiaires, introduits par le générateur de contraintes ou les étapes précédentes de l'algorithme de résolution, ne sont pas accessibles *via* le contexte où apparaît la contrainte. Nous allons en effet voir qu'il est systématiquement possible de réécrire la contrainte sous une forme équivalente qui ne les fait pas intervenir. Ce problème de dépoussiérage est cependant plus subtil au niveau atomique que dans le cas non-terminal (section 10.3.2, page 199), puisqu'une variable apolaire peut ici être reliée à d'autres variables par des inégalités et des gardes. Son élimination nécessite donc d'effectuer une clôture transitive du graphe formé

par les prédicats d'inégalité et de garde, de manière à pouvoir retirer simultanément les arcs dont elle est l'origine ou l'extrémité. Par exemple, si on souhaite éliminer la variable α_2 de la contrainte $\alpha_1 \leq \alpha_2 \wedge \alpha_2 \leq \alpha_3 \wedge \alpha_1 \approx \alpha_2 \approx \alpha_3$, il faut remplacer les deux inégalités par $\alpha_1 \leq \alpha_3$.

De plus, étant donné que la contrainte considérée est supposée satisfiable, il n'est nécessaire de prendre en compte, dans ce calcul de clôture, que les chemins du graphe qui ont une chance d'être utiles par la suite. Prenons l'exemple d'une variable non négative α. Comme je l'ai expliqué à la section 10.3.1 (page 196), cette variable ne pourra recevoir ultérieurement aucun nouveau minorant. Il est ainsi inutile de conserver un chemin $\alpha \leq \beta$ ou $\alpha \lessdot \beta$ dans le graphe : en effet, celui-ci ne peut intervenir dans une nouvelle phase de résolution qu'en combinaison avec une inégalité ou une garde dont α est le membre droit qui serait introduite, *via* le contexte, ce qui précisément ne peut advenir. De manière symétrique, si α est une variable non positive, un chemin $\beta \leq \alpha$ ou $\beta \lessdot \alpha$ dont α est l'extrémité peut être abandonné.

Définition 10.51 (Clôture polarisée) *Soit I une contrainte atomique, unifiée et satisfiable ; soit Π une polarisation. Soit V^+ (respectivement V^-) un ensemble contenant exactement une variable positive (respectivement négative) de chaque multi-équation de I qui en comporte une. Soit $V^\bullet$ un ensemble contenant exactement une variable de chaque multi-équation comportant une constante atomique. Soit I' la contrainte atomique, bien marquée et unifiée définie par :*

$$
\begin{aligned}
&\textbf{\textit{(i)}}\ \alpha \approx \beta \in I' \ \Leftrightarrow\ \alpha \approx \beta \in I \quad \textit{et}\ (\Pi(\alpha) \neq \varnothing\ \textit{ou}\ \alpha \in V^\bullet)\ \textit{et}\ (\Pi(\beta) \neq \varnothing\ \textit{ou}\ \alpha \in V^\bullet)\\
&\textbf{\textit{(ii)}}\ \alpha = \beta \in I' \ \Leftrightarrow\ \alpha = \beta \in I \quad \textit{et}\ (\Pi(\alpha) \neq \varnothing\ \textit{ou}\ \alpha \in V^\bullet)\ \textit{et}\ (\Pi(\beta) \neq \varnothing\ \textit{ou}\ \alpha \in V^\bullet)\\
&\textbf{\textit{(iii)}}\ \alpha = \ell \in I' \ \Leftrightarrow\ \alpha = \ell \in I \quad \textit{et}\ (\Pi(\alpha) \neq \varnothing\ \textit{ou}\ \alpha \in V^\bullet)\\
&\textbf{\textit{(iv)}}\ \alpha \leq \beta \in I' \ \Leftrightarrow\ \alpha \leq \beta \in^* I\ \textit{et}\ (\alpha,\beta) \in (V^- \times V^+) \cup (V^\bullet \times V^+) \cup (V^- \times V^\bullet)\ \textit{et}\ \alpha = \beta \notin I\\
&\textbf{\textit{(v)}}\ \alpha \lessdot \beta \in I' \ \Leftrightarrow\ \alpha \lessdot \beta \in^* I\ \textit{et}\ (\alpha,\beta) \in (V^- \times V^+) \cup (V^\bullet \times V^+) \cup (V^- \times V^\bullet)\ \textit{et}\ \{\alpha,\beta\} \not\subseteq \mathcal{V}_{\mathsf{Atom}}
\end{aligned}
$$

On dit alors que I' est une **clôture polarisée** *de I sous Π* □

Pour effectuer le calcul de la clôture polarisée d'une contrainte, il est nécessaire de choisir de manière arbitraire, dans chaque multi-équation où cela est possible, un représentant positif et un représentant négatif ; ainsi qu'un représentant pour chaque multi-équation contenant une constante atomique : le processus de clôture ne produit ensuite des inégalités ou des gardes qu'entre ces variables distinguées, évitant ainsi la génération de paires de contraintes redondantes dont les membres seraient par ailleurs deux à deux unifiés.

Les points (i), (ii) et (iii) de la définition reproduisent les multi-squelettes et les multi-équations de I dans sa clôture polarisée, en éliminant de ceux-ci les variables apolaires. Une variable éventuellement apolaire est cependant conservée pour chaque constante atomique mentionnée dans la contrainte. Il s'agit essentiellement d'un artefact de la représentation des contraintes : puisque qu'une constante atomique n'est pas acceptée comme membre d'une inégalité ou d'une garde, un nom doit être introduit pour la désigner. Les points (iv) et (v) introduisent une inégalité ou une garde pour chaque chemin menant d'une variable (distinguée) négative à une variable (distinguée) positive dans la contrainte I. Les deux conditions supplémentaires permettent d'éviter l'introduction de contraintes inutiles : une inégalité entre deux variables de la même multi-équation ou bien une garde entre deux variables de sorte Atom (puisqu'une inégalité équivalente est déjà générée dans ce cas particulier). Le théorème suivant énonce la correction sémantique du processus de clôture polarisée.

Théorème 10.52 (Clôture polarisée) *Soit I une contrainte atomique, unifiée et satisfiable, et Π une polarisation. Soit I' une clôture polarisée par Π de I. Alors $I \equiv I' \mod \Pi$.* □

⌜ *Preuve.* Soient I, Π, V^+, V^-, $V^\bullet$ et I' définis comme dans l'énoncé de la définition 10.51. L'implication $I \Vdash I'$ **(1)** étant une conséquence immédiate de la définition 10.51 et du lemme 10.32

(page 192), il reste à montrer $I' \Vdash I \mod \Pi$ **($\mathbf{C_1}$)** pour conclure. Soit φ' telle que $\varphi' \vdash I'$ **(2)** ; je cherche à construire une solution φ de I telle que $\varphi \leq^{\Pi} \varphi'$.

Soit α une variable de type de sorte κ. Je pose $\mathrm{skel}(\alpha) = \{\, \beta \mid \alpha \approx \beta \in I \text{ et } \beta \in V^+ \cup V^- \cup V^\bullet \,\}$. Si cet ensemble n'est pas vide alors, par (2) et le point (i), I étant unifiée, tous ses éléments ont une image par φ' appartenant au même squelette brut, que je note $\mathrm{gskel}(\alpha)$. Si $\mathrm{skel}(\alpha)$ est vide, je définis — de manière arbitraire — $\mathrm{gskel}(\alpha)$ comme étant le squelette brut contenant le type $\Lambda_\kappa(\bot)$. Je pose également :

$$\begin{aligned} \mathrm{lbs}_{\leq}(\alpha) &= \{\, \varphi'(\beta) \mid \beta \leq \alpha \in^* I \text{ et } \beta \in V^- \cup V^\bullet \,\} \\ \mathrm{lbs}_{\lessdot}(\alpha) &= \{\, \mathrm{lift}(\varphi'(\beta)/\,\mathrm{gskel}(\alpha)) \mid \beta \lessdot \alpha \in^* I \text{ et } \beta \in V^- \cup V^\bullet \,\} \end{aligned}$$

Soit $t \in \mathrm{lbs}_{\leq}(\alpha)$ **(3)**. Il existe β tel que $t = \varphi'(\beta)$ **(4)** et $\beta \leq \alpha \in^* I$ **(5)**. Puisque I est unifiée, (5) entraine $\beta \approx \alpha \in I$, donc $\beta \in \mathrm{skel}(\alpha)$ puis $\varphi'(\beta) \in \mathrm{gskel}(\alpha)$ et, par (4), $t \in \mathrm{gskel}(\alpha)$. En déchargeant l'hypothèse (3), on en déduit que $\mathrm{lbs}_{\leq}(\alpha) \subseteq \mathrm{gskel}(\alpha)$. De même, par la propriété 10.4 (page 174), on a $\mathrm{lbs}_{\lessdot}(\alpha) \subseteq \mathrm{gskel}(\alpha)$. On peut donc définir l'affectation φ comme suit :

$$\varphi(\alpha) = \bigsqcup\nolimits_{\mathrm{gskel}(\alpha)} \big(\mathrm{lbs}_{\leq}(\alpha) \cup \mathrm{lbs}_{\lessdot}(\alpha) \big) \qquad \textbf{(6)}$$

Établissons quelques propriétés liminaires relatives à l'affectation φ.

∘ Remarquons tout d'abord que si $\alpha_1 \leq \alpha_2 \in^* I$ alors, I étant unifiée, on a $\mathrm{gskel}(\alpha_1) = \mathrm{gskel}(\alpha_2)$ ainsi que, par ATM-LEQ-LEQ, $\mathrm{lbs}_{\leq}(\alpha_1) \subseteq \mathrm{lbs}_{\leq}(\alpha_2)$ et, par ATM-GD-LEQ, $\mathrm{lbs}_{\lessdot}(\alpha_1) \subseteq \mathrm{lbs}_{\lessdot}(\alpha_2)$ **($\mathbf{P_1}$)**.

∘ *Montrons que pour tout $\alpha \in V^- \cup V^\bullet$ on a $\varphi'(\alpha) \leq \varphi(\alpha)$* **($\mathbf{P_2}$)**. Soit $\alpha \in V^- \cup V^\bullet$ **(7)**. Puisque I est unifiée, ATM-EQ donne $\alpha \leq \alpha \in^* I$. Avec (7), on en déduit que $\varphi'(\alpha) \in \mathrm{lbs}_{\leq}(\alpha)$ puis, par (6), $\varphi'(\alpha) \leq \varphi(\alpha)$.

∘ *Montrons que pour tout $\alpha \in V^+ \cup V^\bullet$ on a $\varphi(\alpha) \leq \varphi'(\alpha)$* **($\mathbf{P_3}$)**. Soit $\alpha \in V^+ \cup V^\bullet$ **(8)**. Soit $t \in \mathrm{lbs}_{\leq}(\alpha)$ **(9)**. Il existe β tel que $t = \varphi'(\beta)$ et $\beta \leq \alpha \in^* I$ **(10)** et $\beta \in V^- \cup V^\bullet$. Si $\alpha, \beta \in V^\bullet$, il existe ℓ_1 et ℓ_2 tels que $\beta = \ell_1 \in I$ **(11)** et $\alpha = \ell_2 \in I$ **(12)**. Par le théorème 10.33 (page 192), I étant satisfiable, (10), (11) et (12) impliquent $\ell_1 \leq \ell_2$. Par le point (iii), (11) et (12) impliquent $\alpha = \ell_2 \in I'$ et $\beta = \ell_1 \in I'$; par (2), on en déduit $\varphi'(\beta) = \ell_1$ et $\varphi'(\alpha) = \ell_2$, d'où $\varphi'(\beta) \leq \varphi'(\alpha)$ **(13)**. Sinon, on a $(\alpha, \beta) \in (V^- \times V^+) \cup (V^\bullet \times V^+) \cup (V^- \times V^\bullet)$. Par le point (iv), (10) donne $\beta \leq \alpha \in I'$ puis, par (2), $\varphi'(\beta) \leq \varphi'(\alpha)$ **(14)**. En déchargeant (9) sur (13) ou (14), on obtient $\forall t \in \mathrm{lbs}_{\leq}(\alpha)\ t \leq \varphi'(\alpha)$ **(15)**. Considérons maintenant $t \in \mathrm{lbs}_{\lessdot}(\alpha)$ **(16)**. Il existe β tel que $t = \mathrm{lift}(\varphi'(\beta)/\,\mathrm{gskel}(\alpha))$ et $\beta \lessdot \alpha \in^* I$ **(17)** et $\beta \in V^- \cup V^\bullet$ **(18)**. Si $\alpha, \beta \in \mathcal{V}_{\mathsf{Atom}}$ alors, $t = \mathrm{lift}(\varphi'(\beta)/\,\mathrm{gskel}(\alpha)) = \varphi'(\beta)$ **(19)** et, par ATM-ATOM-GD, $\beta \leq \alpha \in^* I$, d'où $t \in \mathrm{lbs}_{\leq}(\alpha)$. On en déduit, par (15), $t \leq \varphi'(\alpha)$ **(20)**. Sinon, par le point (v), (17), (18) et (8) impliquent $\beta \lessdot \alpha \in I'$. Par (2), on en déduit $\varphi'(\beta) \lessdot \varphi'(\alpha)$, puis, grâce à la propriété 10.4 (page 174) et par (19), $t \leq \varphi'(\alpha)$ **(21)**. En déchargeant l'hypothèse (16) sur (20) ou (21), $\forall t \in \mathrm{lbs}_{\lessdot}(\alpha)\ t \leq \varphi'(\alpha)$ **(22)** s'ensuit. Par (15) et (22), $\varphi(\alpha)$ est définie par (6) comme étant la plus petite borne supérieure d'un ensemble dont tous les éléments sont majorés par $\varphi'(\alpha)$. On en déduit que $\varphi(\alpha) \leq \varphi'(\alpha)$.

Je montre maintenant que φ satisfait I **($\mathbf{C_2}$)**. Puisque I est unifiée et atomique, il me suffit de vérifier les propriétés suivantes.

∘ *Si $\alpha_1 \approx \alpha_2 \in I$ alors $\varphi(\alpha_1) \approx \varphi(\alpha_2)$* **($\mathbf{P_4}$)**. Supposons $\alpha_1 \approx \alpha_2 \in I$. Puisque I est unifiée, on a $\mathrm{skel}(\alpha_1) = \mathrm{skel}(\alpha_2)$. On en déduit que $\mathrm{gskel}(\alpha_1) = \mathrm{gskel}(\alpha_2)$, puis, par (6), $\varphi(\alpha_1) \approx \varphi(\alpha_2)$.

∘ *Si $\alpha_1 = \alpha_2 \in I$ alors $\varphi(\alpha_1) = \varphi(\alpha_2)$* **($\mathbf{P_5}$)**. Supposons $\alpha_1 = \alpha_2 \in I$. Par ATM-EQ, on a $\alpha_1 \leq \alpha_2 \in^* I$ et $\alpha_2 \leq \alpha_1 \in^* I$. Par ($\mathrm{P}_1$), on en déduit $\mathrm{gskel}(\alpha_1) = \mathrm{gskel}(\alpha_2)$, $\mathrm{lbs}_{\leq}(\alpha_1) = \mathrm{lbs}_{\leq}(\alpha_2)$ et $\mathrm{lbs}_{\lessdot}(\alpha_1) = \mathrm{lbs}_{\lessdot}(\alpha_2)$. Par (6), $\varphi(\alpha_1) = \varphi(\alpha_2)$ s'ensuit.

∘ *Si $\alpha = \ell \in I$ alors $\varphi(\alpha) = \ell$* **(23)**. Supposons $\alpha = \ell \in I$ **(24)**. Par la définition de $V^\bullet$, il existe $\beta \in V^\bullet$ tel que $\alpha = \beta \in I$ **(25)** et $\beta = \ell \in I'$ **(26)**. Par les propriétés (P_2) et (P_3), on a $\varphi(\beta) = \varphi'(\beta)$. De plus, par ($\mathrm{P}_5$), (25) implique $\varphi(\alpha) = \varphi(\beta)$. Enfin, (26) et (2) impliquent $\varphi'(\beta) = \ell$. On en conclut que $\varphi(\alpha) = \ell$.

∘ *Si* $\alpha_1 \leq \alpha_2 \in I$ *alors* $\varphi(\alpha_1) \leq \varphi(\alpha_2)$ **($\mathbf{P_6}$)**. Supposons $\alpha_1 \leq \alpha_2 \in I$. Par ($P_1$), on en déduit $\mathrm{gskel}(\alpha_1) = \mathrm{gskel}(\alpha_2)$, $\mathrm{lbs}_{\leq}(\alpha_1) \subseteq \mathrm{lbs}_{\leq}(\alpha_2)$ et $\mathrm{lbs}_{\lessdot}(\alpha_1) \subseteq \mathrm{lbs}_{\lessdot}(\alpha_2)$. Par (6), on en déduit $\varphi(\alpha_1) \leq \varphi(\alpha_2)$.

∘ *Si* $\alpha_1 \lessdot \alpha_2 \in I$ *alors* $\varphi(\alpha_1) \lessdot \varphi(\alpha_2)$ **($\mathbf{P_7}$)**. Supposons $\alpha_1 \lessdot \alpha_2 \in I$ **(27)**. Soit $t \in \mathrm{lbs}_{\leq}(\alpha_1)$ **(28)**. Il existe β tel que $t = \varphi'(\beta)$ **(29)** et $\beta \leq \alpha_1 \in^* I$ **(30)** et $\beta \in V^- \cup V^\bullet$ **(31)**. Par ATM-LEQ-GD, (30) et (27) donnent $\beta \lessdot \alpha_2 \in^* I$. Grâce à (29) et (31), on en déduit que $\mathrm{lift}(t/\,\mathrm{gskel}(\alpha_2)) \in \mathrm{lbs}_{\lessdot}(\alpha_2)$ puis, par (6) et la propriété 10.2 (page 173), $\mathrm{lift}(t/\,\mathrm{gskel}(\alpha_2)) \leq \varphi(\alpha_2)$. En utilisant la propriété 10.4 (page 174), on obtient finalement $t \lessdot \varphi(\alpha_2)$. En déchargeant l'hypothèse (28), cela donne $\forall t \in \mathrm{lbs}_{\leq}(\alpha_1)\ t \lessdot \varphi(\alpha_2)$ **(32)**. De même, soit $t \in \mathrm{lbs}_{\lessdot}(\alpha_1)$ **(33)**. Il existe β tel que $t = \mathrm{lift}(\varphi'(\beta)/\,\mathrm{gskel}(\alpha_1))$ **(34)** et $\beta \lessdot \alpha_1 \in^* I$ **(35)** et $\beta \in V^- \cup V^\bullet$ **(36)**. Par ATM-GD-GD, (35) et (27) donnent $\beta \lessdot \alpha_2 \in^* I$. Grâce à (36), on en déduit que $\mathrm{lift}(\varphi'(\beta)/\,\mathrm{gskel}(\alpha_2)) \in \mathrm{lbs}_{\lessdot}(\alpha_2)$. Or, par la propriété 10.4 (page 174), $\mathrm{lift}(\varphi'(\beta)/\,\mathrm{gskel}(\alpha_1)) \lessdot \mathrm{lift}(\varphi'(\beta)/\,\mathrm{gskel}(\alpha_2))$. Par (6) et la propriété 10.2 (page 173), on en déduit $t \lessdot \varphi(\alpha_2)$. En déchargeant l'hypothèse (33), on a $\forall t \in \mathrm{lbs}_{\lessdot}(\alpha_1)\ t \lessdot \varphi(\alpha_2)$ **(37)**. On déduit de (32) et (37) que $\forall t \in \left(\{\bot_{\mathrm{gskel}(\alpha)}\} \cup \mathrm{lbs}_{\leq}(\alpha) \cup \mathrm{lbs}_{\lessdot}(\alpha)\right)\ t \lessdot \varphi(t_2)$. Par (6) et la propriété 10.2 (page 173), cela permet de conclure $\varphi(t_1) \lessdot \varphi(t_2)$.

Je montre maintenant que $\varphi \leq^{\Pi} \varphi'$ **($\mathbf{C_3}$)**. Soit $\alpha \in \mathcal{V}$ **(38)**. Si $- \in \Pi(\alpha)$ **(39)**, alors il existe $\beta \in V^-$ **(40)** tel que $\alpha = \beta \in I$ **(41)**. Par (P_2), on a $\varphi'(\beta) \leq \varphi(\beta)$ **(42)** et, par (P_5), (41) implique $\varphi(\alpha) = \varphi(\beta)$ **(43)**. De plus, par (39) et (40), (41) donne $\alpha = \beta \in I'$. Il s'ensuit, par (2), $\varphi'(\alpha) = \varphi'(\beta)$ **(44)**. On déduit de (44), (42) et (43) que $\varphi'(\alpha) \leq \varphi(\alpha)$. On montre de même, en exploitant (P_3), que si $+ \in \Pi(\alpha)$ alors $\varphi(\alpha) \leq \varphi'(\alpha)$. On en déduit que $\varphi(\alpha) \leq^{\Pi(\alpha)} \varphi'(\alpha)$, ce qui donne ($C_3$), en déchargeant (38).

Enfin, en déchargeant l'hypothèse (2) sur les résultats intermédiaires (C_2) et (C_3), on obtient le but (C_1) : $I' \Vdash I \mod \Pi$. ⌟

On peut appliquer la procédure de clôture polarisée à la partie atomique d'une contrainte réduite dépoussiérée.

Corollaire 10.53 *Soit* $I = \mathbb{X}[I_s \wedge I_a]$ *une contrainte bien marquée, unifiée, acyclique, réduite et dépoussiérée pour* Π*, écrite sous la forme donnée à la définition 10.48 (page 209). Soit* I'_a *la clôture polarisée de* I_a *sous* $\Pi[\mathrm{dtv}(\mathbb{X}) \mapsto \varnothing] \otimes I_s$*. Alors* $\mathbb{X}[I_s \wedge I'_a] \equiv I \mod \Pi$ *et* $\mathbb{X}[I_s \wedge I'_a]$ *est bien marquée, unifiée, acyclique et réduite.* □

▶ **Minimisation** L'objectif de la deuxième transformation que je propose pour les contraintes atomiques est d'identifier des variables qui jouent des rôles exactement identiques. J'entends ici par « rôle » d'une variable la façon dont elle est reliée aux autres variables, à travers les prédicats de la contrainte, ainsi que, par sa polarité, au contexte. La dénomination de cette transformation est due à Pottier [Pot98, Pot01b] ; elle est empruntée à la théorie des automates : *minimiser* un automate consiste à en déterminer une partition en classes d'états équivalents, pour construire l'automate quotient qui reconnaît le même langage, en étant de taille minimale.

D'une manière similaire, la minimisation d'une contrainte atomique est réalisée en introduisant une relation d'équivalence entre variables — ou, plus précisément, entre multi-équations — puis en unifiant les variables d'une même classe d'équivalence. Comme je l'ai mentionné ci-dessus, deux variables peuvent être considérées équivalentes si elles sont reliées aux autres variables et au contexte de façons identiques, c'est-à-dire :

- Si elles sont toutes deux non-positives, dans le même multi-squelette et dotées des mêmes successeurs pour $\leq$ et $\lessdot$,
- ou bien si elles sont toutes deux non-négatives, dans le même multi-squelette et dotées des mêmes prédécesseurs pour $\leq$ et $\lessdot$.

Ce critère est précisé par la définition suivante. Étant donnée une contrainte atomique I, j'écris $\bar{\bar{\tau}} \in I$ si $\bar{\bar{\tau}}$ est une multi-équation de I, *i.e.* I peut s'écrire $\langle \bar{\bar{\tau}} \rangle^\iota \approx \bar{\bar{\tau}} \wedge I'$. J'utilise les notations

suivantes pour désigner les ensembles de prédécesseurs et de successeurs de chaque multi-équation *sans constante* :

$$\begin{aligned}
\text{leq-succ}_I(\bar\alpha) &= \{\, \bar{\bar\tau} \in I \mid \exists \alpha \in \bar\alpha \;\; \exists \beta \in \bar{\bar\tau} \quad \alpha \le \beta \in I \,\} \\
\text{gd-succ}_I(\bar\alpha) &= \{\, \bar{\bar\tau} \in I \mid \exists \alpha \in \bar\alpha \;\; \exists \beta \in \bar{\bar\tau} \quad \alpha \lessdot \beta \in I \,\} \\
\text{leq-pred}_I(\bar\alpha) &= \{\, \bar{\bar\tau} \in I \mid \exists \alpha \in \bar\alpha \;\; \exists \beta \in \bar{\bar\tau} \quad \beta \le \alpha \in I \,\} \\
\text{gd-pred}_I(\bar\alpha) &= \{\, \bar{\bar\tau} \in I \mid \exists \alpha \in \bar\alpha \;\; \exists \beta \in \bar{\bar\tau} \quad \beta \lessdot \alpha \in I \,\}
\end{aligned}$$

Définition 10.54 (Multi-équations équivalentes) *Soit I une contrainte atomique et unifiée. Soit Π une polarisation. Soient $\bar{\bar\alpha}_1$ et $\bar{\bar\alpha}_2$ deux multi-équations de I. On a $\bar{\bar\alpha}_1 \sim_{I/\Pi} \bar{\bar\alpha}_2$ si et seulement si $\bar{\bar\alpha}_1$ et $\bar{\bar\alpha}_2$ appartiennent au même multi-squelette et*

(i) $\Pi(\bar{\bar\alpha}_1) = \Pi(\bar{\bar\alpha}_2) = -$, $\text{leq-succ}_I(\bar{\bar\alpha}_1) = \text{leq-succ}_I(\bar{\bar\alpha}_2)$ *et* $\text{gd-succ}_I(\bar{\bar\alpha}_1) = \text{gd-succ}_I(\bar{\bar\alpha}_2)$
ou bien

(ii) $\Pi(\bar{\bar\alpha}_1) = \Pi(\bar{\bar\alpha}_2) = +$, $\text{leq-pred}_I(\bar{\bar\alpha}_1) = \text{leq-pred}_I(\bar{\bar\alpha}_2)$ *et* $\text{gd-pred}_I(\bar{\bar\alpha}_1) = \text{gd-pred}_I(\bar{\bar\alpha}_2)$ □

La minimisation consiste à unifier les multi-équations équivalentes pour $\sim_{I/\Pi}$ dans la contrainte I. La correction de ce procédé est donnée par le prochain théorème.

Théorème 10.55 (Minimisation) *Soit I une contrainte atomique bien marquée et unifiée et Π une polarisation. On a*

$$I \equiv I \wedge \big(\bigwedge \{\, \bar{\bar\alpha}_1 = \bar{\bar\alpha}_2 \mid \bar{\bar\alpha}_1 \sim_{I/\Pi} \bar{\bar\alpha}_2 \,\} \big) \mod \Pi$$

⌜ *Preuve.* Puisque l'implication $I \wedge (\bigwedge_{\bar{\bar\alpha}_1 \sim_{I/\Pi} \bar{\bar\alpha}_2} \bar{\bar\alpha}_1 = \bar{\bar\alpha}_2) \Vdash I$ est immédiate, il suffit de montrer $I \Vdash I \wedge (\bigwedge_{\bar{\bar\alpha}_1 \sim_{I/\Pi} \bar{\bar\alpha}_2} \bar{\bar\alpha}_1 = \bar{\bar\alpha}_2) \mod \Pi$ **($\mathbf{C_1}$)** pour conclure. Soit φ une solution de I **(1)**. Je note V^+ (respectivement V^-) l'ensemble des variables de type membre d'une multi-équation $\bar{\bar\alpha}$ de I telle que $\Pi(\bar{\bar\alpha}) = +$ (respectivement $\Pi(\bar{\bar\alpha}) = -$). Puisque I est bien marquée, unifiée et atomique, V^+ et V^- sont disjoints, et toute variable α de $V^+ \cup V^-$ apparaît dans exactement une multi-équation de I. Je la note $\text{meq}_I(\alpha)$. Par abus de notation, si $\alpha_1, \alpha_2 \in V^+ \cup V^-$, j'écris $\alpha_1 \sim_{I/\Pi} \alpha_2$ pour $\text{meq}_I(\alpha_1) \sim_{I/\Pi} \text{meq}_I(\alpha_2)$.

Soient α_1 et α_2 tels que $\alpha_1 \sim_{I/\Pi} \alpha_2$. Par définition, on a $\alpha_1 \approx \alpha_2 \in I$. Par (1), on en déduit que $\varphi(\alpha_1) \approx \varphi(\alpha_2)$. On peut donc définir une affectation φ' comme suit :

$$\varphi'(\alpha) = \begin{cases} \bigsqcup \{\, \varphi(\beta) \mid \beta \in V^- \text{ et } \beta \sim_{I/\Pi} \alpha \,\} & \text{si } \alpha \in V^- \\ \bigsqcap \{\, \varphi(\beta) \mid \beta \in V^+ \text{ et } \beta \sim_{I/\Pi} \alpha \,\} & \text{si } \alpha \in V^+ \\ \varphi(\alpha) & \text{sinon} \end{cases}$$

On a par construction, pour tout $\alpha \in \mathcal{V}$, $\varphi(\alpha) \approx \varphi'(\alpha)$ **(2)**.

Vérifions tout d'abord que $\varphi' \le^\Pi \varphi$ **($\mathbf{C_2}$)**. Soit α une variable de type **(3)**, montrons $\varphi'(\alpha) \le^{\Pi(\alpha)} \varphi(\alpha)$ **(4)**. Trois cas sont à envisager :

∘ *Si* $\alpha \in V^-$. On a alors $\varphi'(\alpha) = \bigsqcup \{\, \varphi(\beta) \mid \beta \sim_{I/\Pi} \alpha \,\}$ **(5)** et $\Pi(\alpha) \subseteq -$ **(6)**. Puisque $\sim_{I/\Pi}$ est une relation d'équivalence, on a $\alpha \sim_{I/\Pi} \alpha$. On en déduit, par (5), $\varphi(\alpha) \le \varphi'(\alpha)$. Grâce à (6), on conclut $\varphi'(\alpha) \le^{\Pi(\alpha)} \varphi(\alpha)$.

∘ *Le cas* $\alpha \in V^+$ est symétrique du précédent.

∘ *Sinon,* on a $\varphi'(\alpha) = \varphi(\alpha)$. On en déduit que $\varphi'(\alpha) \le^{\Pi(\alpha)} \varphi(\alpha)$.

En déchargeant l'hypothèse (3) sur (4), on obtient le but (C_2). Notons que nous avons également montré que $\forall \alpha \in \mathcal{V} \backslash V^+ \;\; \varphi(\alpha) \le \varphi'(\alpha)$ **(7)** et $\forall \alpha \in \mathcal{V} \backslash V^- \;\; \varphi'(\alpha) \le \varphi(\alpha)$ **(8)**. Montrons maintenant que $\varphi' \vdash I$ **(9)**. Puisque I est atomique et unifiée, il me suffit d'établir les propriétés suivantes.

∘ *Si* $\alpha_1 \approx \alpha_2 \in I$ *alors* $\varphi'(\alpha_1) \approx \varphi'(\alpha_2)$ **($\mathbf{P_1}$)**. Supposons $\alpha_1 \approx \alpha_2 \in I$. Par (1), on a $\varphi(\alpha_1) \approx \varphi(\alpha_2)$. De plus, par (2), on a $\varphi'(\alpha_1) \approx \varphi(\alpha_1)$ et $\varphi'(\alpha_2) \approx \varphi(\alpha_2)$. Par transitivité, on en conclut $\varphi'(\alpha_1) \approx \varphi'(\alpha_2)$.

∘ *Si* $\alpha_1 = \alpha_2 \in I$ *alors* $\varphi'(\alpha_1) = \varphi'(\alpha_2)$ **($\mathbf{P_2}$)**. Supposons $\alpha_1 = \alpha_2 \in I$. Puisque α_1 et α_2 appartiennent à la même multi-équation de I, on est dans l'un des trois cas suivants.

· *Soit* $\alpha_1, \alpha_2 \in V^-$. On a alors $\alpha_1 \sim_{I/\Pi} \alpha_2$. Puisque $\sim_{I/\Pi}$ est une relation d'équivalence, on en déduit que $\{\, \beta \mid \beta \sim_{I/\Pi} \alpha_1 \,\} = \{\, \beta \mid \beta \sim_{I/\Pi} \alpha_2 \,\}$, puis que $\varphi'(\alpha_1) = \varphi'(\alpha_2)$.

· *Soit* $\alpha_1, \alpha_2 \in V^+$. Ce cas est symétrique du précédent.

· *Soit* $\alpha_1, \alpha_2 \notin V^+ \cup V^-$. On a alors $\varphi'(\alpha_1) = \varphi(\alpha_1)$ et $\varphi'(\alpha_2) = \varphi(\alpha_2)$. Or, par (1), $\varphi(\alpha_1) = \varphi(\alpha_2)$. Par transitivité, $\varphi'(\alpha_1) = \varphi'(\alpha_2)$ s'ensuit.

∘ *Si* $\alpha = \ell \in I$ *alors* $\varphi'(\alpha) = \ell$ **(P₃)**. Supposons $\alpha = \ell \in I$. Par construction, on a $\alpha \notin V^+ \cup V^-$ (puisque ces ensembles contiennent des variables appartenant à une multi-équation sans constante et I est unifiée), donc $\varphi'(\alpha) = \varphi(\alpha)$. De plus, (1) implique $\varphi(\alpha) = \ell$. Par transitivité, on en déduit $\varphi'(\alpha) = \ell$.

∘ *Si* $\alpha_1 \leq \alpha_2 \in I$ *alors* $\varphi'(\alpha_1) \leq \varphi'(\alpha_2)$ **(P₄)**. Supposons $\alpha_1 \leq \alpha_2 \in I$ **(10)**, par (1), on a $\varphi(\alpha_1) \leq \varphi(\alpha_2)$ **(11)**. Quatre cas sont envisageables.

· *Si* $\alpha_1 \notin V^-$ *et* $\alpha_2 \notin V^+$. Par (8) et (7), on a respectivement $\varphi'(\alpha_1) \leq \varphi(\alpha_1)$ et $\varphi(\alpha_2) \leq \varphi'(\alpha_2)$. Par (11), on en déduit $\varphi'(\alpha_1) \leq \varphi'(\alpha_2)$.

· *Si* $\alpha_1 \in V^-$ *et* $\alpha_2 \notin V^+$. Soit $\beta_1 \in V^-$ tel que $\beta_1 \sim_{I/\Pi} \alpha_1$ **(12)**. On a $\text{leq-succ}_I(\text{meq}_I(\alpha_1)) = \text{leq-succ}_I(\text{meq}_I(\beta_1))$. Par (10), on en déduit que $\alpha_2 \in \text{leq-succ}_I(\text{meq}_I(\beta_1))$ et, grâce à (1), on en déduit que $\varphi(\beta_1) \leq \varphi(\alpha_2)$. Puisque $\alpha_1 \in V^-$, en déchargeant (12), $\varphi'(\alpha_1) \leq \varphi(\alpha_2)$ s'ensuit. Par (7), on en conclut $\varphi'(\alpha_1) \leq \varphi'(\alpha_2)$.

· *Si* $\alpha_1 \notin V^-$ *et* $\alpha_2 \in V^+$. Ce cas est symétrique au précédent.

· *Si* $\alpha_1 \in V^-$ *et* $\alpha_2 \in V^+$. Soient $\beta_1 \in V^-$ tel que $\beta_1 \sim_{I/\Pi} \alpha_1$ **(13)** et $\beta_2 \in V^+$ tel que $\beta_2 \sim_{I/\Pi} \alpha_2$ **(14)**. On a $\text{leq-succ}_I(\text{meq}_I(\beta_1)) = \text{leq-succ}_I(\text{meq}_I(\alpha_1))$ et $\text{leq-pred}_I(\text{meq}_I(\beta_2)) = \text{leq-pred}_I(\text{meq}_I(\alpha_2))$. On en déduit qu'il existe $\beta_1' \in \text{meq}_I(\beta_1)$ et $\beta_2' \in \text{meq}_I(\beta_2)$ tels que $\beta_1' \leq \beta_2' \in I$. Par (1), $\varphi(\beta_1') \leq \varphi(\beta_2')$ puis $\varphi(\beta_1) \leq \varphi(\beta_2)$ s'ensuivent. Puisque $\alpha_1 \in V^-$ et $\alpha_2 \in V^+$, en déchargeant (13) et (14), on obtient $\varphi'(\alpha_1) \leq \varphi'(\alpha_2)$.

∘ *Si* $\alpha_1 \lessdot \alpha_2 \in I$ *alors* $\varphi'(\alpha_1) \leq \varphi'(\alpha_2)$ **(P₅)**. Supposons $\alpha_1 \lessdot \alpha_2 \in I$ **(15)**, par (1), on a $\varphi(\alpha_1) \lessdot \varphi(\alpha_2)$ **(16)**. Quatre cas sont envisageables.

· *Si* $\alpha_1 \notin V^-$ *et* $\alpha_2 \notin V^+$. Par (8) et (7), on a respectivement $\varphi'(\alpha_1) \leq \varphi(\alpha_1)$ et $\varphi(\alpha_2) \leq \varphi'(\alpha_2)$. Par (16), on en déduit, grâce au propriété 10.2 (page 173), $\varphi'(\alpha_1) \leq \varphi'(\alpha_2)$.

· *Si* $\alpha_1 \in V^-$ *et* $\alpha_2 \notin V^+$. Soit $\beta_1 \in V^-$ tel que $\beta_1 \sim_{I/\Pi} \alpha_1$ **(17)**. On a $\text{gd-succ}_I(\text{meq}_I(\alpha_1)) = \text{gd-succ}_I(\text{meq}_I(\beta_1))$. Par (15), on en déduit que $\alpha_2 \in \text{gd-succ}_I(\text{meq}_I(\beta_1))$ et, grâce à (1), $\varphi(\beta_1) \lessdot \varphi(\alpha_2)$ s'ensuit. Puisque $\alpha_1 \in V^-$, en déchargeant (17), on en déduit, grâce à la propriété 10.2 (page 173), que $\varphi'(\alpha_1) \lessdot \varphi(\alpha_2)$. En utilisant (7), on en conclut, grâce au propriété 10.2 (page 173), $\varphi'(\alpha_1) \lessdot \varphi'(\alpha_2)$.

· *Si* $\alpha_1 \notin V^-$ *et* $\alpha_2 \in V^+$. Ce cas est symétrique au précédent.

· *Si* $\alpha_1 \in V^-$ *et* $\alpha_2 \in V^+$. Soient $\beta_1 \in V^-$ tel que $\beta_1 \sim_{I/\Pi} \alpha_1$ **(18)** et $\beta_2 \in V^+$ tel que $\beta_2 \sim_{I/\Pi} \alpha_2$ **(19)**. On a $\text{gd-succ}_I(\text{meq}_I(\beta_1)) = \text{gd-succ}_I(\text{meq}_I(\alpha_1))$ et $\text{gd-pred}_I(\text{meq}_I(\beta_2)) = \text{gd-pred}_I(\text{meq}_I(\alpha_2))$. On en déduit qu'il existe $\beta_1' \in \text{meq}_I(\beta_1)$ et $\beta_2' \in \text{meq}_I(\beta_2)$ tels que $\beta_1' \lessdot \beta_2' \in I$. Par (1), $\varphi(\beta_1') \lessdot \varphi(\beta_2')$ puis $\varphi(\beta_1) \lessdot \varphi(\beta_2)$ s'ensuivent. Puisque $\alpha_1 \in V^-$ et $\alpha_2 \in V^+$, en déchargeant (13) et (14), on obtient, grâce à la propriété 10.2 (page 173), $\varphi'(\alpha_1) \leq \varphi'(\alpha_2)$.

Par ailleurs, de par la définition de φ', si $\alpha_1 \sim_{I/\Pi} \alpha_2$ alors $\varphi'(\alpha_1) = \varphi'(\alpha_2)$. On en déduit que $\varphi' \vdash (\bigwedge_{\bar\alpha_1 \sim_{I/\Pi} \bar\alpha_2} \bar\alpha_1 = \bar\alpha_2)$ **(C₃)**. En déchargeant l'hypothèse (1) sur (C_2), (9) et (C_3), on obtient le but (C_1). ⌟

La minimisation peut être appliquée à la partie atomique d'une contrainte réduite, comme expliqué par le théorème suivant. La contrainte $I_a \wedge \left(\bigwedge\{\, \bar\alpha_1 = \bar\alpha_2 \mid \bar\alpha_1 \sim_{I/\Pi[\bar\alpha \mapsto \varnothing] \otimes I_s} \bar\alpha_2 \,\}\right)$ peut être lue comme la superposition de I_a avec la demande d'unifier les multi-équations de I_a trouvées équivalentes. Elle doit donc être transmise à l'algorithme d'unification, de manière à obtenir une forme unifiée.

Corollaire 10.56 *Soit* $I = \mathbb{X}[I_s \wedge I_a]$ *une contrainte bien marquée, unifiée, acyclique et réduite, écrite sous la forme donnée à la définition 10.27 (page 189). Soit* I'_a *telle que* $I_a \wedge \big(\bigwedge\{\, \bar{\bar{\alpha}}_1 = \bar{\bar{\alpha}}_2 \mid \bar{\bar{\alpha}}_1 \sim_{I/\Pi[\mathrm{dtv}(\mathbb{X})\mapsto\varnothing]\otimes I_s} \bar{\bar{\alpha}}_2 \,\}\big) \twoheadrightarrow_u^{**} I'_a$*. Alors* $\mathbb{X}[I_s \wedge I'_a] \equiv I \mod \Pi$ *et* $\mathbb{X}[I_s \wedge I'_a]$ *est bien marquée, unifiée, acyclique et réduite.* □

Pour être pleinement efficace, la minimisation d'une contrainte atomique doit être effectuée après sa clôture polarisée. (Cet ordre n'est cependant pas nécessaire à la correction ni de l'une ni de l'autre des procédures.) En effet, l'élimination des variables apolaires et la clôture transitive du graphe de contraintes sont susceptibles de faire apparaître de nouvelles équivalences entre multi-équations. De plus, dans une contrainte issue de la clôture polarisée, les successeurs d'une variable négative (respectivement les prédécesseurs d'une variable positive) sont des variables positives (respectivement négatives) ou unifiées avec une constante. Le sous-ensemble du graphe de contraintes considéré par la minimisation est donc bi-parti, avec d'un côté les multi-équations positives et de l'autre les multi-équations négatives. Cette observation garantit que les unifications réalisées par la minimisation ne peuvent pas faire apparaître de nouvelles équivalences entre contraintes. En d'autres termes, il n'est pas utile d'itérer plusieurs fois ce processus ou de procéder par raffinement successifs de la relation d'équivalence $\sim_{I/\Pi}$. Il est cependant utile d'appliquer la procédure de réduction des chaînes après la minimisation, comme l'illustre l'exemple discuté à la fin de cette section.

L'application de techniques de minimisation aux systèmes de contraintes est due à Felleisen et Flanagan [FF96, FF97], qui l'ont appliquée au cas des *set constraints*. Elle a été reprise et adaptée par Pottier [Pot98, Pot01b] dans le cadre du sous-typage non-structurel. Cependant, les contraintes manipulées par Pottier ne peuvent être décomposées en une partie structurelle et une partie atomique, de telle sorte que la minimisation doit pouvoir s'appliquer à une contrainte comportant des types construits. Cela donne lieu à une définition plus complexe, car récursive, de la notion d'équivalence entre variables, dont le calcul s'effectue par raffinements successifs, grâce à une variante de l'algorithme de Hopcroft pour la minimisation des automates finis. Ici, le calcul de la relation $\sim_{I/\Pi}$, qui s'effectue au seul niveau atomique, est direct. Les équivalences entre types construits, c'est-à-dire entre variables non terminales, peuvent être trouvées, après par la minimisation au noyau atomique, par une simple passe de *hash-consing*.

▶ **Hash-consing** Le *hash-consing* permet de propager, le long de la structure des types, les équivalences trouvées par la minimisation (ou par les autres procédures de simplification) entre des variables terminales, de manière à obtenir un partage maximal. L'idée est la même que dans les systèmes à base d'unification : elle consiste à identifier des nœuds ayant des fils deux à deux égaux. Cependant, elle doit ici être réalisée à deux niveaux : celui des multi-squelettes puis celui des multi-équations. Cette procédure est formalisé par le système de réécriture donné figure 10.15. Comme à l'habitude, la réduction peut être effectuée sous contexte arbitraire. La règle SPH-ATOM identifie deux multi-équations contenant la même constante atomique. Les règles SPH-UNIFY-$\approx$ et SPH-UNIFY-$=$ peuvent être vues comme des versions retournées de SPU-DEC-$\approx$ et SPU-DEC-$=$: elles fusionnent deux multi-squelettes ou deux multi-équations dont les fils sont eux-mêmes reliés deux à deux. Naturellement, une stratégie de réduction efficace consiste à appliquer ces règles sur les multi-squelettes en remontant la structure des types, c'est-à-dire dans l'ordre inverse de celui pratiqué pour l'expansion.

Théorème 10.57 (Hash-consing) *Les règles de la figure 10.15 préservent le caractère bien marqué et unifié, ainsi que la sémantique des contraintes. Elles terminent.* □

▶ **Exemple** La figure 10.14-b (page 207) poursuit le traitement de l'exemple commencé à la section 10.3.2 (page 199), en considérant le graphe atomique des inégalités portées par le dernier multi-squelette. La clôture polarisée permet d'éliminer les variables apolaires β_1 et β_2, en reliant

$$\langle \bar{\bar{\alpha}}_1 = \ell \rangle^\bullet \approx \langle \bar{\bar{\alpha}}_2 = \ell \rangle^\bullet \approx \bar{\bar{\tau}} \;\rightarrowtriangle_h\; \langle \bar{\bar{\alpha}}_1 = \bar{\bar{\alpha}}_2 = \ell \rangle^\bullet \approx \bar{\bar{\tau}} \qquad \text{SPH-ATOM}$$

$$I \wedge \langle \bar{\bar{\tau}} = d\,\vec{\alpha} \rangle^\bullet \approx \bar{\bar{\tau}} \wedge \langle \bar{\bar{\tau}}' = d\,\vec{\beta} \rangle^\bullet \approx \bar{\bar{\tau}}' \;\rightarrowtriangle_h\; I \wedge \langle \bar{\bar{\tau}} = d\,\vec{\alpha} \rangle^\bullet \approx \bar{\bar{\tau}} \approx \langle \bar{\bar{\tau}}' = d\,\vec{\beta} \rangle^\circ \approx \bar{\bar{\tau}}' \quad si\ \forall i \in [1,n]\ \ \vec{\alpha}_{|i} \approx^{d.i} \vec{\beta}_{|i} \in I \qquad \text{SPH-UNIFY-}\approx$$

$$I \wedge \langle \bar{\bar{\tau}} = d\,\vec{\alpha} \rangle^\iota \approx \langle \bar{\bar{\tau}}' = d\,\vec{\beta} \rangle^\circ \approx \bar{\bar{\tau}} \;\rightarrowtriangle_h\; I \wedge \langle \bar{\bar{\tau}} = d\,\vec{\alpha} \rangle^\iota \approx \langle \bar{\bar{\tau}}' \rangle^\circ \approx \bar{\bar{\tau}} \quad si\ \forall i \in [1,n]\ \ \vec{\alpha}_{|i} = \vec{\beta}_{|i} \in I \qquad \text{SPH-UNIFY-}=$$

Figure 10.15 – Hash-consing

directement les variables négatives β_5 et β_6 aux variables positives β_3 et β_4 par quatre inégalités. La figure obtenue, une « 2-2-couronne », ne peut pas être simplifiée par la réduction des chaînes, puisque chacune des variables positives β_1 et β_2 a deux bornes inférieures, et chacune des variables négatives β_5 et β_6 deux bornes supérieures. Ces variables sont cependant deux à deux équivalentes pour la procédure de minimisation, ce qui permet de réduire le graphe à une simple chaîne, laquelle peut ensuite être réduite. Le hash-consing permet de propager ces simplifications effectuées sur le graphe terminal au niveau supérieur, en partageant la structure des variables α_2, α_5, α_7 et α_8 (figure 10.14-c, page 207). Après résolution et simplification, le schéma obtenu pour la fonction f est :

$$\forall\alpha\left[\begin{array}{r}\exists\alpha_2\alpha_3\alpha_5\alpha_6\alpha_7\alpha_8\beta_5.(\alpha = \alpha_2 \to \alpha_3 \wedge \alpha_3 = \alpha_5 \to \alpha_6 \wedge \alpha_6 = \alpha_7 \times \alpha_8 \\ \wedge\ \alpha_2 = \alpha_5 = \alpha_7 = \alpha_8 = \mathsf{int}\,\beta_5)\end{array}\right].\alpha$$

La contrainte portée par ce schéma décrit la structure du type α, en introduisant un nom pour chaque nœud intermédiaire. En général, lorsqu'on affiche un tel schéma à l'utilisateur, on préfère donner une représentation sous forme d'arbre :

$$\forall\beta_5[\mathsf{true}].\,\mathsf{int}\,\beta_5 \to \mathsf{int}\,\beta_5 \to \mathsf{int}\,\beta_5 \times \mathsf{int}\,\beta_5$$

Celle-ci est légèrement moins riche que la précédente — puisqu'elle ne montre pas que le nœud $\mathsf{int}\,\beta_5$ est partagé entre les différents sous-termes dans la représentation interne — mais elle est plus lisible.

De manière générale, l'algorithme de résolution et de simplification présenté dans ce chapitre permet de présenter à l'utilisateur les schémas de type clos sous une forme relativement simple à interpréter. En effet, grâce à LOG-LET-EX, après résolution de sa contrainte, un schéma clos peut être écrit $\forall\bar{\alpha}\bar{\beta}[I_s \wedge I_a].\tau$ où I_s est une contrainte structurelle de support $\bar{\alpha}$ et I_a une contrainte atomique telle que $\mathrm{ftv}(I_a) \subseteq \bar{\alpha}$. De plus, grâce au dépoussiérage et à la clôture polarisée, toutes les variables libres de I_s et I_a sont accessibles depuis une variable polaire, c'est-à-dire, dans le cas d'un schémas clos, depuis la racine τ. Ainsi, on peut substituer itérativement les variables $\bar{\alpha}$ dans le type τ par leurs descripteurs dans I_s, ce qui permet d'obtenir un type τ' tel que $I_s \Vdash \tau = \tau'$ et $\mathrm{ftv}(\tau') \subseteq \bar{\beta}$. Puisque I_a implique $\exists\bar{\alpha}.(I_s \wedge I_a)$, le schéma peut alors être écrit $\forall\bar{\alpha}[I_a].\tau'$. Cette forme est particulièrement intéressante car toute la structure est ici décrite par la racine τ' et la contrainte I_a ne porte que sur les variables qui apparaissent à ses feuilles.

CHAPITRE ONZE

Du solveur primaire à un solveur complet

Après avoir présenté un algorithme efficace de résolution et de simplification pour un sous-ensemble du langage de contraintes, formé des conjonctions d'inégalités et de gardes, je montre maintenant comment obtenir un solveur pour le langage complet. Toutefois, j'effectue ce développement dans un cadre général, sans faire d'hypothèse particulière sur les prédicats p de la logique, qui peuvent à nouveau être arbitraires. Je suppose simplement que je dispose d'un algorithme de résolution — tel que celui décrit au chapitre précédent — pour les ***contraintes primaires*** définies par la grammaire

$$I ::= p\ \vec{\tau} \mid I \wedge I \mid \exists \bar{\alpha}.I$$

Ce solveur primaire est donné par une relation de réécriture $-\pi\rightarrowtriangle$ entre contraintes primaires qui vérifie les trois propriétés suivantes.

Hypothèse 11.1 (Correction) *Si* $I -\pi\rightarrowtriangle I'$ *alors* $I \equiv I' \mod \Pi$. □

Hypothèse 11.2 (Terminaison) *La relation* $-\pi\rightarrowtriangle$ *est fortement normalisante. Si* $\exists\bar{\alpha}.I$ *est une forme normale pour* $-\pi\rightarrowtriangle$ *alors* I *aussi.* □

Hypothèse 11.3 *Une forme normale pour* $-\pi\rightarrowtriangle$ *est soit satisfiable, soit égale à* false. □

Ainsi, la relation $-\pi\rightarrowtriangle$ donne un algorithme permettant de décider la satisfiabilité d'une contrainte primaire arbitraire. Sa sortie n'est cependant pas un seul Booléen, mais une contrainte équivalente à l'entrée (modulo la polarité Π) : le solveur primaire est également utilisé par la strate *secondaire* du solveur pour simplifier les contraintes primaires.

11.1 États de l'algorithme

Comme le corps de l'algorithme décrit au chapitre 10 (page 171), le solveur complet est défini par un ensemble de règles de réduction. Cependant, ces règles ne manipulent pas de simples contraintes : adoptant la même formalisation que Pottier et Rémy [PR03], je distingue en effet maintenant le langage externe de contraintes, qui permet au client de formuler les problèmes à résoudre, de celui utilisé pour décrire l'état interne de l'algorithme de résolution. Dans ce qui suit, la meta-variable

C dénote des contraintes externes, qui sont vues comme des arbres de syntaxe abstraits définis par la grammaire donnée à la section 1.4 (page 25). Les structures de données internes incluent les contraintes primaires I ainsi que les ***piles*** définies comme suit :

$$S ::= [\,] \mid S[[\,] \wedge C] \mid S[\exists \bar{\alpha}.[\,]] \mid S[I \wedge \mathsf{let}\, x : \forall \bar{\alpha}[[\,]].\tau \;\mathsf{in}\; C] \mid S[\mathsf{let}\, x : \forall \bar{\alpha}[I].\tau \;\mathsf{in}\; [\,]] \qquad \text{(pile)}$$

Chaque pile peut être vue comme un contexte de contrainte $\mathbb{C}$, ou bien comme une liste de ***cadres*** de l'une des quatre formes

$$[\,] \wedge C \qquad \exists \bar{\alpha}.[\,] \qquad I \wedge \mathsf{let}\, x : \forall \bar{\alpha}[[\,]].\tau \;\mathsf{in}\; C \qquad \mathsf{let}\, x : \forall \bar{\alpha}[I].\tau \;\mathsf{in}\; [\,]$$

appelées respectivement *cadre conjonction*, *cadre existentiel*, *cadre liaison* et *cadre environnement*. Dans une pile, les cadres peuvent être ajoutés (ou *empilés*) et supprimés (ou *dépilés*) au point le plus interne, c'est-à-dire au niveau du trou du contexte qu'elle représente. Les ensembles $\mathrm{dtv}(S)$ et $\mathrm{dpv}(S)$ des variables de type ou de programme définis par une pile sont définis comme pour un contexte.

Un ***état*** $S; I; C$ de l'algorithme est un triplet formé d'une pile S, d'une contrainte primaire I et d'une contrainte externe C. L'état $S; I; C$ doit être compris comme une représentation de la contrainte $S[I \wedge C]$. La notion d'α-équivalence entre états est défini conformément à cette interprétation. On peut expliquer le rôle de chaque composant d'un état comme suit. Tout d'abord, C représente la contrainte externe que le solveur s'apprête à examiner. Par la suite, j'utilise par convention à cette position la constante true pour représenter la situation où l'analyse de la contrainte externe est achevée. La contrainte I correspond quant-à-elle à l'état interne du solveur primaire utilisé par l'algorithme. Enfin, la pile S joue un double rôle. Les cadres conjonctions et de liaison enregistrent le travail restant à faire une fois que la contrainte C a été entièrement traitée. Les cadres existentiels permettent quant-à-eux d'enregistrer où les variables de type libres dans I et C sont liées ; les cadres d'environnement font de même avec les variables de programme, en leur associant également des schémas, comme dans un environnement traditionnel. Notons que les contraintes de ces schémas sont des contraintes primaires I, *i.e.* elles ont déjà été considérées par l'algorithme de résolution et traduites sous forme interne. L'état initial de l'algorithme de résolution est normalement $[\,]; \mathsf{true}; C$, où C est la contrainte externe à résoudre.

11.2 Définition de l'algorithme

L'algorithme de résolution consiste du système de réécriture (non-déterministe) entre états donné figure 11.1. La première règle, S-PRIM, permet au solveur primaire d'être appelé à tout moment, pour résoudre la contrainte I. Il reçoit la polarisation Π du contexte courant. Les quantifications existentielles introduites par le solveur primaire au sommet de la contrainte I peuvent directement être placées dans un cadre de la pile grâce à S-EX-PRIM.

Les règles S-EX-AND et S-EX-IN permettent de remonter les quantificateurs existentiels à travers les cadres conjonctions et environnement, jusqu'à ce qu'ils atteignent le sommet de la pile, ou un cadre liaison. Dans ce deuxième cas, ils peuvent être éliminés par S-EX-LET. Les conditions d'application de ces règles préviennent les captures de variables de type ; elles peuvent toujours être satisfaites par une α-conversion appropriée de l'état apparaissant dans le membre gauche. Dans un état normal pour ces trois règles, chaque variable de type qui apparaît dans un état est soit universellement quantifiée dans un schéma, soit quantifiée existentiellement au sommet soit libre. En d'autres termes, si ces règles sont appliquées avidement, les cadres existentiels n'ont pas besoin d'apparaître dans la représentation en machine des piles. À la place, il est suffisant de maintenir, dans chaque cadre liaison ou environnement une liste des variables de type qui sont universellement quantifiées dans le schéma, ainsi que, si l'état n'est pas clos, la liste de ses variables de type libres. Les règles S-PRED, S-INST, S-AND, S-EX et S-AND analysent la forme de la contrainte externe de l'état courant. Il y a une règle par construction du langage. S-PRED correspond au cas

$$
\begin{array}{rcll}
S; I; C & \to & S; I'; C & \textsc{s-prim} \\
 & & \textit{si } I \multimap_{\Pi} I' \textit{ et } \Pi \vdash S[[\,] \wedge C] & \\
S; \exists \bar{\alpha}.I; C & \to & S[\exists \bar{\alpha}.[\,]]; I; C & \textsc{s-ex-prim} \\
 & & \textit{si } \bar{\alpha} \# \mathrm{ftv}(C) & \\
S[(\exists \bar{\alpha}.S') \wedge D]; I; C & \to & S[\exists \bar{\alpha}.(S' \wedge D)]; I; C & \textsc{s-ex-and} \\
 & & \textit{si } \bar{\alpha} \# \mathrm{ftv}(D) & \\
S[\mathsf{let}\, x \,:\, \sigma \,\mathsf{in}\, \exists \bar{\alpha}.S']; I; C & \to & S[\exists \bar{\alpha}.\, \mathsf{let}\, x \,:\, \sigma \,\mathsf{in}\, S']; I; C & \textsc{s-ex-in} \\
 & & \textit{si } \bar{\alpha} \# \mathrm{ftv}(\sigma) & \\
S[I' \wedge \mathsf{let}\, x \,:\, \forall \bar{\alpha}[\exists \bar{\beta}.S'].\tau \,\mathsf{in}\, D]; I; C & \to & S[I' \wedge \mathsf{let}\, x \,:\, \forall \bar{\alpha}\bar{\beta}[S'].\tau \,\mathsf{in}\, D]; I; C & \textsc{s-ex-let} \\
 & & \textit{si } \bar{\beta} \# \mathrm{ftv}(\tau) & \\
S; I; p\, \vec{\tau} & \to & S; I \wedge p\, \vec{\tau}; \mathsf{true} & \textsc{s-pred} \\
 & & \textit{si } p \neq \mathsf{true} & \\
S; I; x \preceq \tau & \to & S; I; S(x) \preceq \tau & \textsc{s-inst} \\
S; I; C_1 \wedge C_2 & \to & S[[\,] \wedge C_2]; I; C_1 & \textsc{s-and} \\
S; I; \exists \bar{\alpha}.C & \to & S[\exists \bar{\alpha}.[\,]]; I; C & \textsc{s-ex} \\
 & & \textit{si } \bar{\alpha} \# \mathrm{ftv}(I) & \\
S; I; \mathsf{let}\, x \,:\, \forall \bar{\alpha}[C_1].\tau \,\mathsf{in}\, C_2 & \to & S[I \wedge \mathsf{let}\, x \,:\, \forall \bar{\alpha}[[\,]].\tau \,\mathsf{in}\, C_2]; \mathsf{true}; C_1 & \textsc{s-let} \\
S[[\,] \wedge C]; I; \mathsf{true} & \to & S; I; C & \textsc{s-pop-and} \\
S[I' \wedge \mathsf{let}\, x \,:\, \forall \bar{\alpha}[[\,]].\tau \,\mathsf{in}\, C]; I; \mathsf{true} & \to & S[\mathsf{let}\, x \,:\, \forall \bar{\alpha}[I].\tau \,\mathsf{in}\, [\,]]; I'; C & \textsc{s-pop-let} \\
S[\mathsf{let}\, x \,:\, \forall \bar{\alpha}[I'].\tau \,\mathsf{in}\, [\,]]; I; \mathsf{true} & \to & S[\exists \bar{\alpha}.[\,]]; I \wedge I'; \mathsf{true} & \textsc{s-pop-env} \\
 & & \textit{si } \bar{\alpha} \# \mathrm{ftv}(I) &
\end{array}
$$

Figure 11.1 – Algorithme de résolution

où un prédicat est découvert. Celui-ci est alors déplacé dans la contrainte interne, afin de pouvoir être traité par le solveur primaire. La condition $p \neq \mathsf{true}$ tient de l'emploi particulier de true dans la formalisation du solveur qui sert à représenter la fin de l'exploration de la contrainte : le cas $p = \mathsf{true}$ sera considéré par les règles S-POP-AND, S-POP-LET et S-POP-AND. Lorsqu'une contrainte d'instanciation $x \preceq \tau$ est atteinte, le solveur la remplace par $\sigma \preceq \tau$, où $\sigma = S(x)$ est le schéma de type associé à la variable x par la pile S. Il est défini comme suit :

$$\begin{aligned}
S[[\,] \wedge C](x) &= S(x) \\
S[\exists \bar{\alpha}.[\,]](x) &= S(x) \quad \text{si } \bar{\alpha} \# \mathrm{ftv}(S(x)) \\
S[I \wedge \mathsf{let}\, y : \forall \bar{\alpha}[[\,]].\tau \,\mathsf{in}\, C](x) &= S(x) \quad \text{si } \bar{\alpha} \# \mathrm{ftv}(S(x)) \\
S[\mathsf{let}\, y : \sigma \,\mathsf{in}\, [\,]](x) &= S(x) \quad \text{si } x \neq y \\
S[\mathsf{let}\, x : \sigma \,\mathsf{in}\, [\,]](x) &= \sigma
\end{aligned}$$

Si x n'est pas lié par S, alors $S(x)$ n'est pas défini et la règle n'est pas applicable. L'état courant est bloquant : il correspond à un programme où une variable de programme est utilisée sans avoir été définie. Sinon, la règle S-INST peut toujours être appliquée en considérant une α-variante convenable de l'état. S-AND traite le cas où la contrainte à résoudre est une conjonction. La résolution continue — de manière arbitraire — avec le membre gauche, le membre droit étant stocké dans la pile. Lorsqu'une quantification existentielle est rencontrée au sommet de la contrainte externe, la règle S-EX permet de la transférer sur la pile : celle-ci peut être éliminée par S-EX-AND, S-EX-IN et S-EX-LET comme je l'ai expliqué précédemment. Enfin, S-EX-LET s'applique lorsqu'une contrainte let est rencontrée. Dans ce cas, le solveur commence par explorer la contrainte C_1 contenue dans le schéma. L'état courant du solveur primaire, I, est temporairement stocké sur la pile et un nouvel état interne vide est créé. Ainsi, lorsque la résolution de C_1 sera terminée, la contrainte interne obtenue correspondra à la forme résolue de C_1.

Les trois dernières règles s'appliquent quand l'exploration de la contrainte externe est achevée : le solveur examine alors la pile pour déterminer le travail restant à accomplir. Quand le sommet de la pile est un cadre conjonction, S-POP-AND s'applique et la résolution se poursuit avec le membre droit de la conjonction qui avait été stocké par S-AND. S-POP-LET traite le cas d'un cadre de liaison : la contrainte portée par le schéma a été entièrement explorée pour former la contrainte primaire I. Celle-ci est replacée dans le schéma pour former un cadre environnement. L'état du solveur primaire qui avait été empilé par S-LET, I', est restauré et le solveur continue son processus en examinant le corps de la contrainte let, C. Enfin, la règle S-POP-ENV considère un cadre environnement. Puisque le membre droit du let a été résolu, c'est-à-dire mis sous la forme d'une contrainte primaire, il ne peut plus contenir d'occurrence de x. Sa liaison peut donc être supprimé. Le solveur garde cependant une copie de la contrainte portée par le schéma I', pour qu'il soit tenu compte que le schéma doit être instanciable même si la variable x n'a pas été utilisée.

11.3 Correction et terminaison

J'énonce et établis maintenant les propriétés du système de réécriture donné figure 11.1. Tout d'abord, la réduction $\rightarrow$ termine, de telle sorte qu'elle définit un algorithme.

Lemme 11.4 (Terminaison) *Le système de réduction $\rightarrow$ est fortement normalisant.* □

J'omets la preuve de cette propriété, qui est purement technique et ne présente pas d'intérêt particulier. Elle consiste à définir une mesure sur les états diminuée par les règles S-PRED à S-POP-ENV, et préservée par les règles de remontée des quantificateurs existentiels et S-PRIM. On peut alors conclure grâce à l'hypothèse 11.2 (page 217).

Deuxièmement, chaque étape de réécriture préserve la signification de la contrainte que l'état courant représente.

Lemme 11.5 (Correction) *Si $S; I; C \rightarrow S'; I'; C'$ alors $S[I \wedge C] \equiv S'[I' \wedge C']$.* □

⌜ *Preuve.* Par examen de chacune des règles.

- *Cas* S-PRIM. Par l'hypothèse 11.1 (page 217) et le théorème 10.40 (page 198).
- *Cas* S-EX-PRIM, S-EX-AND *et* S-EX. Par LOG-EX-AND.
- *Cas* S-EX-IN. Par LOG-IN-EX.
- *Cas* S-EX-LET. Par LOG-LET-EX.
- *Cas* S-PRED, S-LET *et* S-POP-AND. Par LOG-DUP.
- *Cas* S-INST. Puisque $S(x)$ est de la forme $\forall\bar{\alpha}[I].\tau$, on a $\mathrm{fpv}(S(x)) = \varnothing$. Le résultat s'ensuit par LOG-IN-ID.
- *Cas* S-AND. Les deux contraintes sont identiques.
- *Cas* S-POP-LET. Par LOG-AND et LOG-IN-AND*.
- *Cas* S-POP-ENV. Par LOG-IN*. ⌟

Enfin, je détermine les formes normales du système de réécriture.

Lemme 11.6 (Formes normales) *Une forme normale pour le système de réduction $\rightarrow$ peut s'écrire* ***(i)*** *$S; I; x \preceq \tau$, où $x \notin \mathrm{dpv}(S)$, ou bien* ***(ii)*** *$S; I; \mathsf{true}$ où S ne contient que des cadres existentiels et I est une forme normale pour le solveur primaire.* □

⌜ *Preuve.* Je considère un état $S; I; C$ et suppose qu'il s'agit d'une forme normale pour $\rightarrow$. Je commence par examiner la forme de la contrainte C. Puisque les règles S-PRED, S-AND, S-EX et S-LET ne s'appliquent pas, C est soit true soit une contrainte d'instanciation $x \preceq C$. Dans le deuxième cas, puisque S-INST ne s'applique pas, on en déduit $x \notin \mathrm{dpv}(S)$, de telle sorte que l'état considéré est de la forme (i). J'examine maintenant le cas où la contrainte C est true, *i.e.* où l'état bloqué est de la forme $S; I; \mathsf{true}$. Supposons que la pile S contienne un cadre non existentiel, et considérons le dernier d'entre eux. Si $S = S_1[S_2 \wedge C']$ où S_2 ne comporte que des cadres existentiels alors $S; I; \mathsf{true}$ peut être réduit par S-POP-AND dans le cas où S_2 est vide, et par S-EX-AND sinon. De même, si $S = S_1[I' \wedge \mathsf{let}\, x : \forall\bar{\alpha}[S_2].\tau\ \mathsf{in}\ C']$ (respectivement $S = S_1[\mathsf{let}\, x : \bar{\alpha}\ \mathsf{in}\ I'\tau S_2]$) où S_2 ne comporte que des cadres existentiels alors $S; I; \mathsf{true}$ peut être réduit par S-POP-LET (respectivement S-POP-ENV) dans le cas où S_2 est vide et par S-EX-LET (respectivement S-EX-IN) sinon. Puisque l'état $S; I; \mathsf{true}$ est bloqué, on en déduit que S ne comporte que des cadres existentiels. Enfin, puisque S-PRIM ne s'applique pas, I est une forme normale pour $-\!\!\!\circledcirc\!\!\!\rightarrow$; l'état $S; I; \mathsf{true}$ est donc de la forme (ii). ⌟

Dans le cas (i), la contrainte $S[I \wedge C]$ a une variable de programme libre, de telle sorte qu'elle n'est pas satisfiable. Cela correspond au cas où le programme source contient une variable de programme qui n'est pas liée. Dans les autres cas, la contrainte $S[I]$ est satisfiable si et seulement si I l'est, c'est-à-dire, grâce à l'hypothèse 11.3 (page 217), si le solveur primaire n'a pas signalé une erreur en produisant la contrainte false. Les lemmes 11.4, 11.5 et 11.6 montrent ainsi que $\rightarrow$ donne un algorithme déterminant la satisfiabilité des contraintes.

Comme pour le solveur primaire, la spécification de l'algorithme de résolution donnée par les règles de réduction de la figure 11.1 (page 219) n'est pas déterministe. L'efficacité du processus de résolution est cependant largement dépendante de la stratégie suivie pour appliquer les règles. Pour discuter cette question, mettons un instant de côté les règles S-EX-PRIM à S-EX-LET : comme je l'ai expliqué ci-avant, il est judicieux de les appliquer de manière avide pour ne pas avoir à représenter les cadres existentiels en mémoire, mais cela reste cependant *a priori* orthogonal aux questions d'efficacité. Une stratégie naïve d'application des règles de la figure 11.1 (page 219) consiste à reporter l'utilisation de la règle S-PRIM à la fin de la résolution : les autres règles permettent en effet de réécrire tout état (sans variable de programme libre) en un état de la forme $S; I; \mathsf{true}$ où S ne contient que des cadres existentiels. Cet état peut alors être traité par le solveur primaire. Cette stratégie revient en fait à réécrire la contrainte donnée en entrée en substituant les variables de

programme liée par let, *i.e.* en réécrivant chaque construction $\mathsf{let}\, x : \sigma \,\mathsf{in}\, C$ en $C[x \leftarrow \sigma]$, jusqu'à élimination complète des formes d'introduction et d'élimination de schémas. Abstraction faite de la phase de génération de contraintes, elle revient donc également à substituer syntaxiquement les liaisons let du programme source, puis à typer ce dernier de manière monomorphe. Une telle approche fait en pratique croître la taille du problème à résoudre de manière exponentielle et est très inefficace. Pour déterminer une bonne stratégie d'implémentation du solveur, remarquons que la règle S-INST extrait une contrainte d'un cadre environnement de la pile pour en insérer une copie dans la contrainte primaire courante. On a donc intérêt à ce que cette contrainte soit aussi petite que possible : il faut pour cela la présenter au solveur primaire, qui est supposé pouvoir la simplifier. Cependant, de manière à pouvoir partager ce travail de simplification entre les différents lieux où un schéma est utilisé, il est préférable d'appeler le solveur primaire non pas à chaque instanciation mais lorsque le schéma est placé dans l'environnement (*i.e.* un cadre environnement de la pile), c'est-à-dire juste avant d'appliquer S-POP-LET. Une bonne stratégie d'implémentation consiste donc à faire appel au solveur primaire, par la règle S-PRIM, avant chaque application de S-POP-LET, de manière à former des schémas contenant une contrainte normale pour l'algorithme primaire.

11.4 Une version révisée de l'algorithme

Un état de l'algorithme décrit à la section 11.2 (page 218) peut faire intervenir plusieurs contraintes primaires : outre la deuxième composante de l'état, chaque cadre liaison ou environnement de la pile contient également une contrainte primaire. Chacune d'entre elles correspond à l'état interne d'une « instance » indépendante du solveur primaire. Ces contraintes peuvent avoir des variables libres, qu'elles partagent alors avec les autres structures de données utilisées par l'algorithme : la pile et la contrainte externe, ainsi qu'éventuellement avec les autres contraintes primaires. Toutefois, ce partage empêche apparemment de réaliser une implémentation du solveur secondaire indépendante de l'algorithme primaire utilisé. En effet, ce dernier est susceptible de posséder sa propre représentation des variables de type, laquelle peut être incompatible avec celle adoptée au niveau supérieur. Par exemple, le solveur primaire présenté au chapitre 10 (page 171) identifie la notion de variable de type avec celle de multi-équation. De plus, il est généralement impossible de permettre à plusieurs instances du solveur primaire dont les exécutions s'entrelacent de manipuler la *même* variable de type, c'est à dire le même objet physique en mémoire. Par exemple, dans une implémentation efficace de l'algorithme du chapitre 10 (page 171), une inégalité est représentée par un pointeur bi-directionnel entre les structures de données qui représentent ses deux membres : celles-ci ne peuvent donc pas être partagées de manière simple entre plusieurs instances du solveur. La formalisation du solveur secondaire donnée à la section 11.2 (page 218) n'est donc pas tout à fait satisfaisante.

Ce problème n'est naturellement pas lié à l'utilisation des contraintes d'introduction et d'instanciation de schémas de type dans le processus d'inférence : il apparaît *a priori* dans tout synthétiseur de types à base de contraintes doté de polymorphisme. Il est cependant peu abordé dans la littérature, bien qu'il s'agisse probablement d'une des principales difficultés présentées par l'implémentation d'un tel synthétiseur. C'est pourquoi il me semble utile d'expliquer brièvement comment il peut être résolu d'une manière générale.

Une première solution élégante a été proposée par Trifonov et Smith [TS96], puis reprise par Pottier [Pot01b]. Elle consiste à ne considérer que des schémas de type clos, *i.e.* où toutes les variables de type sont universellement quantifiées. En contrepartie, ceux-ci sont dotés d'un *environnement local* A qui est une fonction partielle des variables de programme vers les types : intuitivement, une expression admet le schéma $\forall\mathcal{V}[C].A \Rightarrow \tau$ si et seulement si, pour toute solution de C, elle a le type τ dans l'environnement A. Il est possible de reformuler de manière équivalente un système tel que $\mathrm{HM}(\mathcal{X})$ (ou bien entendu $\mathrm{MLIF}(\mathcal{X})$) en utilisant de tels schémas, à condition de distinguer les variables de programme liées par let — dites *polymorphes* — auxquelles sont associés des schémas

clos, de celles liées par λ qui, étant *monomorphes*, peuvent être placées dans les environnement locaux A des schémas. Pottier et Rémy [PR03] ont ensuite remarqué que cette formulation alternative se traduisait de manière naturelle dans le langage de contraintes doté des formes d'introduction et d'instanciation de schémas : l'environnement local A correspond à une conjonction de contraintes d'instanciation, *i.e.* $\forall\mathcal{V}[C].A \Rightarrow \tau$ est interprété comme $\forall\mathcal{V}[C \wedge \big(\bigwedge_{x\in\mathrm{dom}(A)} x \preceq A(x)\big)].\tau$. Ils ont également observé que, lors de la génération de contraintes, les schémas de types produits sont de deux formes : une variable x liée par λ donne naissance à une contrainte let « monomorphe », c'est à dire de la forme $\mathsf{let}\, x : \tau \,\mathsf{in}\, \ldots$, tandis qu'une variable de programme liée par let produit une contrainte let dont le schéma ne comporte pas de variable de type libre, mais seulement des variables de programme monomorphes : $\mathsf{let}\, x : \forall\alpha[C].\alpha \,\mathsf{in}\, \ldots$ où $\mathrm{ftv}(C) \subseteq \{\alpha\}$. À partir de cette remarque, on peut contourner le problème soulevé au début de cette section en écrivant l'algorithme de résolution sous une forme où les schémas de type — et par conséquent les contraintes primaires — ne partagent pas de variables de type, mais seulement des variables de programme. Puisque celles-ci n'apparaissent que dans les formes d'instanciation, et pas dans les prédicats traités par le solveur primaire, il est facile de les gérer à l'extérieur de ce dernier.

Je propose ici de manière informelle une nouvelle formulation de cette technique, qui est plus générale car elle permet de traiter des contraintes impliquant des problèmes arbitraires, alors que Pottier et Rémy se restreignent à un sous-ensemble du langage de contraintes correspondant aux problèmes produits par le générateur de contraintes, *i.e.* où les schémas liés par let ont l'une des deux formes indiquées ci-avant. Elle me semble également plus simple, car elle ne nécessite pas de distinguer d'emblée des variables monomorphes ou polymorphes, et plus naturelle : au lieu d'utiliser les variables de programme pour introduire une indirection entre les structures de données du solveur secondaire et celles du solveur primaire, j'associe explicitement à chaque contrainte primaire mentionnée par état du solveur secondaire une ***correspondance*** ϑ qui est une fonction partielle injective et de domaine fini des variables de type (manipulées par le solveur secondaire) vers les variables de type (du solveur primaire). Je note $\vartheta = [\beta_1 \leftarrow \alpha_1, \cdots, \beta_n \leftarrow \alpha_n]$ la correspondance de domaine $\alpha_1 \cdots \alpha_n$ et d'image $\beta_1 \cdots \beta_n$ qui, pour tout $i \in [1, n]$ envoie α_i sur β_i, et ϑ^{-1} sa réciproque. Si ϑ est une correspondance telle que $\alpha \notin \mathrm{dom}(\vartheta)$ et $\vartheta \notin \mathrm{img}(\beta)$, j'écris $\vartheta[\beta \leftarrow \alpha]$ pour la correspondance qui envoie α sur β et coïncide avec ϑ sur le reste de son domaine. Un état du solveur primaire — ou ***état primaire*** — manipulé par le solveur secondaire est maintenant une paire formée d'une contrainte primaire I et d'une correspondance ϑ, notée $\exists\vartheta.I$, où $\mathrm{ftv}(I) \subseteq \mathrm{img}(\vartheta)$. La correspondance ϑ établit une correspondance entre chaque variable de type α telle qu'elles apparaît à l'*extérieur* de l'état, et sa représentation dans la contrainte I, $\vartheta(\alpha)$. Sémantiquement, si $\vartheta = [\beta_1 \leftarrow \alpha_1, \cdots, \beta_n \mapsto \alpha_n]$ alors l'état primaire $\exists\vartheta.I$ est interprété comme la contrainte $I[\beta_1 \leftarrow \alpha_1] \cdots [\beta_n \leftarrow \alpha_n]$ (noté : $I\vartheta$) ou bien, de manière équivalente, $\exists\beta_1 \cdots \beta_n.\big(I \wedge (\bigwedge_{i\in[1,n]} \alpha_i = \beta_i)\big)$. Les notions de variable libre et d'α-conversion d'un état primaire sont définies en conséquence. La syntaxe des piles doit être modifiée en remplaçant chaque occurrence d'une contrainte primaire par un état primaire :

$$S ::= [\,] \mid S[[\,] \wedge C] \mid S[\exists\bar{\alpha}.[\,]] \mid S[\exists\vartheta.I \wedge \mathsf{let}\, x : \forall\bar{\alpha}[[\,]].\tau \,\mathsf{in}\, C] \mid S[\mathsf{let}\, x : \forall\bar{\alpha}[\exists\vartheta.I].\tau \,\mathsf{in}\, [\,]]$$

Les états du solveur secondaire prennent naturellement la forme $S; \exists\vartheta.I; C$. La correspondance doit donner une représentation à l'intérieur de l'état primaire pour chaque variable disponible à l'extérieur, *i.e.* on doit avoir $\mathrm{dtv}(S) \cup \mathrm{ftv}(C) \subseteq \mathrm{dom}(\vartheta)$. Grâce aux correspondances, les variables d'une instance du solveur primaire n'interagissent jamais directement avec celles d'une autre instance ou du solveur secondaire, c'est-à-dire sans passer explicitement par une correspondance. Elle pourraient en fait appartenir à des classes syntaxiques différentes.

La figure 11.2 donne les nouvelles règles de réécriture. Je commente les principales différences par rapport à la formulation initiale de l'algorithme. La règle S'-EX-PRIM introduit de nouveaux noms externes $\vec{\alpha}$ pour des variables $\vec{\beta}$ introduites par le solveur primaire dans la contrainte courante. Ces noms sont naturellement insérés dans la correspondance. Lorsqu'un prédicat est présenté au

$$
\begin{array}{rcll}
S; \exists\vartheta.I; C & \to & S[\exists\bar\alpha.[\,]]; \exists\vartheta.I'; C & \textsc{s'-prim} \\
& & \textit{si } I \multimap_{\Pi\vartheta} I' \textit{ et } \pi \vdash S & \\
S; \exists\vartheta.(\exists\bar\beta.I); C & \to & S[\exists\bar\alpha.[\,]]; \exists\vartheta[\vec\beta \leftarrow \vec\alpha].I; C & \textsc{s'-ex-prim} \\
& & \textit{si } \bar\alpha \# \mathrm{ftv}(C), \textit{ } \vec\alpha \textit{ distinctes et } \vec\beta \textit{ distinctes} & \\
S[(\exists\bar\alpha.S') \wedge D]; \exists\vartheta.I; C & \to & S[\exists\bar\alpha.(S' \wedge D)]; \exists\vartheta.I; C & \textsc{s'-ex-and} \\
& & \textit{si } \bar\alpha \# \mathrm{ftv}(D) & \\
S[\mathsf{let}\, x : \sigma \,\mathsf{in}\, \exists\bar\alpha.S']; \exists\vartheta.I; C & \to & S[\exists\bar\alpha.\, \mathsf{let}\, x : \sigma \,\mathsf{in}\, S']; \exists\vartheta.I; C & \textsc{s'-ex-in} \\
& & \textit{si } \bar\alpha \# \mathrm{ftv}(\sigma) & \\
S[\exists\vartheta'.I' \wedge \mathsf{let}\, x : \forall\bar\alpha[\exists\bar\beta.S'].\tau \,\mathsf{in}\, D]; & & & \\
\exists\vartheta.I; C & \to & S[\exists\vartheta'.I' \wedge \mathsf{let}\, x : \forall\bar\alpha\bar\beta[S'].\tau \,\mathsf{in}\, D]; \exists\vartheta.I; C & \textsc{s'-ex-let} \\
& & \textit{si } \bar\beta \# \mathrm{ftv}(\tau) & \\
S; \exists\vartheta.I; p\ \vec\tau & \to & S; I \wedge p\ \vec\tau\vartheta; \mathsf{true} & \textsc{s'-pred} \\
& & \textit{si } p \neq \mathsf{true} & \\
S; \exists\vartheta.I; x \preceq \tau & \to & S[\exists\bar\alpha.[\,]]; & \textsc{s'-inst} \\
& & \exists\vartheta_1.\big(I \wedge I_2\vartheta_2^{-1}\vartheta_1 \wedge (\tau_2 \leq \tau)\vartheta_1\big); \mathsf{true} & \\
& & \vartheta_1 = \vartheta[\vec\beta \leftarrow \vec\alpha] \textit{ et } S(x) = \forall\bar\alpha[\exists\vartheta_2.I_2].\tau_2 & \\
& & \vec\alpha \textit{ distinctes et } \vec\beta \textit{ distinctes} & \\
S; \exists\vartheta.I; C_1 \wedge C_2 & \to & S[[\,] \wedge C_2]; \exists\vartheta.I; C_1 & \textsc{s'-and} \\
S; \exists\vartheta.I; \exists\bar\alpha.C & \to & S[\exists\bar\alpha.[\,]]; \exists\vartheta[\vec\beta \leftarrow \vec\alpha].I; C & \textsc{s'-ex} \\
& & \textit{si } \bar\alpha \# \mathrm{ftv}(\exists\vartheta.I), \textit{ } \vec\alpha \textit{ distinctes et } \vec\beta \textit{ distinctes} & \\
S; \exists\vartheta.I; \mathsf{let}\, x : \forall\bar\alpha[C_1].\tau \,\mathsf{in}\, C_2 & \to & S[\exists\vartheta.I \wedge \mathsf{let}\, x : \forall\bar\alpha[[\,]].\tau \,\mathsf{in}\, C_2]; \exists\vartheta'.\mathsf{true}; C_1 & \textsc{s'-let} \\
& & \textit{si } \mathrm{dom}(\vartheta') = \mathrm{dom}(\vartheta) \uplus \bar\alpha & \\
S[[\,] \wedge C]; \exists\vartheta.I; \mathsf{true} & \to & S; \exists\vartheta.I; C & \textsc{s'-pop-and} \\
S[\exists\vartheta'.I' \wedge \mathsf{let}\, x : \forall\bar\alpha[[\,]].\tau \,\mathsf{in}\, C]; & & & \\
\exists\vartheta.I; \mathsf{true} & \to & S[\mathsf{let}\, x : \forall\bar\alpha[\exists\vartheta.I].\tau \,\mathsf{in}\, [\,]]; \exists\vartheta'.I'; C & \textsc{s'-pop-let} \\
S[\mathsf{let}\, x : \forall\bar\alpha[\exists\vartheta_2.I_2].\tau \,\mathsf{in}\, [\,]]; & & & \\
\exists\vartheta.I; \mathsf{true} & \to & S[\exists\bar\alpha.[\,]]; \exists\vartheta_1.(I \wedge I'\vartheta_2^{-1}\vartheta_1); \mathsf{true} & \textsc{s'-pop-env} \\
& & \textit{si } \vartheta_1 = \vartheta[\vec\beta \leftarrow \vec\alpha], \textit{ } \vec\alpha \textit{ distinctes et } \vec\beta \textit{ distinctes} &
\end{array}
$$

Figure 11.2 – Algorithme de résolution révisé

solveur primaire par S'-PRED, les variables de type que contiennent les types $\vec{\tau}$ doit être traduites en leur appliquant la correspondance courante. La règle S'-INST réalise l'instanciation d'un schéma en explicitant le processus de copie puis de renommage des variables : tout d'abord, de nouvelles variables $\vec{\beta}$ doivent être générées dans le solveur primaire pour représenter les variables locales du schéma $\vec{\alpha}$: cela nécessite d'étendre la correspondance ϑ en ϑ_1. On peut ensuite effectuer une copie de la contrainte I_2 du schéma, en remplaçant chaque variable par sa représentation locale, *i.e.* en lui appliquant la substitution $\vartheta_2^{-1}\vartheta_1$. Enfin, les types τ_2 et τ sont représentés en utilisant les formes externes des variables de type, et doivent donc être convertis par ϑ_1 avant d'être présentés au solveur primaire. S'-LET crée des représentants à l'intérieur du solveur primaire pour les variables $\bar{\alpha}$ extraites de la quantification existentielle. Comme S-LET, la règle S'-LET stocke l'état primaire courant sur la pile, et crée une nouvelle instance du solveur primaire. Celle-ci doit recevoir une nouvelle correspondance ϑ', donnant une représentation fraîche à chaque variable de type définie dans la pile ou libre dans C. Dans le cas d'un schéma monomorphe, *i.e.* si $\bar{\alpha} = \varnothing$ et $C_1 = \mathsf{true}$, une optimisation immédiate consiste à ne pas créer cette correspondance, puisqu'elle n'est pas utilisée. Cela correspond au traitement particulier des variables monomorphes de Pottier et Rémy. Enfin S'-POP-ENV procède comme S'-INST, pour insérer la contrainte du schéma éliminé dans la contrainte primaire courante.

Cette nouvelle formulation montre clairement les fonctionnalités que le solveur primaire doit offrir au solveur secondaire. Tout d'abord, puisque chaque état du solveur secondaire peut contenir plusieurs états primaires, le solveur primaire doit être capable de gérer simultanément un ensemble d'états indépendants. Sur chacun de ces états, il doit fournir quatre opérations. La première, qui correspond à S'-PRIM, consiste à mettre un état en forme normale. Lors de cet appel, le solveur primaire peut demander à la strate secondaire la création de nouvelles variables quantifiées existentiellement, grâce à S'-EX-PRIM. En d'autres termes, le solveur primaire n'a pas besoin de conserver de quantificateurs existentiels dans sa représentation des contraintes. La deuxième opération, utilisée par S'-PRED doit permettre d'introduire chaque prédicat disponible dans le langage au sein d'un état existant. Pour réaliser S'-INST et S'-POP-ENV, le solveur primaire doit pouvoir effectuer une copie d'un de ses états, puis l'insérer, en substituant ses variables, dans un autre état. Enfin, pour implémenter S'-LET et S'-EX, le solveur primaire doit permettre la création de nouvelles variables de type fraîches dans un état.

CHAPITRE DOUZE

12 Discussion

12.1 Implémentation et résultats expérimentaux

L'algorithme de résolution de contraintes décrit dans cette partie de la thèse a été implémenté en Objective Caml sous la forme d'une librairie, appelée *Dalton*, dont le code source est disponible électroniquement [Sim02a]. Cette librairie est légèrement moins modulaire que la présentation que j'ai donnée ici, puisque les deux strates n'y sont pas aussi clairement indépendantes : j'ai réalisé la possibilité de cette présentation postérieurement à ce travail d'implémentation. La librairie reste cependant indépendante du langage des types manipulés : elle est donnée sous la forme d'un foncteur Objective Caml, dont les paramètres sont des modules décrivant les ensembles d'atomes, de constructeurs de types et d'étiquettes de rangées. J'espère ainsi qu'elle pourra être utilisée comme moteur d'inférence de types pour différents compilateurs ou analyseurs statiques.

Pour évaluer les performances de cette librairie, j'ai également réalisé une implémentation du compilateur Caml Light qui est modulaire vis-à-vis du système de types et du solveur de contraintes utilisé pour l'inférence de types. J'en ai ensuite considéré deux instances. La première est munie d'un solveur standard de contraintes d'unification, tel que celui décrit par Pottier et Rémy [PR03], et implémente donc sensiblement le même système de types que Caml Light lui-même. La seconde utilise la librairie *Dalton* pour mettre en œuvre une extension du système de types précédent avec du sous-typage structurel, où chaque constructeur de type (et donc chaque nœud d'un type) porte une annotation atomique covariante appartenant à un treillis quelconque. Ce deuxième système de type n'a pas d'intérêt pour lui-même, mais il est représentatif, pour ce qui concerne la résolution de contraintes, d'un système effectuant une analyse de flots de données ou d'information. Il donne ainsi une mesure pertinente de l'efficacité du solveur.

J'ai ensuite utilisé ces deux maquettes pour typer divers programmes ou librairies en Caml Light, écrits par différents auteurs. Les résultats étant relativement constants, j'ai choisi de reproduire, dans les deux premières colonnes de la figure 12.1, les données relatives aux deux plus importants d'entre eux (pour la taille du code) : le compilateur Caml-Light lui-même et sa librairie standard.

De manière à comparer mon moteur d'inférence pour le sous-typage structurel avec l'unification habituelle, j'ai tout d'abord mesuré le temps d'exécution de la seule phase de typage du compilateur : ils sont reproduits dans la moitié supérieure de la figure 12.1. Les mesures ont été effectuées sur un micro-ordinateur doté d'un micro-processeur Intel Pentium III à une fréquence de 1 GHz. Sur ces tests, la librairie *Dalton* s'est montrée seulement deux à trois fois plus coûteuse que

	Caml Light librairie	Caml Light compilateur	Flow Caml librairie
Taille du code (nœuds syntaxiques)	14002	22996	13123
1. Coût de l'inférence de types			
Unification	0.346 s	0.954 s	
Sous-typage structurel (*Dalton*)	0.966 s	2.213 s	n.a.
rapport	2.79	2.31	
2. Mesures internes			
Multi-équations	30345	65946	73328
Unification des cycles	501 (2%)	1381 (2%)	1764 (2%)
Réduction des chaînes	9135 (30%)	15967 (24%)	17239 (24%)
Élimination des variables apolaires	15288 (50%)	31215 (47%)	18460 (25%)
Minimisation	424 (1%)	644 (1%)	815 (1%)
Variables expansées	948 (3%)	1424 (2%)	1840 (3%)

Figure 12.1 – Mesures expérimentales

l'unificateur. De telles mesures doivent naturellement être interprétées avec précaution ; cependant, l'inférence dans les systèmes à base d'unification est reconnue comme étant efficace et est largement utilisée. Il me semble donc qu'elles tendent à montrer que le sous-typage structurel est utilisable *en pratique*.

Outre ces observations « externes », j'ai également modifié le code source de la librairie *Dalton* de manière à dénombrer certaines des opérations effectuées par le solveur lors de son exécution. J'ai tout d'abord calculé le nombre total de multi-équations générées par le processus de typage, à la fois par la génération des contraintes initiales et par la résolution, puis ensuite le nombre d'entre-elles éliminées par chacune des techniques de simplification appliquées. Enfin, j'ai comptabilisé les multi-équations qui ont dû être expansées. Ces mesures sont reproduites dans la partie inférieure de la figure 12.1. Pour chaque technique de simplification, je donne le nombre de multi-équations qu'elle a permis d'éliminer, puis, entre parenthèses, le pourcentage du nombre total de multi-équations que cela représente.

La réduction des chaînes apparaît clairement être une simplification cruciale : en effet, elle élimine de l'ordre d'un quart des multi-équations, qui sont des variables, et ce *avant* leur expansion. La contribution directe de l'unification des cycles est dix fois moins importante ; j'ai cependant observé que son retrait affectait notablement l'efficacité de la réduction des chaînes. Grâce à ces deux heuristiques, la « contribution » du processus d'expansion devient relativement marginale, comment l'indique la dernière ligne de la figure 12.1 : le nombre de variables expansées représente seulement quelques pourcents du nombre total de multi-équations, et de l'ordre d'un dixième de celles qui ne sont pas terminales. Notons également que le fait d'entrelacer l'application des heuristiques de simplification avec la résolution a un impact très fort sur l'efficacité : en modifiant l'implémentation du solveur de manière à retarder l'élimination des chaînes au terme de l'expansion, j'ai vu le nombre de variables expansées être multiplié par un facteur 20. Les mesures montrent également que la contribution de l'élimination des variables apolaires est comparable à celle de l'élimination des chaînes. D'un point de vue quantitatif, l'impact de la minimisation est peu important. Cependant, la pratique m'a montré qu'elle était importante pour la lisibilité des schémas de type obtenus.

Enfin, j'ai naturellement utilisé la librairie *Dalton* comme moteur d'inférence pour le système Flow Caml [Sim]. J'ai effectué les mêmes mesures que précédemment sur le typage de sa librairie standard. Elles sont reproduites dans la dernière colonne de la figure 12.1 et confirment les précédentes.

12.2 Comparaison de schémas

Lorsque l'on s'intéresse à un langage de programmation doté d'un système de modules à la ML, comme Flow Caml, ou même de simples unités de compilation munies d'interface, il est nécessaire de disposer d'un algorithme permettant de comparer des schémas de type, c'est-à-dire de vérifier que le schéma de type inféré à partir d'un fragment de code est au moins aussi général que celui donné par le programmeur dans l'interface correspondante. Dans le cas de ML, la comparaison des schémas peut être réalisée de manière purement syntaxique : un schéma σ_1 est plus général que σ_2 si et seulement si il existe une substitution des variables de type vers les types φ telle que $\sigma_2 = \varphi(\sigma_1)$. Dans un système de types à base de contraintes, sa définition nécessite de prendre en compte les contraintes portées par les schémas, en considérant par exemple leurs instances.

Définition 12.1 (Comparaison de schémas) *Soient σ_1 et σ_2 deux schémas de type. σ_1 est* **plus général** *que σ_2 (noté : $\sigma_1 \preccurlyeq \sigma_2$) si et seulement, pour tous φ, X et t, si $\varphi, X \vdash \sigma_2 \preceq t$ alors $\varphi, X \vdash \sigma_1 \preceq t$.* □

En d'autres termes, σ_1 est plus général que σ_2 si et seulement si toute instance du second est également instance du premier. Cette définition peut être expliquée grâce la propriété suivante :

$$\text{pour toute contrainte } C, \text{ si } \quad \sigma_1 \preccurlyeq \sigma_2 \quad \text{ alors } \quad \mathsf{let}\, x : \sigma_2 \,\mathsf{in}\, C \Vdash \mathsf{let}\, x : \sigma_1 \,\mathsf{in}\, C$$

En effet, elle montre que, si un fragment de programme a reçu un schéma de type σ_1 alors il peut être utilisé avec un autre schéma moins général σ_2.

Le problème de la comparaison de schémas de type se ramène naturellement à celui de l'implication de contraintes : sous l'hypothèse de fraîcheur $\mathrm{ftv}(\tau_2) \# \bar{\alpha}_1$, $\forall \bar{\alpha}_1[C_1].\tau_1 \preccurlyeq \forall \bar{\alpha}_2[C_2].\tau_2$ est équivalent à $C_2 \Vdash \exists \bar{\alpha}_1.(C_1 \wedge \tau_1 \leq \tau_2)$. On peut faire, pour continuer, deux remarques. Tout d'abord, les définitions d'un programme qui mentionnent des identificateurs non définis dans l'environnement sont rejetées par le typage, et les schémas de type fournis dans les interfaces ne sont généralement pas autorisés à mentionner des variables de type libres. Ainsi, grâce à LOG-LET-DUP, on peut se restreindre à la comparaison de schémas sans variable de programme libre. De plus, l'algorithme de résolution des chapitres 10 et 11 permet de réécrire toute contrainte sans variable de programme libre sous la forme d'une contrainte primaire. On peut ainsi ramener la comparaison de schémas de type au test d'implication entre contraintes primaires.

Si cette question reste ouverte pour les formes non-structurelles de sous-typage [Pot98, NP01, NP03], le problème de l'implication de contraintes est connu pour être décidable dans le cas du sous-typage structurel [KR03]. L'algorithme de résolution que j'ai décrit au chapitre 10 (page 171) peut facilement être adapté pour obtenir une procédure de décision efficace. Par manque de place, je n'en donne pas ici une description précise, me contentant d'expliquer ses grandes lignes, en faisant de plus abstraction des rangées, qui introduisent quelques difficultés techniques. Le lecteur intéressé pourra consulter [Sim03c] où cette procédure est précisément décrite puis prouvée. Elle prend en entrée deux contraintes primaires I_1 et I_2 dans la forme produite par l'algorithme de résolution (dans le cas où au moins l'une des contraintes I_1 et I_2 n'est pas satisfiable, le problème $I_1 \Vdash I_2$ est immédiat). On peut supposer, sans perte de généralité, que la fonction I_1 est sans quantification existentielle (puisque le problème $\exists \bar{\alpha}.I_1 \Vdash I_2$ est équivalent à $I_1 \Vdash I_2$, sous la condition $\bar{\alpha} \# I_2$). La procédure commence par construire la conjonction $I_1 \wedge I_2$ pour l'unifier. Si cette phase d'unification échoue, cela signifie que I_1 et I_2 posent des contraintes incompatibles sur la forme d'un type, de telle sorte que $I_1 \not\Vdash I_2$. On obtient sinon une contrainte unifiée I_u. De même, si une variable est terminale dans I_1 mais pas dans I_u, alors I_2 pose une contrainte sur la forme d'un type qui n'est pas présente dans I_1, de telle sorte que $I_1 \not\Vdash I_2$. Sinon, la comparaison se poursuit en expansant et décomposant la contrainte I_u, jusqu'à obtenir un problème réduit I_r. Il suffit alors de comparer la clôture transitive du graphe formé par les inégalités et gardes entre variables de types libres dans ces deux contraintes : on peut montrer, à quelques détails près, que I_1 implique I_2 si et seulement si tout chemin entre deux variables libres de I_r est également présent dans I_1.

Avant de refermer cette parenthèse, j'aimerais remarquer que cette procédure reste en fait syntaxique : à peu de choses près, elle consiste à réécrire les deux contraintes sous une forme particulière, qui permet ensuite de comparer directement les prédicats qu'elles font intervenir. Elle est implémentée dans la librairie *Dalton* et utilisée par le système Flow Caml pour vérifier que les structures satisfont leurs interfaces.

12.3 Quelques améliorations possibles

Je termine cette discussion en présentant trois extensions ou améliorations de l'algorithme de résolution de contraintes qui me semblent intéressantes.

Tout d'abord, l'une des limitations les plus importantes de cet algorithme réside, me semble-t-il, dans le fait qu'il ne permet pas de traiter des types récursifs. Cette impossibilité m'a par exemple amené à exclure les objets d'Objective Caml du langage Flow Caml, car leur typage nécessite des types récursifs — au contraire des structures de données traditionnelles dont la récursivité est cachée grâce aux déclarations de types. L'inférence de types en présence de sous-typage structurel et de types récursifs a été peu étudiée : le seul algorithme publié est du à Tiuryn et Wand, mais sa complexité est exponentielle [TW93]. Dans le cas présent, la difficulté apportée par les types récursifs est étroitement liée à la stratégie de résolution qui consiste à retarder la clôture autant que possible en réalisant une expansion de la structure des types. La procédure d'expansion décrite à la section 10.2.3 (page 184) ne termine pas en présence de types récursifs : par exemple, si c est un constructeur de type unaire covariant, la contrainte $\langle c\,\alpha\rangle^\bullet \approx \langle\alpha\rangle^\bullet$ donne successivement par ce processus $\exists\alpha_1.\big(\langle c\,\alpha\rangle^\bullet \approx \langle\alpha = c\,\alpha_1\rangle^\circ \approx \langle\alpha_1\rangle^\bullet\big)$ puis $\exists\alpha_1\alpha_2.\big(\langle c\,\alpha\rangle^\bullet \approx \langle\alpha = c\,\alpha_1\rangle^\circ \approx \langle\alpha_1 = c\,\alpha_2\rangle^\circ \approx \langle\alpha_2\rangle^\bullet\big)$, etc. De plus, la structure des types décrite par une contrainte de squelette est régulière (au sens d'un arbre régulier), mais les atomes portés par cette structure n'ont pas *nécessairement* une disposition régulière. Par exemple, si c est un constructeur de type de signature $\mathsf{Type}^+ \cdot \mathsf{Atom}^+$, alors les solutions de la contrainte $\exists\beta.(\alpha \approx c\,\alpha\,\beta)$ envoient α sur un type brut de la forme $c\,(c\,(c\,(c\cdots\ell_n)\cdots\ell_3)\,\ell_2)\,\ell_1$, où les atomes ℓ_i sont arbitraires. Ainsi, même si on restreint le modèle à des types réguliers (à la fois pour la structure et les décorations atomiques), il reste impossible de mettre cette contrainte sous une forme expansée, puisque la période des annotations ℓ_i reste libre. C'est la raison pour laquelle il me semble difficile d'adapter de manière directe le processus d'expansion et donc l'algorithme présenté dans cette thèse, à un système doté de type récursifs. Pour un tel système, il faudrait probablement considérer des méthodes basées sur une clôture transitive du graphe des inégalités, proches de celles de Pottier [Pot98, Pot01b]. Toutefois, elles mèneraient probablement à un solveur beaucoup moins efficace, et produisant des schémas de type plus difficiles à interpréter par l'utilisateur. Pour améliorer cela, on pourrait envisager un algorithme hybride, effectuant l'expansion lorsque cela est possible, et une clôture transitive en présence de structures cycliques dans les types.

Pour permettre au solveur du chapitre 10 (page 171) de traiter le prédicat $\blacktriangleleft$ introduit par l'extension du système MLIF($\mathcal{X}$) décrite à la section 8.2 (page 154), il est nécessaire d'augmenter légèrement l'expressivité de la garde $\lessdot$, de manière à pouvoir coder $t \blacktriangleleft \ell$ par $t \lessdot \ell$. Il faut pour cela permettre la décomposition du prédicat $\lessdot$ sur des paramètres invariants des constructeurs de type, *i.e.* autoriser les ensembles guarded-l(c) à contenir des positions i telles que $c.i = \pm$. En effet, $\blacktriangleleft$ se décompose, par exemple, sur les deux paramètres du constructeur ref, dont le premier est invariant. Cette possibilité ne préserve pas la propriété 10.4 (page 174) établie ci-avant et nécessite donc d'adapter la clôture polarisée qui repose sur elle. Considérons par exemple la contrainte $\exists\alpha_2.(\alpha_1 \approx \alpha_2 \wedge \alpha_2 \blacktriangleleft \alpha_3)$. La procédure de clôture décrite à la section 10.3.3 (page 209) simplifie cette contrainte en true, quelles que soient les polarités de α_1 et α_3 : intuitivement, cette simplification est justifiée par le fait que, pour toute affectation du type α_1, on peut choisir l'élément minimal de son squelette brut pour α_2 ce qui permet de satisfaire $\alpha_2 \lessdot \alpha_3$. Ce raisonnement n'est plus correct si $\lessdot$ peut se décomposer sur des paramètres invariants : par exemple, dans un contexte contenant le prédicat $\alpha_1 = \mathsf{ref}(\mathsf{int}\,\beta)\,\gamma$, la contrainte $\exists\alpha_2.(\alpha_1 \approx \alpha_2 \wedge \alpha_2 \blacktriangleleft \alpha_3)$ implique $\beta \leq \alpha_3$, de telle

sorte qu'elle doit *a priori* être conservée. Je n'ai pas inclus cette extension dans le développement du chapitre 10 (page 171), pour ne pas l'augmenter de détails techniques peu intéressants, elle est cependant traitée dans mon implémentation du système, la librairie *Dalton* [Sim02a].

Je n'ai pas parlé jusqu'à présent des rapports d'erreur à présenter à l'utilisateur lorsque le typage échoue, c'est-à-dire quand le solveur découvre que la contrainte courante n'est pas satisfiable. Il s'agit d'un problème important dans le développement d'un compilateur, qui est à l'origine de nombreux travaux dans le cas de ML, mais a peu été étudié dans les systèmes à base de sous-typage. L'algorithme de résolution présenté aux chapitres 10 et 11 peut déclencher trois sortes d'erreurs. La première est produite par la strate secondaire, lorsque la variable de programme d'une contrainte d'instanciation n'est pas liée dans l'environnement. Comme je l'ai expliqué, cela correspond au cas où l'utilisateur utilise dans son programme une variable sans la définir, et peut donc facilement être signalé. Bien souvent, ces erreurs sont même détectées par une des passes du compilateur préalables au typage. Les deux autres sortes d'erreurs sont issues du solveur primaire, et correspondent à des cas où le programme étudié est réellement mal typé. Tout d'abord l'algorithme d'unification échoue s'il doit unifier deux multi-squelettes comportant des constructeurs de types incompatibles (règle SPU-CLASH). Ce type d'erreurs est similaire à celui produit dans les systèmes à base d'unification seule, et peut donc être signalé en utilisant les mêmes techniques. Par exemple, Flow Caml signale une telle erreur d'une manière comparable à Objective Caml : il donne la structure du type trouvée par le synthétiseur pour une sous-expression du code, puis celle du type attendu par son contexte et qui est incompatible. Les autres erreurs engendrées par le solveur primaire apparaissent au niveau atomique lors de la vérification des chemins entre constantes atomiques (section 10.2.4, page 192). La production de bons rapports pour ces erreurs est plus difficile. Flow Caml se contente pour l'instant d'afficher les constantes atomiques reliées par un chemin qui n'est pas satisfiable, et d'indiquer une sous-expression dans laquelle il a été trouvé. Cela semble raisonnable en pratique, mais pourrait être sensiblement amélioré. On pourrait en effet, lors de la génération de contraintes, attacher à chaque inégalité ou garde une étiquette indiquant le point du programme où elle a été engendrée ; puis essayer, lors de l'unification, de l'expansion et de la décomposition, de propager ces informations. Ainsi, lorsqu'un chemin invalide est découvert dans le graphe atomique, on serait en mesure d'indiquer à l'utilisateur une suite de points du programme, si possible minimale, dont elle est issue. Cela correspondrait en fait, dans le cas de Flow Caml, à la trace d'un flot d'information illégal. Une approche raisonnable consisterait à avoir deux solveurs différents, éventuellement dotés de stratégies ou de structures de données différentes : l'un, tel que celui décrit ci-avant, adapté pour l'efficacité de la résolution, et l'autre à la production de messages d'erreur précis. Ces deux objectifs sont en effet relativement contradictoires, puisque les techniques de simplification éliminent une partie de l'information initialement présente dans les contraintes, qui serait justement utile aux rapports d'erreurs. Les programmes corrects seraient acceptés rapidement grâce au premier solveur. En cas d'erreur, le second pourrait ensuite être lancé pour obtenir un bon rapport.

Conclusion

J'espère avoir donné, dans les parties précédentes, une description claire et intéressante des principales contributions de cette thèse. Naturellement, l'ordre de présentation que j'ai adopté ne respecte pas le chemin parcouru lors de mon travail de thèse. C'est pourquoi, il me semble intéressant de le retracer brièvement, vu de son extrémité, avant de conclure.

Le point de départ de mes travaux a été l'article de Pottier et Conchon [PC00] présenté à la conférence ICFP en septembre 2000 : j'ai commencé par essayer d'étendre leur travail à un langage doté de références. Il est assez rapidement apparu que leur approche s'adaptait difficilement au cas d'un langage doté d'effets de bord, d'une part à cause de l'utilisation d'une caractérisation *dynamique* des flots d'information sur une seule trace d'exécution, et d'autre part en raison de l'idée de se baser sur un système de type arbitraire, qui nécessitait l'énoncé d'hypothèses complexes pour être conservée. De la première difficulté est née l'idée des crochets $\langle \cdot \mid \cdot \rangle$ et de la sémantique associée. Le deuxième obstacle m'a amené à considérer le seul cas du système de type Hindley–Milner (le seul intéressant *in fine*, la généricité de Pottier et Conchon étant motivée essentiellement par son caractère modulaire). Je me suis ensuite attaché aux exceptions. L'objectif était relativement clair — un système comparable à celui de Myers — mais la méthode à utiliser pour l'atteindre l'était moins. La première voie que j'ai suivie a été de coder les exceptions sous forme de sommes, et de raffiner le typage de ces dernières : cela semblait à la fois simple et élégant. J'ai formulé plusieurs tentatives, mais toutes se sont soldées par des échecs : il était difficile de voir *comment* améliorer le typage des sommes pour obtenir celui attendu pour les exceptions. Je me suis donc résigné à une approche directe, qui s'est en fait révélée très intéressante par sa simplicité, le calcul à crochets développé pour les références se montrant tout à fait adapté.

Un premier prototype — relativement inefficace — a permis de valider la conception du système. Il a montré que le traitement des exceptions de première classe était trop complexe, et qu'une restriction était souhaitable. Aussi surprenant que cela puisse paraître, ce n'est qu'à ce moment qu'a été effectué le choix d'un sous-typage *structurel* : celui-ci avait été laissé ouvert dans les étapes précédentes. Deux arguments ont prévalu : il laissait espérer un algorithme d'inférence de bonne efficacité, et permettait seul de garantir que tout programme Caml bien typé dans le système annoté l'était également au sens habituel. Parallèlement à ces travaux expérimentaux, j'ai continué ma réflexion sur les sommes, pour donner un typage fin de ces constructions puis des exceptions dans le cas purement fonctionnel [Sim02b]. Ce travail n'a pas été intégré dans la branche principale, pour les raisons que j'ai déjà évoquées, mais a été essentiel à une bonne compréhension du système direct conçu quelques mois auparavant.

La deuxième année de mon travail a été consacrée à l'implantation du système. J'ai commencé par réaliser un solveur efficace pour les contraintes de sous-typage structurel et les gardes, de manière à pouvoir réaliser l'inférence de types. Implantation et formalisation ont été menées de front, la première ayant cependant souvent une longueur d'avance, car elle était — c'est peut-être surprenant — plus simple pour moi. La formalisation de l'algorithme a permis de mieux le comprendre, et d'améliorer la structure du code de l'implantation. La preuve a exhibé à plusieurs

reprises des erreurs, parfois subtiles. Son utilité a été d'autant plus forte que les *bugs* mis en évidence par des tests étaient souvent difficiles à corriger : de par la complexité des structures de données mises en œuvre, il était peu aisé de « lire » et comprendre l'état interne du solveur au cours de son exécution. Une fois ce solveur devenu utilisable, je me suis attaqué à la réalisation du système Flow Caml. Le code proprement dit n'était pas difficile à écrire — la principale difficulté ayant été isolée dans le solveur de contraintes. Cette séparation a d'ailleurs été, me semble-t-il, essentielle à la réussite de ce travail. Plusieurs difficultés nouvelles relatives à l'analyse de flots d'information, qui n'étaient pas apparues avec l'étude du langage noyau, ont dû être résolues : déclarations de types, règles de séquencement des phrases *toplevel*, définition du treillis des niveaux d'information, etc. Une première version de Flow Caml a été publiée fin 2002, suivie quelques mois plus tard d'une documentation détaillée.

Contributions

Du travail présenté dans ce mémoire, je crois que l'on peut retenir deux principales contributions. Le première concerne l'inférence de types en présence de sous-typage structurel. L'algorithme de résolution de contraintes que j'ai proposé reprend plusieurs idées issues du *folklore* ou de travaux précédents (certains relatifs à d'autres formes de contraintes), pour les combiner et obtenir un ensemble cohérent et bien compris. Il s'agit, à ma connaissance, de la première description complète et prouvée d'un tel solveur. Les résultats expérimentaux que j'ai obtenus montrent qu'il est efficace. Enfin, malgré l'abondant travail réalisé ces dix dernières années dans ce domaine, Flow Caml est — à ma connaissance — l'une des premières réalisations d'un langage de programmation « grandeur réelle » doté à la fois de polymorphisme, de sous-typage et d'un algorithme d'inférence de types.

La deuxième contribution de cette thèse, qui formait son objectif initial, est d'avoir réduit l'écart entre théorie et pratique dans l'univers des analyses de flots d'information. Celle présentée et prouvée dans cette thèse améliore sensiblement les travaux précédents, aussi bien au niveau de l'expressivité du langage considéré, que de la puissance du système de types utilisé. Toutefois, sa preuve de correction est plus simple que beaucoup de celles que l'on trouve dans la littérature. La technique que j'ai développée a d'ailleurs été réutilisée par la suite par différents chercheurs [Pot02, Zda03]. Enfin, je pense que la réalisation du système Flow Caml est une avancée certaine dans le domaine de l'analyse de flots d'information : elle permet d'éprouver ses possibilités et d'envisager l'écriture de programmes réels. J'ai par exemple développé d'une bibliothèque permettant d'écrire et de vérifier des scripts CGI [The] à l'aide de Flow Caml.

Une comparaison avec Jif

À ma connaissance, un seul autre analyseur de flots d'information qui traite un langage de programmation réaliste a été publié : il s'agit de *Jif* développé à l'Université de Cornell par Myers et son équipe [MNZZ01] pour le langage Java. J'ai déjà comparé ce système à plusieurs reprises avec mon travail dans ce mémoire, je rappelle les principaux points. Il faut tout d'abord mentionner que le travail de Myers ne donne aucune preuve formelle de sa correction ou d'énoncé formel de la propriété garantie par le système. En ce qui concerne les systèmes de types, celui de Jif ne possède qu'une forme restreinte de polymorphisme, restreint aux méthodes statiques. Un moyen quelque peu détourné permet cependant d'écrire des méthodes virtuelles génériques pour les niveaux de leurs arguments, qui deviennent autant d'arguments supplémentaires passés dynamiquement. De plus, Jif n'effectue qu'une inférence *locale* des types et des niveaux, dans le même esprit que Java. Le programmeur doit donc annoter de son code, en particulier au niveau des en-têtes de méthodes. Plusieurs mécanismes d'annotations « par défaut » astucieux allègent toutefois cette tâche. Inversement, deux fonctionnalités intéressantes de Jif ne sont pas présentes dans Flow Caml. Tout d'abord, les niveaux d'information mis en œuvre par Jif ont une structure plus élaborée que

ceux de Flow Caml, et sont utilisés pour offrir une forme de déclassification des données, contrôlée par les principaux. La propriété alors vérifiée par le système n'est cependant pas très claire. Enfin, dans le langage Jif, les niveaux d'information (*labels* suivant la terminologie de Myers) peuvent être manipulés comme des objets Java habituels : cela permet d'effectuer des tests dynamiques lorsque les niveaux ne peuvent pas être déterminés statiquement de manière suffisamment précise. L'analyse statique est censée garantir la correction du programme quel que soit le résultat des tests à l'exécution. Elle peut de plus exploiter leur présence pour raffiner le typage dans certaines branches du programme.

Depuis quelques mois, je me suis intéressé à l'introduction de cette possibilité dans le langage Flow Caml. Il m'est rapidement apparu que la preuve de correction (c'est-à-dire de non-interférence) d'une telle extension ne posait pas de problème particulier : les crochets semblent, encore une fois, tout à fait adaptés. Le point difficile semble être l'inférence de types, que je souhaite conserver. C'est à ce problème que je me suis intéressé dans mes derniers travaux : avec François Pottier, nous avons remarqué que le mécanisme de niveaux dynamiques peut être réalisé grâce à une variante des *types de données gardés* proposés par Xi, Chen et Chen [XCC03]. Nous nous sommes donc intéressés à une formulation de ce système dans le style de HM($\mathcal{X}$), et avons montré comment l'inférence de types se ramenait à la résolution de contraintes, dans un langage non-standard comportant des quantifications universelles et des implications [SP03]. J'ai également donné un algorithme permettant de résoudre ces contraintes dans le cas du sous-typage structurel [Sim03a]. Ces travaux permettent donc d'envisager, à court terme, une extension intéressante de Flow Caml.

Perspectives

Au terme de ce travail, il me semble que la principale limitation de ces deux systèmes d'analyse de flots d'information — comme de la plupart des autres qui ont été étudiés sur le papier — vient de la propriété de sécurité que l'on cherche à vérifier : une forme de *non-interférence*. Par certains aspects, cette propriété est trop *faible*, car, comme je l'ai expliqué dans l'introduction, elle ne prend pas en considération certaines caractéristiques observables lors de l'exécution des programmes. De ce fait, il n'est pas raisonnable d'utiliser un outil tel que Jif ou Flow Caml pour analyser un programme fourni par un tiers qui n'est pas de confiance, et l'exécuter sans autre vérification : il serait en effet très facile au programmeur mal intentionné d'effectuer des opérations illicites qui ne seraient pas détectées par l'analyse, par exemple en utilisant un canal lié au temps d'exécution. Cela ne signifie pas que ces outils sont inutiles : ils peuvent en revanche être utilisés pour *aider* les développeurs à écrire des programmes sûrs.

À l'inverse, la non-interférence se montre dans d'autres cas trop être une propriété trop *forte*, car elle empêche toute fuite d'information, ou bien les approximations effectuées par l'analyse statique sont trop grossières. Un premier exemple concerne la *déclassification* des données. Dans de nombreux programmes, on souhaite pouvoir comparer une chaîne entrée par l'utilisateur à un mot de passe (*secret*) stocké en mémoire, et considérer le résultat de ce test comme *public* : il ne révèle en effet qu'un seul bit d'information sur le mot de passe, ce qui paraît raisonnable, à condition que le test ne puisse être réalisé de manière répétitive. Puisque la non-interférence empêche toute fuite d'information, même partielle, une telle déclassification ne peut être effectué en Flow Caml, à moins de faire appel à une fonction non typée (dont la correction ne sera pas prouvée). Ce problème a été considéré par de nombreux travaux, mais aucune solution aboutissant à un système de type pratiquement utilisable n'a été proposée. Zdancewic et Myers ont défini une notion de *déclassification robuste* [ZM01b] dont le but est d'empêcher une déclassification légalement introduite par le programmeur d'être exploitée par un attaquant : en quelques mots, elle garantit qu'un attaquant actif (qui peut affecter le comportement du programme) ne peut obtenir plus d'information qu'un attaquant passif (qui se contente de l'observer). Volpano et Smith se sont spécifiquement intéressés au cas de la vérification de mots de passe [VS00, Vol00].

Une deuxième limitation des langages Flow Caml et Jif concerne leur caractère séquentiel : on

souhaiterait en effet pouvoir analyser des programmes faisant intervenir plusieurs processus, comme le permet par exemple la bibliothèque `Threads` d'Objective Caml. Il s'agit d'une question difficile : l'exécution parallèle de plusieurs parties d'un programme permet à ce dernier d'observer *lui-même* des canaux cachés que j'ai négligé jusqu'ici, car ils n'étaient observables que de manière extérieure. Plusieurs chercheurs se sont intéressés à ce problème, en proposant des systèmes de types de plus en plus évolués [SV98, VS98, VS99, SS00, Sab01, Smi01, Pot02, HY02, BC01, BC02, MS03]. Leur introduction dans celui présenté dans cette thèse nécessiterait de considérer une propriété de non-interférence *forte* (*i.e.* prenant en compte la terminaison des programmes), ce qui serait alors probablement inacceptable pour des programmes séquentiels. Zdancewic et Myers [ZM03] ont proposé de se restreindre à des programmes dont le comportement paraît déterministe aux attaquants. Cette limitation leur permet de ne pas affecter le traitement des programmes purement séquentiels, mais cela exclut un certain nombre d'exemples intéressants de programmes concurrents.

Comme je l'ai mentionné à plusieurs reprises, l'analyse typée de flots d'information ne permet pas une étude satisfaisante de certains programmes effectuant des opérations de trop bas niveau. Un exemple intéressant est celui des protocoles cryptographiques, auquel est consacré un autre axe de recherche. Abadi et Gordon [AG99] ont défini une extension du π-calcul dotée de primitives cryptographiques, le *spi-calcul*. Abadi et Blanchet [AB01b, AB01a] puis Gordon et Jeffrey [GJ02], ont développé des systèmes de types pour analyser les propriétés de confidentialité dans les protocoles exprimés avec ce langage. Parallèlement, Blanchet [Bla01, AB01a] a développé un vérificateur de protocoles basé sur une représentation de ceux-ci sous forme de règles Prolog, dont les résultats obtenus sur des exemples réels semblent particulièrement intéressants. Cependant, ces outils ne permettent d'analyser que des fragments relativement spécialisés des programmes, à l'inverse de l'analyse de flots d'information qui cherche à fournir un outil relativement généraliste. Un axe de recherche intéressant consisterait à essayer de rapprocher ces deux voies.

Appendices

Table des figures

Table des notations

▶ Minuscules romaines

▶ Majuscules romaines

▶ Majuscules romaines doubles

▶ Majuscules cursives

▶ Minuscules grecques

▶ Majuscules grecques

Index

Bibliographie

[AB01a] Martín Abadi et Bruno Blanchet. Analyzing Security Protocols with Secrecy Types and Logic Programs. In *Proceedings of the 29th ACM symposium on Principles of Programming Languages (POPL)*, pages 33–44. Portland, Oregon, avril 2001.
http://www.di.ens.fr/~blanchet/publications/AbadiBlanchetPOPL02.html

[AB01b] Martín Abadi et Bruno Blanchet. Secrecy Types for Asymmetric Communication. In *Proceedings of the International Conference on Foundations of Software Science and Computation Structures (FOSSACS)*, tome 2030 de *Lecture Notes in Computer Science*, pages 25–41. Springer Verlag, avril 2001.
http://www.di.ens.fr/~blanchet/fossacs01.html

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze et Jon G. Riecke. A Core Calculus of Dependency. In *Proceedings of the 26th ACM symposium on Principles of Programming Languages (POPL)*, pages 147–160. ACM Press, janvier 1999.
http://www.soe.ucsc.edu/~abadi/Papers/flowpopl.ps

[AF96] Alexander S. Aiken et Manuel Fähndrich. Making Set-Constraint Based Program Analyses Scale. Computer Science Division Technical Report CSD-96-917, University of California, Berkeley, septembre 1996.
http://research.microsoft.com/~maf/scw96.ps.gz

[AG99] Martin Abadì et Andrew D. Gordon. A Calculus for Cryptographic Protocols : The Spi Calculus. *Information and Computation*, tome 148(1), pages 1–70, janvier 1999.
http://www.cse.ucsc.edu/~abadi/Papers/spi.ps

[Aik94] Alexander S. Aiken. The Illyria system, 1994.
http://http.cs.berkeley.edu:80/~aiken/Illyria-demo.html

[Aik99] Alexander S. Aiken. Introduction to Set Constraint-Based Program Analysis. *Science of Computer Programming*, tome 35, pages 79–111, 1999.
http://www.cs.berkeley.edu/~aiken/publications/papers/scp99.ps

[ALL96] Martín Abadi, Butler Lampson et Jean-Jacques Lévy. Analysis and Caching of Dependencies. In *Proceedings of the 1st ACM International Conference on Functional Programming (ICFP)*, pages 83–91. ACM Press, Philadelphia, Pennsylvania, mai 1996.
http://www.soe.ucsc.edu/~abadi/Papers/make-preprint.ps

[AR80] Gregory R. Andrews et Richard P. Reitman. An Axiomatic Approach to Information Flow in Programs. *ACM Transactions on Programming Languages and Systems*, tome 2(1), pages 56–76, janvier 1980.

[AW92] Alexander S. Aiken et Edward L. Wimmers. Solving Systems of Set Constraints. In *Proceedings of the 7th IEEE Symposium on Logic in Computer Science*, pages 329–340. IEEE Computer Society Press, juin 1992.
http://http.cs.berkeley.edu/~aiken/ftp/lics92.ps

[AW93] Alexander S. Aiken et Edward L. Wimmers. Type Inclusion Constraints and Type Inference. In *Proceedings of the 6th Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 31–41. ACM Press, 1993.
http://http.cs.berkeley.edu/~aiken/ftp/fpca93.ps

[AWP96] Alexander S. Aiken, Edward L. Wimmers et Jens Palsberg. Optimal Representations of Polymorphic Types with Subtyping. Rapport technique CSD-96-909, University of California, Berkeley, juillet 1996.
http://http.cs.berkeley.edu/~aiken/ftp/quant.ps

[BBL94] Jean-Pierre Banâtre, Ciarán Bryce et Daniel Le Métayer. Compile-time detection of information flow in sequential programs. In *European Symposium on Research in Computer Security*, tome 875 de *Lecture Notes in Computer Science*, pages 55–74. Springer Verlag, 1994.
ftp://ftp.irisa.fr/local/lande/dlm-esorics94.ps.Z

[BC01] Gérard Boudol et Ilaria Castellani. Noninterference for concurrent programs. In *Proceedings of the 28th International Colloquium Automata, Languages, and Programming (ICALP)*, tome 2076 de *Lecture Notes in Computer Science*, pages 382–395. juillet 2001.
ftp://ftp-sop.inria.fr/mimosa/personnel/gbo/non-interf.ps.gz

[BC02] Gérard Boudol et Ilaria Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, tome 281(1), pages 109–130, juin 2002.

[BL73] D. E. Bell et L. J. LaPadula. Secure computer systems : Mathematical foundations. Rapport technique MTR-2547, MITRE Corporation, Bedford, MA, 1973.

[Bla01] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW)*. Cape Breton, Nova Scotia, juin 2001.

[BN02] Anindya Banerjee et David A. Naumann. Secure Information Flow and Pointer Confinement in a Java-like Language. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW)*, pages 253–270. IEEE Computer Society Press, juin 2002.
http://guinness.cs.stevens-tech.edu/~naumann/publications/csfw15.ps

[Coh77] Ellis S. Cohen. Information transmission in computational systems. *ACM SIGOPS Operating Systems Review*, tome 11(5), pages 133–139, 1977.

[Coh78] Ellis S. Cohen. Information transmission in sequential programs. *Foundations of Secure Computation*, pages 297–335, 1978.

[DD77] Dorothy E. Denning et Peter J. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, tome 20(7), pages 504–513, juillet 1977.

[Den82] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, Massachusetts, 1982. ISBN 0-201-10150-5.

[Dep85] Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria*. décembre 1985. DOD 5200.28-STD (The Orange Book) edition.

[EST95] Jonathan Eifrig, Scott Smith et Valery Trifonov. Sound polymorphic type inference for objects. *ACM SIGPLAN Notices*, tome 30(10), pages 169–184, 1995.
http://rum.cs.yale.edu/trifonov/papers/sptio.ps.gz

[Fei80] Richard J. Feiertag. A Technique for proving Specifications are multilevel secure. Rapport technique CSL-109, SRI International Computer Science Lab, Menlo Park, California, janvier 1980.

[Fen73] J. S. Fenton. *Information Protection Systems*. Thèse de doctorat, University of Cambridge, Cambridge, UK, 1973.

[Fen74] J. S. Fenton. Memoryless sybsystems. *Computing Journal*, tome 17(2), pages 143–147, mai 1974.

[FF96] Cormac Flanagan et Matthias Felleisen. Modular and Polymorphic Set-Based Analysis : Theory and Practice. Rapport technique TR96-266, Rice University, novembre 1996.
http://www.cs.rice.edu/CS/PLT/Publications/Scheme/tr96-266.ps.gz

[FF97] Cormac Flanagan et Matthias Felleisen. Componential Set-Based Analysis. In *Proceedings of the 10th ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 235–248. ACM Press, Las Vegas, Nevada, juin 1997.
http://www.cs.rice.edu/CS/PLT/Publications/Scheme/pldi97-ff.ps.gz

[FFA99] Jeffrey S. Foster, Manuel Fähndrich et Alexander Aiken. A Theory of Type Qualifiers. In *Proceedings of the 12th ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 192–203. ACM Press, Atlanta, Georgia, mai 1999.
http://www.cs.umd.edu/~jfoster/papers/pldi99.ps.gz

[FG95] Riccardo Focardi et Roberto Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, tome 3(1), pages 5–33, 1995.
http://www.cs.unibo.it/~gorrieri/Papers/jcsfinal.ps.gz

[FM88] You-Chin Fuh et Prateek Mishra. Type inference with subtypes. In H. Ganzinger, rédacteur, *Proceedings of the European Symposium on Programming (ESOP)*, tome 300 de *Lecture Notes in Computer Science*, pages 94–114. Springer Verlag, 1988.

[FM89] You-Chin Fuh et Prateek Mishra. Polymorphic Subtype Inference : Closing the Theory-Practice Gap. In J. Díaz et F. Orejas, rédacteurs, *Proceedings of the European Joint Conferences on Theory and Practice of Software (ETAPS)*, tome 352 de *Lecture Notes in Computer Science*, pages 167–183. Springer Verlag, Berlin, mars 1989.

[Fre97] Alexandre Frey. Satisfying Subtype Inequalities in Polynomial Space. In Pascal Van Hentenryck, rédacteur, *Proceedings of the 4th International Static Analysis Symposium (SAS)*, numéro 1302 in Lecture Notes in Computer Science, pages 265–277. Springer Verlag, Paris, France, septembre 1997.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba et Matthias Felleisen. The Essence of Compiling with Continuations. In *Proceedings of the 6th ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 237–247. ACM Press, Albuquerque, New Mexico, juin 1993.
http://www.cs.rice.edu/CS/PLT/Publications/pldi93-fsdf.ps.gz

[FT90] John Field et Tim Teitelbaum. Incremental Reduction in the Lambda Calculus. In *Proceedings of the 1st ACM Conference on LISP and Functional Programming (LFP)*, pages 307–322. ACM Press, juin 1990.

[FTA02] Jeffrey S. Foster, Tachio Terauchi et Alex Aiken. Flow-Sensitive Type Qualifiers. In *Proceedings of the 15th ACM Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 2002.
http://http.cs.berkeley.edu/~jfoster/papers/pldi02.ps.gz

[GJ02] Andrew D. Gordon et Alan Jeffrey. Types and Effects for Asymmetric Cryptographic Protocols. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW)*, pages 77–91. Cape Breton, Nova Scotia, juin 2002.
http://research.microsoft.com/~adg/Publications/spi-v2.pdf

[GM82] Joseph Goguen et José Meseguer. Security policies and security models. In *Proceedings of the 3rd Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, Oakland, California, avril 1982.

[GP02] Murdoch J. Gabbay et Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, tome 13(3–5), pages 341–363, juillet 2002.

[GSSS01] Kevin Glynn, Peter J. Stuckey, Martin Sulzmann et Harald Søndergaard. Boolean Constraints for Binding-Time Analysis. In O. Danvy et A. Filinsky, rédacteurs, *Proceedings of the 2nd Symposium on Programs as Data Objects (PADO)*, tome 2053 de *Lecture Notes in Computer Science*, pages 39–63. Springer Verlag, 2001.

[Hen93] Fritz Henglein. Type Inference with Polymorphic Recursion. *ACM Transactions on Programming Languages and Systems*, tome 15(2), pages 253–289, 1993.

[HM95] My Hoang et John C. Mitchell. Lower Bounds on Type Inference with Subtypes. In *Proceedings of the 22nd ACM symposium on Principles of Programming Languages (POPL)*, pages 176–185. ACM Press, New York, NY, USA, janvier 1995.
http://theory.stanford.edu/people/jcm/papers/subtype-popl95.dvi

[HM97] Nevin Heintze et David McAllester. Linear-Time Subtransitive Control Flow Analysis. In *Proceedings of the 10th ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 261–272. ACM Press, juin 1997.
http://www.autoreason.com/PLDI97.ps

[HR98] Nevin Heintze et Jon G. Riecke. The SLam Calculus : Programming with Secrecy and Integrity. In *Proceedings of the 25th ACM symposium on Principles of Programming Languages (POPL)*, pages 365–377. ACM Press, janvier 1998.
http://cm.bell-labs.com/cm/cs/who/nch/slam.ps

[Hue76] Gérard Huet. *Résolution d'équations dans les langages d'ordre* $1, 2, \ldots \omega$. Thèse de doctorat, Université Paris 7, 1976.

[HY02] Kohei Honda et Nobuko Yoshida. A Uniform Type Structure for Secure Information Flow. In *Proceedings of the 29th ACM symposium on Principles of Programming Languages (POPL)*, pages 81–92. ACM Press, Portland, Oregon, janvier 2002.
http://www.mcs.le.ac.uk/~nyoshida/paper/ifa_long.ps.gz

[Knu68] Donald E. Knuth. *Fundamental Algorithms*, tome 1 de *The Art of Computer Programming*. Addison-Wesley, 1968.

[KR03] Viktor Kuncak et Martin Rinard. Structural Subtyping of Non-Recursive Types is Decidable. In *Proceedings of the 18th IEEE Symposium on Logic in Computer Science*. Ottawa, Canada, juin 2003.
http://cag.lcs.mit.edu/~rinard/paper/lics03.pdf

[Lam73] Butler W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, tome 16(10), pages 613–615, octobre 1973.
http://research.microsoft.com/lampson/11-Confinement/WebPage.html

[LDG+a] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy et Jérôme Vouillon. The Objective Caml manual.
http://caml.inria.fr/

[LDG+b] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy et Jérôme Vouillon. The Objective Caml system.
http://caml.inria.fr/

[Ler] Xavier Leroy. The Caml-Light compiler.
http://caml.inria.fr/

[Ler00] Xavier Leroy. A modular module system. *Journal of Functional Programming*, tome 10(3), pages 269–303, 2000.
http://pauillac.inria.fr/~xleroy/publi/modular-modules-jfp.ps.gz

[McA96] David McAllester. Inferring Recursive Data Types, juillet 1996. Draft manuscript.
http://www.autoreason.com/equiv.ps

[McA03] David McAllester. A Logical Algorithm for ML Type Inference. In *Proceedings of the 14th Rewriting Techniques and Applications (RTA)*, tome 2706 de *Lecture Notes in Computer Science*, pages 436–451. Springer Verlag, juin 2003.
http://www.autoreason.com/rta03.ps

[McL92] John McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, tome 1(1), pages 37–58, 1992.

[MG85] K. McHugh et D. I. Good. An Information Flow Tool for Gipsy. In *Proceedings of the 6th Symposium on Security and Privacy*, pages 46–48. IEEE Computer Society Press, 1985.

[Mit84] John C. Mitchell. Coercion and type inference. In *Proceedings of the 11th ACM symposium on Principles of Programming Languages (POPL)*, pages 175–185. ACM Press, Salt Lake City, janvier 1984.

[Mit91] John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, tome 1(3), pages 245–286, 1991.
http://theory.stanford.edu/people/jcm/papers/simple-sub.ps

[MNZZ01] Andrew C. Myers, Nathaniel Nystrom, Lantian Zheng et Steve Zdancewic. Jif : Java + information flow, septembre 2001.
http://www.cs.cornell.edu/jif/

[Mog89] Eugenio Moggi. Computational λ-Calculus and Monads. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, pages 14–23. IEEE Computer Society Press, Asilomar, Pacific Grove, California, juin 1989.
http://www.disi.unige.it/person/MoggiE/ftp/lics89.ps.gz

[mos] Moscow ML.
http://www.dina.dk/~sestoft/mosml.html

[MS03] Heiko Mantel et Andrei Sabelfeld. A Unifying Approach to the Security of Distributed and Multi-Threaded Programs. *Journal of Computer Security*, tome 11(4), pages 615–676, 2003.
http://www.cs.cornell.edu/~andrei/mantel-sabelfeld-jcs.ps

[MTHM97] Robin Milner, Mads Tofte, Robert Harper et David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[Mye99a] Andrew C. Myers. JFlow : practical mostly-static information flow control. In *Proceedings of the 26th ACM symposium on Principles of Programming Languages (POPL)*, pages 228–241. ACM Press, janvier 1999.
http://www.cs.cornell.edu/andru/papers/popl99/myers-popl99.ps.gz

[Mye99b] Andrew C. Myers. *Mostly-Static Decentralized Information Flow Control*. Thèse de doctorat, Massachusetts Institute of Technology, janvier 1999. Rapport technique MIT/LCS/TR-783.
http://www.cs.cornell.edu/andru/release/tr783.ps.gz

[Mye01] Andrew C. Myers. Communication personnelle, juin 2001.

[NP01] Joachim Niehren et Tim Priesnitz. Non-Structural Subtype Entailment in Automata Theory. In *Proceedings of the 4th International Symposium on Theoretical Aspects of Computer Software (TACS)*, tome 2215. Springer Verlag, octobre 2001.
ftp://ftp.ps.uni-sb.de/pub/papers/ProgrammingSysLab/pauto.ps.gz

[NP03] Joachim Niehren et Tim Priesnitz. Non-Structural Subtype Entailment in Automata Theory. *Information and Computation*, tome 186(2), pages 319–354, 2003.
http://www.ps.uni-sb.de/Papers/abstracts/subtype.pdf

[ØP97] Peter Ørbæk et Jens Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, tome 7(6), pages 557–591, novembre 1997.
http://www.cs.purdue.edu/homes/palsberg/paper/jfp97.ps.gz

[OSW99] Martin Odersky, Martin Sulzmann et Martin Wehr. Type Inference with Constrained Types. *Theory and Practice of Object Systems (TAPOS)*, tome 5(1), pages 35–55, 1999.
http://www.cs.mu.oz.au/~sulzmann/publications/tapos.ps

[PC00] François Pottier et Sylvain Conchon. Information Flow Inference for Free. In *Proceedings of the 5th ACM International Conference on Functional Programming (ICFP)*, pages 46–57. Montréal, Canada, septembre 2000.
http://pauillac.inria.fr/~fpottier/publis/fpottier-conchon-icfp00.ps.gz

[Pey03] Simon Peyton Jones, rédacteur. *Haskell 98 Language and Libraries : The Revised Report*. Cambridge University Press, avril 2003.
http://www.haskell.org/onlinereport/

[PØ95] Jens Palsberg et Peter Ørbæk. Trust in the λ-calculus. In *Proceedings of the 2nd International Static Analysis Symposium (SAS)*, tome 983 de *Lecture Notes in Computer Science*, pages 314–330. Springer Verlag, septembre 1995.
ftp://ftp.daimi.au.dk/pub/empl/poe/lambda-trust.dvi.gz

[Pot98] François Pottier. *Synthèse de types en présence de sous-typage : de la théorie à la pratique*. Thèse de doctorat, Université Paris 7, juillet 1998.
http://pauillac.inria.fr/~fpottier/publis/these-fpottier.ps.gz

[Pot00] François Pottier. A Versatile Constraint-Based Type Inference System. *Nordic Journal of Computing*, tome 7(4), pages 312–347, novembre 2000.
http://pauillac.inria.fr/~fpottier/publis/fpottier-njc-2000.ps.gz

[Pot01a] François Pottier. A semi-syntactic soundness proof for HM(X). Rapport de recherche 4150, Institut National de Recherche en Informatique et en Automatique, mars 2001.
ftp://ftp.inria.fr/INRIA/publication/RR/RR-4150.ps.gz

[Pot01b] François Pottier. Simplifying subtyping constraints : a theory. *Information and Computation*, tome 170(2), pages 153–183, novembre 2001.
http://pauillac.inria.fr/~fpottier/publis/fpottier-ic01.ps.gz

[Pot02] François Pottier. A Simple View of Type-Secure Information Flow in the π-Calculus. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW)*, pages 320–330. IEEE Computer Society Press, Cape Breton, Nova Scotia, juin 2002.
http://pauillac.inria.fr/~fpottier/publis/fpottier-csfw15.ps.gz

[Pot03] François Pottier. A Constraint-Based Presentation and Generalization of Rows. In *Proceedings of the 18th IEEE Symposium on Logic in Computer Science*. juin 2003.
http://pauillac.inria.fr/~fpottier/publis/fpottier-lics03.ps.gz

[PR03] François Pottier et Didier Rémy. The Essence of ML Type Inference. In Benjamin C. Pierce, rédacteur, *Advanced Topics in Types and Programming Languages*. MIT Press, Cambridge, Massachusetts, 2003. À paraître.

[PS02] François Pottier et Vincent Simonet. Information Flow Inference for ML. In *Proceedings of the 29th ACM symposium on Principles of Programming Languages (POPL)*, pages 319–330. ACM Press, Portland, Oregon, janvier 2002.
http://cristal.inria.fr/~simonet/publis/fpottier-simonet-popl02.ps.gz

[PS03] François Pottier et Vincent Simonet. Information Flow Inference for ML. *ACM Transactions on Programming Languages and Systems*, tome 25(1), pages 117–158, janvier 2003.
http://pauillac.inria.fr/~fpottier/publis/fpottier-simonet-toplas.ps.gz

[Reh97] Jakob Rehof. Minimal Typings in Atomic Subtyping. In *Proceedings of the 24th ACM symposium on Principles of Programming Languages (POPL)*, pages 278–291. ACM Press, Paris, France, janvier 1997.
http://research.microsoft.com/~rehof/popl97.ps

[Rém90] Didier Rémy. *Algèbres Touffues. Application au Typage Polymorphe des Objets Enregistrements dans les Langages Fonctionnels.* Thèse de doctorat, Université Paris 7, 1990.
ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/these.ps.gz

[Rém92] Didier Rémy. Extending ML Type System with a Sorted Equational Theory. Rapport de recherche 1766, Institut National de Recherche en Informatique et en Automatique, 1992.
ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/eq-theory-on-types.ps.gz

[Sab01] Andrei Sabelfeld. The Impact of Synchronisation on Secure Information Flow in Concurrent Programs. In *Proceedings of the Andrei Ershov 4th International Conference on Perspectives of System Informatics*, tome 2244 de *Lecture Notes in Computer Science*, pages 227–241. Springer Verlag, Akademgorodok, Novosibirsk, Russia, juillet 2001.
http://www.cs.cornell.edu/~andrei/sabelfeld-psi01.ps

[SHO98] Bratin Saha, Nevin Heintze et Dino Oliva. Subtransitive CFA using Types. Rapport technique, Yale University, Department of Computer Science, octobre 1998.
http://flint.cs.yale.edu/flint/publications/cfa.ps.gz

[Sim] Vincent Simonet. The Flow Caml system.
http://cristal.inria.fr/~simonet/soft/flowcaml/

[Sim02a] Vincent Simonet. Dalton, an efficient implementation of type inference with structural subtyping, octobre 2002.
http://cristal.inria.fr/~simonet/soft/dalton/

[Sim02b] Vincent Simonet. Fine-grained Information Flow Analysis for a λ-calculus with Sum Types. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW)*, pages 223–237. IEEE Computer Society Press, Cape Breton, Nova Scotia (Canada), juin 2002.
http://cristal.inria.fr/~simonet/publis/simonet-csfw-02.ps.gz

[Sim03a] Vincent Simonet. An Extension of HM(X) with Bounded Existential and Universal Data-Types. In *Proceedings of the 8th ACM International Conference on Functional Programming (ICFP)*, pages 39–50. Uppsala, Sweden, août 2003.
http://cristal.inria.fr/~simonet/publis/simonet-icfp03.ps.gz

[Sim03b] Vincent Simonet. The Flow Caml System : documentation and user's manual. Rapport technique 0282, Institut National de Recherche en Informatique et en Automatique, juillet 2003.
http://www.inria.fr/rrrt/rt-0282.html

[Sim03c] Vincent Simonet. Type inference with structural subtyping : A faithful formalization of an efficient constraint solver, septembre 2003. Full version.
http://cristal.inria.fr/~simonet/publis/simonet-aplas03-full.ps.gz

[SM03] Andrei Sabelfeld et Andrew C. Myers. Language-Based Information-Flow Security. *Journal Selected Areas in Communications*, tome 21(1), pages 5–19, janvier 2003.
http://www.cs.cornell.edu/~andrei/jsac.ps

[Smi01] Geoffrey S. Smith. A New Type System for Secure Information Flow. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW)*, pages 115–125. IEEE Computer Society Press, Cape Breton, Nova Scotia (Canada), juin 2001.
http://www.cs.fiu.edu/~smithg/papers/csfw01.ps

[sml] Standard ML of New Jersey.
http://www.smlnj.org/

[SP03] Vincent Simonet et François Pottier. Constraint-Based Type Inference for Guarded Algebraic Data Types, juillet 2003. Submitted for publication.
http://cristal.inria.fr/~simonet/publis/simonet-pottier-hmg.ps.gz

[SS00] Andrei Sabelfeld et Dave Sands. Probabilistic Noninterference for Multi-threaded Programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW)*, pages 200–214. IEEE Computer Society Press, Cambridge, England, juillet 2000.
http://www.cs.cornell.edu/~andrei/probnon.ps

[SV98] Geoffrey Smith et Dennis Volpano. Secure Information Flow in a Multi-Threaded Imperative Language. In *Proceedings of the 25th ACM symposium on Principles of Programming Languages (POPL)*, pages 355–364. janvier 1998.
http://www.cs.nps.navy.mil/people/faculty/volpano/papers/popl98.ps.Z

[Tar72] Robert E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, tome 1(2), pages 146–160, 1972.

[Tar75] Robert Endre Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm. *Journal of the ACM*, tome 22(2), pages 215–225, avril 1975.

[The] The Apache Group. The Apache HTTP Server.
http://www.apache.org

[Tiu92] Jerzy Tiuryn. Subtype inequalities. In Andre Scedrov, rédacteur, *Proceedings of the 7th IEEE Symposium on Logic in Computer Science*, pages 308–317. IEEE Computer Society Press, Santa Cruz, CA, juin 1992.

[Tof88] Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. Thèse de doctorat, University of Edinburgh, 1988.
http://www.diku.dk/users/tofte/publ/thesis-part1and2.ps.gz

[TS96] Valery Trifonov et Scott Smith. Subtyping Constrained Types. In *Proceedings of the 3rd International Static Analysis Symposium (SAS)*, tome 1145 de *Lecture Notes in Computer Science*, pages 349–365. Springer Verlag, septembre 1996.
http://rum.cs.yale.edu/trifonov/papers/subcon.ps.gz

[TW93] Jerzy Tiuryn et Mitchell Wand. Type Reconstruction with Recursive Types and Atomic Subtyping. In *Proceedings of the 3rd International Joint Conference on the Theory and Practice of Software Development (TAPSOFT)*, tome 668 de *Lecture Notes in Computer Science*, pages 686–701. Springer Verlag, avril 1993.
ftp://ftp.ccs.neu.edu/pub/people/wand/papers/caap-93.dvi

[Vol00] Dennis Volpano. Secure Introduction of One-way Functions. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW)*, pages 246–259. IEEE Computer Society Press, Cambridge, England, juillet 2000.
http://www.cs.nps.navy.mil/people/faculty/volpano/papers/csfw00.ps.gz

[VS97a] Dennis Volpano et Geoffrey Smith. Eliminating Covert Flows with Minimum Typings. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop (CSFW)*, pages 156–168. IEEE Computer Society Press, juin 1997.
http://www.cs.nps.navy.mil/people/faculty/volpano/papers/csfw97.ps.Z

[VS97b] Dennis Volpano et Geoffrey Smith. A Type-Based Approach to Program Security. In *Proceedings of the 7th International Joint Conference on the Theory and Practice of Software Development (TAPSOFT)*, tome 1214 de *Lecture Notes in Computer Science*, pages 607–621. Springer Verlag, avril 1997.
http://www.cs.nps.navy.mil/people/faculty/volpano/papers/tapsoft97.ps.Z

[VS98] Dennis Volpano et Geoffrey Smith. Probabilistic noninterference in a concurrent language. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop (CSFW)*, pages 34–43. juin 1998.
http://www.cs.nps.navy.mil/people/faculty/volpano/papers/csfw98.ps.Z

[VS99] Dennis Volpano et Geoffrey Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, tome 7(2–3), pages 231–253, novembre 1999.
http://www.cs.nps.navy.mil/people/faculty/volpano/papers/jcs99.ps.gz

[VS00] Dennis Volpano et Geoffrey Smith. Verifying secrets and relative secrecy. In *Proceedings of the 27th ACM symposium on Principles of Programming Languages (POPL)*, pages 268–276. ACM Press, janvier 2000.
http://www.cs.nps.navy.mil/people/faculty/volpano/papers/popl00.ps.gz

[VSI96] Dennis Volpano, Geoffrey Smith et Cynthia Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, tome 4(3), pages 167–187, 1996.
http://www.cs.nps.navy.mil/people/faculty/volpano/papers/jcs96.ps.Z

[Wad92] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, tome 2, pages 461–493, 1992.
http://www.research.avayalabs.com/user/wadler/papers/monads/monads.ps.gz

[WAL+90] Pierre Weis, Maria-Virginia Aponte, Alain Laville, Michel Mauny et Ascander Suarez. The Caml reference manual, septembre 1990.

[WF94] Andrew K. Wright et Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, tome 115(1), pages 38–94, novembre 1994.
http://www.cs.rice.edu/CS/PLT/Publications/ic94-wf.ps.gz

[Wri93] Andrew K. Wright. Polymorphism for Imperative Languages without Imperative Types. Rapport technique 93-200, Rice University, février 1993.

[Wri95] Andrew K. Wright. Simple Imperative Polymorphism. *Lisp and Symbolic Computation*, tome 8(4), pages 343–356, décembre 1995.
http://www.cs.rice.edu/CS/PLT/Publications/lasc95-w.ps.gz

[XCC03] Hongwei Xi, Chiyan Chen et Gang Chen. Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM symposium on Principles of Programming Languages (POPL)*. ACM Press, janvier 2003.
http://www.cs.bu.edu/fac/hwxi/academic/papers/popl03.ps

[Zda03] Steve Zdancewic. Communication personnelle, octobre 2003.

[ZM01a] Steve Zdancewic et Andrew C. Myers. Secure Information Flow and CPS. In *Proceedings of the 10th European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science. Springer Verlag, avril 2001.
http://www.cs.cornell.edu/zdance/lincont.ps

[ZM01b] Steve Zdancewic et Andrew C. Myers. Secure Information Flow via Linear Continuations. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society Press, Cape Breton, Nova Scotia, juin 2001.
http://www.cis.upenn.edu/~stevez/papers/ZM01b.ps

[ZM02] Steve Zdancewic et Andrew C. Myers. Secure Information Flow via Linear Continuations. *Higher Order and Symbolic Computation (HOSC)*, tome 15(2–3), pages 209–234, septembre 2002.
http://www.cs.cornell.edu/andru/papers/hosc01.ps.gz

[ZM03] Steve Zdancewic et Andrew C. Myers. Observational Determinism for Concurrent Program Security. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society Press, Asilomar, California, juillet 2003.
http://www.cis.upenn.edu/~stevez/papers/ZM03.ps