Typer la désérialisation sans sérialiser les types

Grégoire Henry

Université Paris Diderot – Paris 7
UFR d'Informatique

Thèse
pour l'obtention du diplôme de
Docteur de l'Université Paris Diderot, spécialité informatique

Typer la désérialisation sans sérialiser les types

présentée et soutenue publiquement par

Grégoire Henry

le 17 juin 2011

devant le jury composé de

M. Emmanuel	Chailloux	directeur
M. Roberto	Di Cosmo	président
M. Jacques	Garrigue	rapporteur
M. Benjamin	Goldberg	
M. Thomas	Jensen	rapporteur
M. Michel	Mauny	directeur
M. Julien	Signoles	

Merci.

Ce travail de thèse n'aurait pu aboutir sans le concours de nombreuses personnes, qu'ils soient tous grandement remerciés. Tout d'abord, mes deux directeurs pour leurs soutiens complémentaires et leur confiance au cours de ces trois dernières années et des quelques années qui les ont précédées. Merci Emmanuel pour m'avoir montré comment un travail de recherche sérieux peut se faire dans la bonne humeur et merci Michel pour les nombreuses idées apportées lors de nos discussions devant un tableau (noir de préférence). Merci aussi à Pascal Manoury pour ses relectures multiples et rigoureuses des brouillons successifs de ce document et ses commentaires précieux tant sur le plan scientifique que sur le travail d'écriture.

La soutenance d'aujourd'hui n'aurait pas pu avoir lieu sans le regard extérieur préalable de Jacques Garrigue et Thomas Jensen. Un grand merci à eux d'avoir accepté, malgré leurs emplois du temps chargés, de relire attentivement ce document et d'être présent à ce jury de soutenance. Merci à Roberto Di Cosmo de présider ce même jury et pour sa présence à quelques moments clé, notamment lors de l'entrée dans le grand bain du MPRI ou, en compagnie de Guy Cousineau, lors de l'attribution des bourses doctorales. Merci à Benjamin Goldberg d'avoir traverser l'océan Atlantique pour être présent aujourd'hui, et à Julien Signoles pour d'avoir si chaleureusement accepté de participer à ce jury.

Et puis merci à toutes les personnes rencontrées à PPS ou au LIAFA qui ont permis d'ouvrir d'autres horizons pendant cette thèse. Que ce soit via l'enseignement : merci à Juliusz Chroboczek pour ses grandes théories et sa bienveillance ; merci à François Laroussinie de m'avoir fait confiance sur un sujet qui n'était pas ma spécialité et merci à Mathilde et Sylvain l'équipe de choc des TDs d'algo ; à Dominique Poulalhon, que je n'ose plus vouvoyer, et Antoine Meyer pour les TDs de systèmes. Ou que ce soit par les échanges à la table à café ou dans le sous-marin. Merci aux participants du séminaire thésard d'être revenus une seconde fois. À Christine pour avoir été là à ce moment là. Et à Claire pour les séjours alpins et écossais qui ont suivis. Merci à Julien pour les séances de déboguage de code qui m'ont appris à programmer en C et pour sa relecture au moment elle était utile. Merci à Florian pour m'avoir trouvé une colocation au moment nécessaire, et à France, la colocataire trouvée, pour son accueil et les contrôles d'efficacité. Merci à Séverine pour son écoute et ses questions quand je me perdais dans une preuve, et merci à Gabriel d'être venu me raconter ses preuves quand il s'y perdait. Merci à Pierre

et Gim pour la semaine de rédaction sous le soleil picard. Merci à Sam pour Beckett, et à Mehdi pour Bin-Jip (mais avant qu'on en discute, il faut que tu soutiennes toi aussi). Merci aux thésards plus anciens pour m'avoir accueillis quand j'étais un petit stagiaire de licence : Anne-Gwen, Manu, François, Pierre, Michel... et Raphaël pour la dizaine de répétition, au moins, de l'exposé d'Oslo que tu as supportée. Et puis merci à Odile pour veiller sur tous ses gens et pour sa présence même pendant mes absences. Merci aux musiciens : à Alexandre Miguel pour sa passion et La Passion (enfin l'une des deux) ; à Vincent, son hautbois et Telleman ; à Samuel pour sa gouaille ; à Thibaut pour cette source inépuisable de partitions qu'est IMSLP ; et merci à Émilie W., Cécile T. et Philippe L. qui m'ont appris à entendre ces partitions.

Avant la thèse, il y a bien sûr Emmanuel et Michel qui m'ont encouragé à essayer, mais aussi Michèle Soria. Merci à elle pour le parcours AP puis son soutien pendant le MPRI. Merci à Anne Brygoo pour m'avoir patiemment aider à construire mes premières tentatives d'écriture en maîtrise. Et puis merci à Sami pour les discussions qui continuent depuis cette époque lointaine.

Merci à ma famille pour m'avoir soutenu pendant toutes ces années de thèse. Et merci à Antoine et Marion pour leur accueil régulier aux différentes étapes de la rédaction de ce document.

Merci à tous ceux qui ont lu ces remerciements jusqu'au bout d'être présent aujourd'hui et d'avoir relevé la tête de temps en temps pour sourire à l'orateur.

Table des matières

Introduction

La grande majorité des langages de programmation offre des primitives ou des bibliothèques de communication. Elles permettent aux programmes de lire ou d'écrire des données dans des fichiers, dans des bases de données ou encore d'interagir avec d'autres programmes à travers un réseau ou avec l'utilisateur à travers une interface graphique ou textuelle.

Pour mettre en œuvre ces processus de communication plusieurs difficultés techniques sont à résoudre : quelles sont les informations que l'on souhaite pouvoir échanger ? de quelles manières ces informations peuvent-elles être représentées pour être à la fois transportables par le support de communication et compréhensibles par le destinataire ?

Si en informatique la représentation des données à l'intérieur de l'ordinateur et sur le canal de communication est souvent une représentation binaire — à base de 0 et de 1 — les contraintes pesant dans les deux cadres ne sont pas les mêmes. En particulier, sur un canal de communication, un message doit souvent être une séquence, alors que dans un programme, il peut être avantageux de fragmenter une donnée en plusieurs endroits de la mémoire. L'opération de conversion d'une donnée fragmentée vers une représentation séquentielle à même d'être communiquée est souvent appelée *sérialisation*. L'opération inverse est alors appelée *désérialisation*.

Une autre question à résoudre est alors comment rendre utilisable par le programmeur ces mécanismes de communication : notamment, est-ce au programmeur de détailler l'opération de sérialisation des données qu'il manipule vers la représentation adaptée au support de communication, ou est-ce que cette conversion peut être effectuée automatiquement par un outil à la disposition du programmeur ? Cette seconde option, lorsqu'elle est possible, épargne au programmeur une tâche répétitive et une source supplémentaire d'erreurs de programmation.

Cette introduction présente d'un point de vue historique l'évolution des langages de programmation et ses conséquences sur les mécanismes de sérialisation. La première partie s'intéresse à la représentation en mémoire des valeurs manipulées par ces langages, y compris les valeurs incluant du contrôle. La seconde partie se consacre aux systèmes de types associés aux langages de programmation et à leurs interactions avec les mécanismes de sérialisation. Une courte dernière partie présente le plan de ce document.

La sérialisation

Données simples et représentation binaire

Les premières versions du langage Fortran [Backus, 1954], imaginées au début des années 50 pour faciliter l'écriture de programmes de calcul numérique, contiennent l'une des formes de communication les plus simples. Les données manipulées y sont constituées de séquences d'entiers et de nombres réels ; les primitives `READ` et `WRITE` permettant respectivement de lire sur une carte perforée les données d'un calcul, et d'écrire le résultat de ce calcul sur une imprimante.

Chacune de ces primitives doit effectuer une conversion entre une représentation dite *interne* des données, qui est une représentation binaire spécifique au processeur de la machine hôte et une représentation dite *externe* de ces même données, qui est souvent une représentation décimale plus compréhensible par un être humain. Dans le cas d'une impression, cette représentation externe est une suite de caractères, comme `"323"`, ou `"3.1415"` pour une approximation de π. Ces primitives de lecture et d'écriture reçoivent en argument un *format* précisant l'ordonnancement des données attendues ainsi que les fonctions de conversion à utiliser pour chacune d'entre elles. Lors d'une lecture, si le format précise que la prochaine donnée attendue est un entier et qu'un nombre réel est perforé sur la prochaine carte à déchiffrer, l'incohérence sera détectée par la fonction de conversion et l'exécution du programme sera immédiatement suspendue.

L'instruction Fortran suivante permet d'écrire successivement dans les variables `X` et `N`, puis dans les cases `1` à `N` d'un tableau `A`, les données lues :

```
READ 27, X, N, (A(I), I = 1, N)
```

Le `27` au début de cette instruction fait référence au format décrivant la *mise-en-carte* des données lues ainsi que le type de ces données. Dans cet exemple, le format référencé par `27` pourrait être décrit par la ligne suivante :

```
27 FORMAT (F6.2, I3 / (12F6.2))
```

Il précise que la première des cartes perforées lues doit contenir un nombre réel avec au plus 6 chiffres avant la virgule et 2 après (`F6.2`) et un nombre entier d'au plus 3 chiffres (`I3`). Les cartes suivantes doivent contenir 12 nombres réels chacune.

Dans le cas d'une communication directe entre programmes, la représentation externe utilisée peut se rapprocher de la représentation interne et simplifier l'étape de conversion. En particulier, lorsque les primitives de communication sont utilisées pour stocker temporairement des résultats intermédiaires et les relire plus tard dans la suite du calcul, les représentations externes et internes peuvent être identiques. C'est le cas des primitives `READ TAPE` et `WRITE TAPE` du langage Fortran permettant de lire et d'écrire des données sur une bande magnétique. Néanmoins, lorsque la communication s'effectue entre des ordinateurs utilisant des représentations internes différentes, ils doivent s'accorder sur une représentation externe commune et au moins une étape de conversion reste nécessaire.

Entiers Il existe plusieurs représentions binaires usuelles pour les entiers selon le nombre d'octets utilisés ou l'ordre de ces octets dans la mémoire. Par exemple, l'entier 323 admet les représentations binaires suivantes, où 0..0 représente une séquence de 16 bits à 0 :

Ordre des octets :	Représentation binaire sur 4 octets
Grand boutien (4321) :	0..0 00000001 01000011 (0x00000143)
Petit boutien (1234) :	01000011 10000000 0..0 (0x43010000)
PDP-moyen [1] boutien (3412) :	0..0 01000011 00000001 (0x00004301)

Lors de la définition d'un protocole de communication binaire, il convient de choisir une de ces représentations. Un choix courant des protocoles utilisés sur Internet est l'ordre grand-boutien, comme par exemple dans le norme XDR (External Data Representation, RFC 4506 [2006]). Certains protocoles permettent de négocier la représentation choisie pour communiquer. Ainsi le protocole X11 utilise comme représentation externe la représentation interne du serveur et c'est au client d'effectuer une conversion si nécessaire [Gettys *et al.*, 1990]. Cela évite une double conversion lorsque le serveur et le client utilisent la même représentation et que celle-ci est distincte de celle imposée par la norme.

Nombres réels Au début des années 80, chaque constructeur de matériel informatique utilisait sa propre représentation à virgules flottantes comme approximation des nombres réels. Kahan [1981] en recense plus d'une douzaine. Depuis, la norme IEEE 754 [2008] décompose un nombre réel r en trois composantes : une mantisse M, un exposant E et un bit de signe, tels que $r = \pm M * 10^E$. Elle permet aussi de distinguer quelques valeurs particulières parmi lesquelles $-\infty$, $+\infty$, et NaN [2].

Caractères La plus ancienne norme d'encodage des caractères est la norme américaine ASCII (*American Standard Code for Information Interchange*). Elle a été définie au début des années 60. Chaque caractère y est encodé à l'aide de 7 bits. Elle ne contient donc que 128 caractères et se restreint essentiellement à l'alphabet latin sans accentuation et à quelques signes de ponctuation. En Europe occidentale, il existe plusieurs extensions incompatibles de la norme ASCII, telles que les normes ISO-8859-1, Windows 1252 ou MacOS Roman. Si la mauvaise conversion est appliquée, par exemple lors d'un envoi de mails entre deux ordinateurs utilisant des encodages différents, l'affichage peut être *trÞs spÚcial* (sic) [3]. Pour faciliter cette conversion, un courrier électronique ou une page HTML contiennent en entête une information décrivant le codage des caractères qui suivent. Ainsi l'entête d'un courriel peut contenir la ligne suivante :

```
Content-Type: text/plain; charset=iso-8859-1
```

Elle précise l'encodage utilisé pour le corps du message. Pour que cet entête, lui-même textuel, soit compréhensible par tous les destinataires potentiels du message, la RFC 5322 [2008] impose d'utiliser l'encodage US-ASCII pour les entêtes.

1. Cette disposition se trouvait principalement sur les ordinateurs PDP-11 de DEC.
2. *Not a Number* est utilisé comme résultat d'un calcul non valide, par exemple $+\infty/-\infty$.
3. le codage du caractère 'è' (resp. 'é') dans la norme ISO-8859-1 est identique à celui du caractère Þ (resp. 'Ú') dans la norme IBM-CP850.

La diversité des encodages et les difficultés de conversion sont encore plus complexes dans les langues orientales. Ce n'est que depuis le début des années 90 avec l'apparition de réseau à l'échelle planétaire qu'un effort d'universalisme a été effectué avec la norme Unicode et ses encodages [Unicode Consortium, 2009].

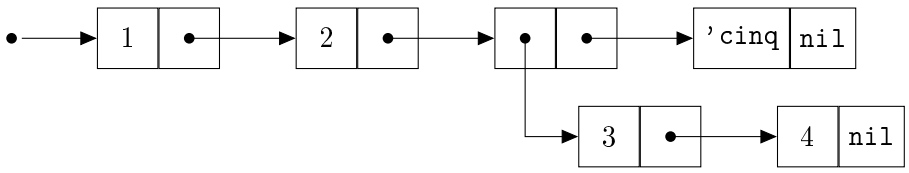
Structures de données et graphe mémoire

Le langage LISP [McCarthy, 1960] a été imaginé à la même époque que le langage Fortran dans le but d'effectuer des calculs symboliques. La structure de données fondamentale de ce langage est la *liste*. Ces listes peuvent contenir d'autres listes ou des *atomes*, comme des entiers, des flottants ou encore des constantes symboliques. Par opposition aux tableaux du Fortran historique, dont le nombre et les tailles sont décidés une fois pour toute par le programmeur avant l'exécution du programme, ces listes sont des structures dynamiques — leur longueur peut varier au cours de l'exécution.

La représentation interne de ces listes est une structure simplement chaînée, c'est-à-dire un ensemble de paires contenant chacune un élément de la liste et un lien vers une autre paire représentant la suite de la liste. La fin d'une liste est marquée par une constante spécifique appelée `nil`. La figure 1 donne une représentation graphique de cette structure de liste chaînée.

Figure 1 Représentation des listes à l'aide de paires

Représentation graphique à l'aide de paires d'une liste à 4 éléments, où le troisième élément est une liste à 2 éléments. Chaque point noir est une référence vers une paire.



Ces paires peuvent être créées lorsque cela est nécessaire au cours de l'exécution d'un programme. Elles peuvent aussi être détruites lorsqu'elles sont devenues inutiles et ainsi permettre de réaffecter l'espace mémoire qui leur était réservé à une autre utilisation. On utilise alors le terme de gestion *dynamique* de la mémoire, par opposition à la gestion *statique* des tableaux Fortran dont l'emplacement en mémoire est définitif.

La représentation interne exacte d'une liste est alors l'*adresse* de la mémoire allouée pour la première paire ; le lien entre chaque paire est l'adresse mémoire de la paire suivante. Une telle adresse dans la zone d'allocation dynamique ne peut pas être utilisée comme représentation externe, c'est une donnée spécifique à l'état actuel d'exécution du programme. Elle n'aura pas de signification dans un autre programme et pourrait même perdre sa signification ultérieurement à l'intérieur du même programme si celle-ci est réaffectée.

Dans le cas de LISP la représentation externe choisie est une représentation textuelle appelée *S-expression*[4] : une liste commence par une parenthèse ouvrante ; elle se termine par une parenthèse fermante ; entre ces deux parenthèses, les éléments sont séparés par des espaces. Par exemple :

```
(1 2 (3 4) 'cinq)
```

est une S-expression qui désigne une liste à 4 éléments. Les deux premiers éléments sont les entiers 1 et 2 ; le troisième est une liste contenant les entiers 3 et 4 ; le dernier élément est le symbole `cinq`. Elle correspond à la liste décrite dans la figure 1.

Par ailleurs, la syntaxe utilisée pour écrire des programmes LISP devait originellement être proche de celles des programmes Fortran [McCarthy, 1978]. Mais cette dernière notation appelée alors *M-expression* n'a jamais été implémentée et l'usage a voulu que les programmes LISP eux-mêmes soient représentés par des S-expressions.

Graphe mémoire Le langage LISP contient des primitives de modification en place d'une donnée. Ainsi, une donnée construite à partir d'une S-expression pourra être modifiée pour représenter des structures d'arbres contenant du partage et des graphes cycliques. On parle alors de *graphe mémoire* pour parler de la représentation mémoire d'une valeur.

Mécanisme générique et informations de type L'opération de sérialisation est réalisée dans le cas de LISP par la primitive `print` qui génère la suite de caractères de la *S-expression* correspondant à la donnée qu'elle reçoit en argument. Réciproquement, la primitive `read` réalise la désérialisation : elle reconstruit un graphe mémoire à partir d'une S-expression.

Contrairement aux primitives Fortran de sérialisation et de désérialisation, celles de LISP ne reçoivent pas de format en argument. Dans le cas de la désérialisation, les S-expressions permettent de reconstruire toute l'information nécessaire sur la structure des valeurs (valeur numériques, atomes, paires). Dans le cas de la sérialisation, le langage LISP impose de pouvoir retrouver pour chaque valeur mémoire une information sur sa nature.

Quelque soit la technique utilisée pour conserver ces informations, bits de *tag* ou autres, ce besoin peut impliquer un surcoût dans l'exécution du programme (voir par exemple Gabriel [1985] pour le cas particulier de LISP). Néanmoins, ces informations sont utiles à d'autres aspects du langage. En particulier, elles sont indispensables à la réalisation du mécanisme de typage dynamique du langage LISP. Elles sont utilisées implicitement par les opérations de base du langage : chacune vérifie que la nature des valeurs qu'elle reçoit en argument sont bien parmi celles qu'elle attend. Par exemple, la primitive d'accès au second élément d'une paire vérifie que son argument est effectivement une paire. Ces informations sur la nature d'une valeur peuvent aussi être utilisées explicitement par le programmeur LISP. On retrouve par exemple des prédicats telles que `intp`, `pairp` ou `nilp` qui testent respectivement si leur argument est un entier, une paire ou la constante `nil`.

4. S pour *Symbolique*

La présence de ces informations est aussi exploitée par le mécanisme de gestion automatique (abrégé en GC, pour *Garbage Collector*) de la mémoire inclus dans LISP. Celui-ci doit pouvoir parcourir l'intégralité du graphe mémoire pour déterminer les données qui ne sont plus utiles à la suite du programme, et ainsi permettre de réutiliser les zones de la mémoire qu'il leur avait allouées.

Si un appel à la fonction générique de désérialisation `read` produit une donnée qui n'a pas la nature ou la structure attendue par le programme, cette incohérence entre les attentes du programmeur et la réalité de la donnée lue ne sera pas détectée au moment de la lecture, au moment de l'utilisation effective de la donnée. En particulier, si cette incohérence se situe en profondeur dans la structure, elle peut n'être détectée qu'un long temps après l'opération de désérialisation. Il est néanmoins possible pour le programmeur d'écrire à l'aide des prédicats de tests de type ses propres fonctions de vérification des valeurs désérialisées. Dans le cas général, ces fonctions devront parcourir l'intégralité des valeurs lues.

Le contrôle d'exécution comme une donnée

Fonction Le langage LISP est aussi un langage dit fonctionnel : une fonction y est une donnée comme une autre. Elle peut être passée en argument à une autre fonction et peut constituer le résultat d'un calcul. À ce titre, il arrive qu'un programmeur souhaite sauvegarder des données fonctionnelles ou qu'il lui semble utile de faire s'échanger ces données entre deux programmes distincts. Pour réaliser ce genre de sérialisation, il existe principalement deux approches : l'une sérialise une représentation du code de la fonction ; l'autre sérialise un identifiant permettant de retrouver ce code lors de la désérialisation.

Il est possible de réaliser la première approche dans le langage LISP à l'aide des primitives déjà étudiées et de la primitive `eval`, qui prend une S-expression correspondant à un code source et calcule une valeur correspondant à l'évaluation de ce programme. Cependant, la plupart des langages de programmation ne contiennent pas ce mécanisme, qui demande à tout programme de contenir un mécanisme d'évaluation du code source, qu'il soit sous la forme d'un interprète ou d'un compilateur muni d'un mécanisme d'édition de liens. Il est néanmoins possible de communiquer le code sous d'autres formes : code-octet, code binaire, assembleur typé, *etc.* pour lesquelles le processus de chargement dynamique demande des mécanismes plus légers.

La seconde approche, qui consiste à ne pas sérialiser le code, qu'il soit sous la forme d'un code source ou sous une forme compilée, est plus courante. On sérialise à la place un identifiant qui va permettre au programme destinataire de trouver le code associé. Dans les mises en œuvre de ce principe, l'identifiant est plus ou moins complexe, selon qu'il doivent par exemple simplement permettre d'identifier un fragment de code commun aux deux programmes communiquants ou qu'il doivent permettre au destinataire de localiser un fragment de code à charger dynamiquement. Des exemples d'identifiants seront détaillés au chapitre 1.

Pour en revenir au cas des langages fonctionnels, et plus particulièrement à celui des langages fonctionnels avec liaison statique des identifiants, tels les dialectes récents de LISP comme Scheme ou le langage ML, la représentation interne usuelle d'une fonction

est une fermeture : celle-ci regroupe à la fois le code de la fonction et l'ensemble des données nécessaires au calcul. En particulier une fermeture doit capturer les données locales au site de définition de la fonction. Dans l'exemple ci-dessous, écrit avec la syntaxe d'Objective Caml, la fonction `creer_compteur` renvoie un couple de fonctions. La première de ces fonctions sert à incrémenter et interroger la valeur d'un compteur ; la seconde réinitialise la valeur de ce compteur avec la constante `init`.

```
let creer_compteur =
  let init = 0 in
  let cpt = ref init in
  (fun () -> incr cpt; !cpt),
  (fun () -> cpt := init)
```

Les fermetures représentant ces deux fonctions doivent capturer dans leur environnement la référence au compteur `cpt`. De manière générale, l'environnement de la seconde fermeture et celui de la fermeture représentant `creer_compteur` doivent capturer la valeur de la constante `init`.

La représentation interne du code est usuellement l'adresse mémoire du code objet. Une fois la représentation externe pour le code choisie, le mécanisme générique de sérialisation/désérialisation déjà présenté peut être étendu avec une fonction de conversion entre ces deux représentations et permettre ainsi de communiquer des valeurs fonctionnelles.

Objet La même problématique se retrouve dans les langages objets tels Smalltalk [Goldberg et Robson, 1983] ou Java [Gosling *et al.*, 2005]. Un objet regroupe un ensemble de données et un ensemble de méthodes effectuant des traitements sur ces données. La présentation courante des objets comme encapsulant des données rend plus évidente encore le souhait de communiquer cette sorte de valeur.

Chaque langage de programmation objet définit sa propre représentation interne pour l'ensemble des méthodes d'un objet [Ducournau, 2011]. Dans certains langages à base de classes, cette représentation peut être arborescente suivant les relations d'héritage. D'autres langages utilisent une représentation regroupant variables et méthodes d'instances dans un même enregistrement [Rémy et Vouillon, 1998]. Différents mécanismes de sérialisation dans les langages objets seront détaillés au chapitre 1.

Sûreté de l'exécution

Typage dynamique et typage statique

En parallèle de l'évolution des valeurs manipulées par les langages de programmation, et face à la complexité croissante des programmes, un autre sujet de recherche a été la sûreté d'exécution : comment, garantir qu'un programme va s'exécuter sans provoquer d'erreur. L'une de ces techniques est le typage statique. Par opposition au typage

dynamique de LISP, le principal objectif de ces systèmes est de vérifier au moment de la compilation qu'aucune erreur de typage ne pourra se produire pendant l'exécution.

Cette vérification demande d'associer au moment de la compilation un type à chaque identifiant et chaque expression d'un programme. Cela impose d'utiliser des langages de types un peu plus expressifs que les types simples évoqués dans la première partie de cette introduction.

Par exemple, dans un langage fonctionnel typé dynamiquement tel LISP, la seule information de type qu'il est nécessaire de conserver pour la bonne exécution d'un appel de fonction est le fait que la valeur représentant la fonction soit effectivement une fermeture. Dans un langage typé statiquement, il est nécessaire de connaître le type de chacun des arguments d'une fonction. Il faut que le langage de types permette de conserver cette information.

Il en est de même avec les objets. Dans un langage dynamique, lors d'un appel de méthode sur un objet, la présence effective de cette méthode dans l'objet est vérifiée au moment d'effectuer l'appel. Dans un langage typé statiquement le type associé à un objet doit à lui seul permettre de garantir la présence de cette méthode dans l'objet, même si le lien effectif avec la méthode appelée ne sera fait que lors de l'exécution. Comme dans le cas des fonctions, ce type doit aussi contenir le type attendu pour les arguments de la méthode appelée.

Généricité

L'approche statique du typage permet de détecter plus rapidement un certain nombre d'erreurs de programmation. Elle permet aussi parfois un gain de temps d'exécution en évitant un certain nombre de vérifications dynamiques. Mais cette approche peut parfois rejeter des programmes qui n'auraient pas provoqué d'erreur de typage à l'exécution. En ce sens, on dit que les systèmes de types sont incomplets. Cela tient au fait qu'un type n'est qu'une abstraction d'un programme et peut ne pas en capturer toutes les utilisations possibles.

Ce défaut est particulièrement flagrant dans les systèmes de types dits *monomorphes* où un seul type peut être associé à un identifiant ou à une expression du langage. Par opposition, les systèmes dits *polymorphes* permettent d'associer plusieurs types à une expression, ou, à défaut, d'associer un seul type représentant un ensemble de types. Cette possibilité permet de diminuer le nombre de programmes rejetés à tort, en permettant, par exemple, à une fonction d'être applicable à des arguments de plusieurs types. Cardelli et Wegner [1985] distinguent 2 grandes catégories de polymorphisme dans les systèmes de types : le polymorphisme *ad hoc* ou apparent, et le polymorphisme universel. Le premier regroupe les systèmes où seul un nombre fini de types peut être associé à un identificateur ; cela inclut les différents mécanismes de surcharge et de *coercion* : au moment où la surcharge est résolue, le contexte n'apporte qu'un ensemble fini de possibilités. Par opposition, le second groupe comprend les systèmes où un ensemble infini de types peut être associé à un identificateur, comme les systèmes à base de sous-typage ou de types paramétrés. Dans ces systèmes, l'ensemble des instances du type associé à un identifiant — c'est-à-dire ses sous-types, ou les instanciations de ses paramètres —

est potentiellement infini : il peut contenir des types qui ne sont pas encore définis.

Le sous-typage est historiquement associé aux objets. Le type d'un objet va déterminer l'ensemble minimum des méthodes qu'il est possible d'appeler sur cet objet. Dans le cas du polymorphisme paramétrique, un type permet de connaître un fragment superficiel de la structure d'une valeur ; ses paramètres vont permettre de compléter cette information. Par exemple, le type `list(`α`)` permet souvent d'identifier une structure de liste chaînée homogène, c'est-à-dire dont tous les éléments ont le même type. Ce type n'étant pas connu, il est noté α. Ces informations partielles sur la structure d'une valeur permettent d'écrire des fonctions génériques opérant sur toutes les listes indépendamment du type de leurs éléments : un exemple classique est une fonction calculant le nombre d'éléments d'une liste.

Sérialisation et typage statique

En présence de fonctions de (dé)sérialisation, il est impossible de vérifier au moment de la compilation le type des valeurs qui seront lues lors des exécutions à venir, et une étape de vérification de compatibilité de la valeur avec le type attendu est nécessaire au moment de la désérialisation.

Certains langages typés statiquement, tels Java, C# ou certaines variantes du langage OCaml, contiennent aussi quelques primitives de typage dynamique [Leroy et Mauny, 1993; Billings *et al.*, 2006]. Dans ces cas, l'environnement d'exécution fournit une représentation des types statiques. Il est possible d'utiliser cette information dynamique de type pour garantir la sûreté de la désérialisation, en sérialisant le type d'une valeur en même temps que la valeur, et lors de la désérialisation d'effectuer une « simple » comparaison entre le type annoncé par la valeur lue et le type attendu par le programme. Cette approche semble aussi possible dans les langages où les types sont des valeurs de premier ordre [Weirich, 2006].

Néanmoins, pour que cette approche soit réalisable, il est nécessaire que les types du langage, et en particulier leur représentation dynamique, aient un sens à l'extérieur d'un programme donné. Ceci est possible par exemple en Java où les classes possèdent un espace de noms et un numéro de version qui permettent de les identifier de manière globalement unique. C'est plus rarement le cas dans les compilateurs des langages fonctionnels actuels.

L'idée de cette thèse est d'explorer une autre définition de compatibilité d'une valeur avec un type. Au lieu de chercher à vérifier dynamiquement la compatibilité du type statique d'une valeur sérialisée avec le type statique attendu par le programme lors de la désérialisation, cette thèse propose une notion de compatibilité mémoire : la valeur mémoire reconstruite par la fonction de désérialisation correspond-elle à une valeur possible pour le type attendu ?

Cette approche rend inutile l'externalisation d'un type ; il reste cependant nécessaire de définir une représentation dynamique des types statiques : elle permet de représenter explicitement le type attendu pour la valeur désérialisée. Cette approche ne permet pas de conserver les invariants usuellement associés aux types abstraits par la signature d'un module. En contrepartie, elle pourrait se révéler plus simple à maintenir lors d'une évo-

lution du système de types et présente l'avantage de permettre une communication sûre avec des programmes « non fiables », ou écrits dans d'autre langage de programmation.

Plan de la thèse

Le premier chapitre de cette thèse revient en détails sur les mécanismes de sérialisation existant dans les langages statiquement typés, en particulier dans les langages objets et les langages fonctionnels. Le second chapitre présente le langage proche de mini-ML qui va servir de cadre initial à cette étude, et permettre de préciser formellement la notion de compatibilité d'une représentation mémoire d'une valeur avec un type. Le troisième chapitre propose un algorithme de vérification de cette relation de compatibilité, ainsi qu'une preuve de correction et de complétude de cet algorithme par rapport à la définition précédente. La quatrième partie décrit une mise en œuvre de cet algorithme dans le compilateur Objective Caml [Leroy *et al.*, 2010] tel que distribué par l'INRIA.

Chapitre 1

Sérialisation et typage statique

L'objet de ce chapitre est de décrire les mécanismes de sérialisation et de désérialisation actuellement utilisés dans certains langages de programmation typés statiquement et en particulier de caractériser les garanties de sûreté d'exécution qu'ils fournissent. Dans cette présentation, ces mécanismes sont regroupés en trois grandes parties, selon la quantité d'information sur les types statiques qu'il est nécessaire de conserver à l'exécution. La première partie de ce chapitre regroupera les langages conservant la totalité des informations de type statiques lors de l'exécution. Une caractéristique commune de ces langages est de permettre la détection d'une incohérence entre le type statique attendu et le type dynamique de la valeur désérialisée immédiatement au moment de la désérialisation par une simple comparaison entre le type statique attendu et le type annoncé par la valeur désérialisée. La seconde partie présentera des langages introduisant une dose de typage dynamique pour conserver la sûreté d'exécution en présence de sérialisation : dans ces systèmes, certaines incohérences de type peuvent ne pas être détectées immédiatement lors de la désérialisation et déclencher une erreur de typage à retardement lors d'une utilisation de la valeur désérialisée. Ces systèmes peuvent ne conserver qu'une partie de l'information de type statique. La troisième partie regroupera les mécanismes *a priori* indépendants de l'information de type disponible dynamiquement et reposant uniquement sur le système de types statique. Ces systèmes n'utilisent pas une unique paire de fonctions génériques de sérialisation et de désérialisation, mais reposent sur un ensemble fonctions de (dé)sérialisation spécifiques à chaque type de données introduit. Dans la plupart des cas ces fonctions peuvent être générées automatiquement à partir des fonctions associées aux types de base et à partir de la définition du type. Une quatrième partie regroupera les différentes solutions existantes dans le cadre le langage OCaml, en les comparant aux solutions décrites dans les autres langages. Une dernière partie présente le mécanisme proposé dans cette thèse.

Les mécanismes présentés ici seront comparés entre autres selon les valeurs qu'elles permettent de sérialiser, notamment s'ils permettent de sérialiser des valeurs contenant du contrôle, selon les abstractions garanties par le système de types statiques qu'ils permettent de préserver, le coût éventuel de la représentation des types à l'exécution, ou encore les contraintes qu'ils imposent au mécanisme de compilation séparée.

1.1 Conserver les types statiques à l'exécution

1.1.1 La sérialisation en Java

Un langage à objets à base de classe Le langage Java est un langage à objets à base de classe. Une classe décrit l'ensemble des variables et des méthodes constituant les objets qui seront instanciés à partir de cette classe. Un mécanisme d'héritage simple permet de définir une classe par spécialisation d'une autre classe, notamment par extension de l'ensemble de variables et de méthodes, ou par redéfinition de certaines de ces méthodes. Le langage impose que toutes les classes héritent d'une même classe appelée `java.lang.Object` ; la hiérarchie d'héritage est alors une unique arborescence.

Un ensemble de définitions de classes peut être regroupé dans un *package* nommé. Ce

package peut lui aussi contenir plusieurs autres *packages*. La documentation du langage incite les développeurs à nommer leur *package* selon un système hiérarchique suivant le principe des noms de domaine Internet : par exemple `fr.inria.ocaml.Callback` désigne la classe `Callback` du *package* `fr.inria.ocaml`. Ce système permet d'espérer un système de nommage globalement unique où le nom d'une classe accompagné de son nom de *package* permet d'identifier cette classe sans confusion possible.

Subsomption et liaison tardive Le système de types utilisé par Java est principalement un système statique où à chaque variable ou expression est associé un type de base ou une classe. Ce système contient un mécanisme de polymorphisme d'inclusion appelé principe de *subsomption* et basé sur la relation d'héritage : si une classe `B` est construite par héritage à partir d'une classe `A`, alors le type `B` est un sous-type du type `A` et toute expression de type `B` a aussi le type `A`. Ce principe permet d'utiliser un objet instancié à partir de la classe `B` là où un objet de type `A` est attendu sans pour autant utiliser de contrainte explicite de sous-typage.

Comme dans tous les langages à objets et comme le reflète le principe de subsomption, Java contient un mécanisme de liaison tardive : lors de l'appel de méthode d'instance, la méthode concrète à appeler peut ne pas être connue statiquement, elle sera déterminée dynamiquement selon la classe ayant servi à instancier l'objet sur lequel la méthode est appelée. Le type statique permet uniquement de garantir la présence d'une méthode ayant la *signature*[1] attendue dans la classe de l'objet présent dynamiquement.

Le langage Java contient aussi un système de classes abstraites, appelées *interface*, et simulant un mécanisme d'héritage multiple. Ce système permet d'assouplir le mécanisme de polymorphisme d'inclusion, en permettant à une classe d'avoir plusieurs sur-types, tout en conservant un héritage simple pour le mécanisme de liaison tardive.

À partir de la version 5 du langage Java, le système de types contient aussi un mécanisme de polymorphisme paramétrique borné appelé *F-bounded polymorphism* [Canning *et al.*, 1989]. Le type défini par une classe peut être paramétré et ce paramètre de type peut être contraint par une borne supérieure pour la relation de sous-typage.

Un peu de typage dynamique Pour permettre encore plus de polymorphisme que le permet son système de types statique, le langage Java contient aussi quelques primitives de typage dynamique.

Lorsqu'un programmeur s'attend à ce que le type dynamique d'un objet — c'est-à-dire le type de sa classe de construction — soit plus précis que le type statique, il doit utiliser le mécanisme de *downcast* : celui-ci permet de préciser le type statique d'une expression en le remplaçant par un de ses sous-types. Pour garantir la sûreté d'exécution, un test dynamique de type doit alors être inséré à chaque *downcast*. Ce mécanisme de *downcast* était fréquemment utilisé pour pallier l'absence de polymorphisme paramétrique dans les premières versions du langage. Il reste nécessaire dans certain cas,

1. La signature d'une méthode en Java comporte le nom de cette méthode et le type de ses arguments et de sa valeur de retour.

notamment pour aider le mécanisme de résolution de surcharge, ou pour certains *design patterns* [Gamma *et al.*, 1995].

La machine virtuelle Java Les programmes Java sont compilés vers un code-objet exécutable par la *machine virtuelle Java* (JVM). Chaque classe est compilée vers un fichier de code-objet chargeable individuellement par la JVM [Liang et Bracha, 1998]. Ce mécanisme de chargement dynamique vérifie la compatibilité de la classe chargée avec les classes déjà existantes et construit la table des méthodes qui sera utilisée par les instances de cette classe.

La sérialisation en Java La bibliothèque standard de Java contient des mécanismes de sérialisation et de désérialisation [Oracle, 2005]. Ces mécanismes génériques peuvent être utilisés sur tous les objets dont la classe a été explicitement déclarée comme respectant l'interface `java.io.Serializable`. La sûreté d'exécution de ces mécanismes repose d'une part sur le mécanisme de chargement de classe et d'autre part sur les primitives de typage dynamique du langage.

Lors de la sérialisation d'un objet, la table des méthodes n'est pas sérialisée, seuls le nom pleinement qualifié de la classe de création et le contenu des variables d'instance sont sérialisés. Le nom de la classe de création permettra à la JVM effectuant la désérialisation d'utiliser son mécanisme de chargement dynamique de classe pour retrouver la table des méthodes associées à l'objet.

Pour garantir la compatibilité des données sérialisées avec la classe chargée par le receveur, le format de sérialisation contient en plus du nom de la classe, un numéro de version ainsi que la liste des noms des variables d'instance de la classe d'origine effectivement sérialisées. Cette liste est utile notamment pour garantir la sûreté d'exécution en cas de conflit de nom de classe. En effet malgré les conventions de nommage hiérarchique utilisées, il reste possible que deux classes distinctes portent le même nom. Dans ce cas, la sûreté d'exécution reste garantie si leurs deux listes de variables d'instance coïncident et si le type dynamique des données désérialisées est effectivement compatible avec le type statique de ces variables dans la classe chargée par le receveur.

En cas de conflit de numéro de version entre la classe de l'objet sérialisé et la classe chargée par le receveur, la liste de noms de variable permet aussi de gérer automatiquement certaines situations où les numéros de version sont distincts. Par exemple si la nouvelle version contient des variables d'instance supplémentaires ou si certaines des variables d'instance communes n'ont pas été sérialisées, le mécanisme de désérialisation initialisera ces variables avec une valeur par défaut. Dans tous les cas, les mécanismes génériques de sérialisation, de désérialisation et de mise à jour peuvent être spécialisés par le programmeur.

Une fois la classe chargée et l'objet correctement reconstruit en mémoire le type de retour de la méthode générique de désérialisation[2] est celui de la classe de tous les objets, c'est-à-dire `java.lang.Object`. C'est ensuite au programmeur d'introduire

2. la méthode `readObject()` de la classe `java.io.ObjectInputStream`.

explicitement un *downcast* vers le type attendu et ainsi imposer une vérification dynamique du type de l'objet désérialisé. Ce test permettra de détecter immédiatement une incohérence entre le type statique attendu et le type dynamique de l'objet.

1.1.2 Polymorphisme paramétrique et langage à objets

L'absence de polymorphisme paramétrique dans les premières versions de Java a motivé de nombreux travaux de recherche visant à son ajout, notamment Odersky et Wadler [1997]; Bracha *et al.* [1998]; Cartwright et Steele [1998]. Si, du point de vue du programmeur, ces propositions diffèrent, notamment, par la syntaxe concrète qu'ils utilisent, ou par le niveau d'intégration au mécanisme d'introspection du langage, elles diffèrent aussi par les techniques de compilation utilisées. Chacune de ces techniques a des implications différentes sur le mécanisme de sérialisation.

Classiquement, ces techniques se répartissent en deux grandes catégories, appelées respectivement approche *hétérogène* et approche *homogène* dans la communauté Java [Odersky et Wadler, 1997]. Dans le premier cas, chaque instanciation distincte du paramètre de type d'une classe paramétrée va générer une nouvelle classe, non paramétrée, mais dont le code et la représentation mémoire ont été spécialisés en fonction du paramètre de type effectif. Dans le second cas, le code généré pour une classe paramétrée sera indépendant du paramètre de type de la classe et la représentation mémoire devra être uniformisée.

Pour ne pas avoir à modifier les JVM déjà installées sur de nombreux ordinateurs, la version actuelle du langage Java utilise une approche homogène dite par effacement et décrite par Bracha *et al.* [1998] : la JVM n'ayant pas initialement prévu d'associer de paramètre de type à une classe, ces paramètres sont effacés dans le code généré. Lors de la compilation d'une classe, chaque occurrence d'un paramètre de type est remplacée par sa borne supérieure, le plus souvent `java.lang.Object`.

Une conséquence de ce choix est de rendre incomplet le mécanisme de *downcast*. En effet lors du test dynamique qui lui est associé, une incompatibilité entre le paramètre du type statique et le paramètre du type dynamique ne peut pas être détectée ; le second ayant été effacé à la compilation. La sûreté de la désérialisation reposant notamment sur le mécanisme de *downcast*, le problème se retrouve à l'identique. La technique utilisée alors dans le compilateur Java pour conserver la sûreté d'exécution en présence de polymorphisme paramétrique et de désérialisation repose sur l'ajout d'une dose supplémentaire de typage dynamique. Son principe est décrit à la section 1.2.1. Avant cela, la fin de cette section décrit deux autres schémas de compilation possibles du polymorphisme paramétrique dans un langage à objets à base de classes tel Java et C#. Pour chacun de ces schémas, les implications sur la sûreté du mécanisme de sérialisation seront détaillées.

Monomorphisation Une des approches étudiées par Odersky et Wadler [1997] pour introduire du polymorphisme paramétrique dans le langage Java consiste à monomorphiser le code généré : chaque instanciation d'une classe polymorphe avec des paramètres

de types différents engendre une nouvelle classe sans paramètre de type et donc compatible avec une machine virtuelle non modifiée. Par exemple, l'utilisation d'une classe `List<A>` avec le paramètre `Int` engendrera une nouvelle classe, nommée par exemple `List_Int`, où toute les occurrences du paramètre de type `A` ont été remplacée par `Int`. Une utilisation de la classe `List<A>` avec le paramètre `String` engendrera une autre classe, nommée par exemple `List_String`. Les classes générées étant sans paramètre de type, cette approche permet de ne pas modifier la JVM.

De plus, cette approche permet de ne pas modifier les propriétés de sûreté d'exécution en présence de désérialisation par rapport à une version de Java sans polymorphisme paramétrique. En effet, lors d'un test dynamique de type lié à un *downcast*, il est possible de distinguer deux instanciations avec des paramètres de types différents d'une même classe polymorphe, ces deux instanciations étant pour la JVM, monomorphe, deux classes différentes.

Cette approche ressemble au mécanisme de *templates* de C++ [Stroustrup, 2000]. Elle possède deux inconvénients majeurs. D'une part, la taille du code généré est sensiblement plus importante, puisqu'au lieu de compiler une seule fois une classe générique, cette classe sera compilée autant de fois que d'instanciations de ses paramètres de type. D'autre part, la monomorphisation complique la réalisation d'un mécanisme de subsomption en présence de covariance ou contravariance du paramètre de type.

Représentation dynamique des paramètres de types Ajouter à un langage à objets à base de classes et sans polymorphisme paramétrique, tel le Java ancestral, une primitive de *downcast* n'introduit pas de surcoût supplémentaire, car il n'est pas nécessaire d'introduire une représentation explicite de l'information de type lors de l'exécution. En effet, la table des méthodes virtuelles nécessaire à la réalisation du mécanisme de liaison tardive peut jouer le rôle de représentant de type : cette table est spécifique à une classe donnée et elle est commune à tous les objets instances de cette classe.

Il faut néanmoins conserver, d'une manière ou d'une autre, dans chaque table des méthodes virtuelles, suffisamment d'information sur la relation d'héritage pour réaliser les tests dynamiques de sous-typage. Cette information étant statique, elle est calculée une seule fois lors de la création de la table des méthodes virtuelles et le surcoût en mémoire est au pire linéaire relativement au nombre de classes et à la profondeur de l'arbre d'héritage.

En présence de polymorphisme paramétrique, cette représentation implicite des types par la table des méthodes virtuelles est conservée si l'on choisit un schéma de compilation par spécialisation : pour chaque instanciation des paramètres de type une nouvelle classe est générée, chacune de ces instanciations introduit donc une nouvelle table des méthodes virtuelles. Cette représentation implicite n'est plus possible dans le cas d'un modèle à partage de code. Dans ce modèle une classe paramétrée est compilée une seule fois vers un code et une représentation mémoire générique. Tous les objets instances de cette classe peuvent donc partager la même table des méthodes virtuelles, indépendamment de ces paramètres de type. Il devient alors nécessaire de conserver explicitement les paramètres de type pour distinguer deux instanciations d'une même classe avec des paramètres de

type différents. Cet ajout d'information peut avoir un coût important en mémoire ou en temps d'exécution s'il n'est pas fait précautionneusement.

Cartwright et Stoler [2002] proposent un compilateur efficace pour NextGen [Cartwright et Steele, 1998], une variante de Java avec polymorphisme paramétrique conservant implicitement les paramètres de type à l'exécution. Pour ne pas modifier la machine virtuelle, leur schéma de compilation repose sur une utilisation subtile du mécanisme d'héritage et d'interface. Pour chaque classe polymorphe, ils génèrent une classe abstraite et générique par effacement du paramètre de type. Puis pour chaque instanciation des paramètres de type, ils génèrent une nouvelle classe minimale ayant la signature attendue et héritant de la classe polymorphe. Cette classe appelée *classe d'instanciation* délègue tous les appels de méthode à la classe polymorphe. Le nom de cette classe encode l'information des paramètres de type. Contrairement au schéma de compilation par monomorphisation, ces classes d'instanciation ne spécialisent pas le code, mais héritent du code générique de la classe polymorphe. La relation d'héritage de ces classes d'instanciations étant différente de la hiérarchie de classe explicitée par le programmeur, celle-ci est reproduite entièrement à l'aide du mécanisme d'interface.

1.1.3 Représentation uniforme des données et sérialisation des types

L'ajout de polymorphisme au langage de programmation typé statiquement a fait l'objet de nombreux travaux de recherche, tant du point de vue de l'expressivité du langage, que des techniques de compilation. Dans le langage Java présenté dans les sections précédentes, le polymorphisme d'inclusion, lié au principe d'héritage, est réalisé à l'aide d'un mécanisme de liaison tardive et d'une table des méthodes. La section 1.1.2 a décrit ensuite deux techniques permettant l'ajout de polymorphisme paramétrique : par spécialisation de code ou par partage de code.

Ces deux principes de compilation du polymorphisme paramétrique se retrouvent dans d'autres langages de programmation, en particulier dans les langages fonctionnels de la famille ML. Par exemple, le compilateur MLton [Weeks *et al.*, 2007] est un compilateur SML agissant par spécialisation du code et de la représentation mémoire. Néanmoins, la plupart des compilateurs de cette famille de langages semble privilégier un principe de compilation par partage de code. La fin de cette section s'attarde un peu plus en détail sur les conséquences de ce schéma de compilation sur le mécanisme de sérialisation. La section 1.3.3 reviendra sur les compilateurs utilisant un mécanisme par spécialisation de code.

Dans un schéma de compilation par partage de code, une fonction ou une méthode doit être compilée avec une information partielle sur le type de ses arguments effectifs. Par exemple, une fonction calculant la longueur d'une liste homogène, de type `List`(α), doit pouvoir être applicable à toutes les instances possibles du type α des éléments de la liste. L'usage est d'utiliser une représentation mémoire dite uniforme, où la taille de la représentation en mémoire d'une valeur doit être indépendante de son type. Généralement, la taille choisie est la taille des adresses mémoire du matériel sous-jacent ; ainsi, les données ne pouvant être représentées dans cet espace limité sont allouées dans le tas et leur adresse mémoire est utilisée comme représentant.

Parcours du graphe mémoire Dans les langages de programmation possédant un gestionnaire automatique de mémoire, ce dernier doit pouvoir discrimer parmi les zones de mémoire allouées sur le tas, celles encore utiles à l'exécution du programme, de celles devenues inutiles. Cela se fait schématiquement en deux étapes : d'abord collecter l'ensemble des adresses mémoires référençant le tas qui sont contenues dans la pile et dans la zone d'allocation statique, ces adresses sont appelées *racines* du graphe mémoire ; puis déterminer l'ensemble des zones mémoire accessibles depuis ces racines, autrement dit parcourir l'intégralité du graphe mémoire. Les zones mémoire allouées dans le tas qui n'ont pas été rencontrées pendant ce parcours sont devenues inutiles pour l'exécution du programme — elles lui sont inaccessibles — le gestionnaire de mémoire peut alors récupérer ces zones.

Pour réaliser ces deux étapes, il est nécessaire de pouvoir distinguer sur la pile et dans le tas, les adresses mémoire des valeurs immédiates, telles des entiers ou des flottants et il est nécessaire de connaître la taille de chaque zone allouée sur le tas. Les techniques pour effectuer cette distinction se répartissent historiquement en deux grandes catégories : celles qui utilisent l'information de type statique associée aux valeurs et celle ne conservant qu'une information partielle sur la structure des valeurs.

La seconde approche est héritée des premières versions du langage LISP. Dans chaque mot mémoire un certain nombre de bits, appelés alors *bits de tag*, sont réservés pour discrimer les valeurs immédiates des adresses mémoire. La taille d'une zone mémoire est conservée explicitement dans la valeur, par exemple, dans un entête spécifique au bloc, ou à la page mémoire [Steele, 1977].

La première approche est similaire, dans l'idée, à celle utilisée dans la JVM. Le compilateur conserve suffisamment d'information du type statique pour distinguer dans la pile les valeurs appartenant à des types de base, des valeurs ayant un type objet. Pour les objets, la table des méthodes virtuelles associée à chaque objet sert de représentation dynamique du type et celle-ci peut être étendue pour permettre au GC de connaître le type des variables d'instance de l'objet [Ducournau, 2011].

Dans les langages fonctionnels de la famille ML, une donnée structurée appartenant à un type algébrique n'est pas directement liée au code le manipulant, comme un objet est lié à sa classe. Il n'y a pas alors de représentation triviale des types à l'exécution. De plus cette information de type statique n'est pas nécessaire pour exécuter correctement le programme.

Néanmoins, Appel [1989] propose le principe d'un gestionnaire de mémoire utilisant les informations de types et ne nécessitant pas de bit de *tag*. L'idée est alors de conserver, d'une manière ou d'une autre, le type des racines du graphe mémoire, puis d'utiliser cette information pour guider le parcours du graphe mémoire. Par exemple, pour effectuer ce parcours en présence d'une valeur d'un type algébrique, on vérifie dans un premier temps que le constructeur de données correspond à l'un des cas possibles pour le type attendu ; puis on calcule à partir de la définition du type le type attendu pour les différents champs du constructeur ; et on continue le parcours avec ces types.

Dans un modèle de compilation du polymorphisme paramétrique par partage de code la principale difficulté est de conserver le type des valeurs contenues dans la pile. En ef-

fet, le type des valeurs contenues dans le bloc d'activation d'une fonction polymorphe dépend de son contexte d'appel. Pour retrouver le type précis de ces valeurs plusieurs techniques ont été proposées : Appel [1989], Goldberg et Gloger [1992] et Aditya *et al.* [1994] explorent la pile d'appel pour reconstruire l'information de type manquante ; Tolmach [1994] représente explicitement les paramètres de type des fonctions polymorphe, à la manière du système F.

Sérialisation des types Dans ces modèles de gestion mémoire permettant de retrouver lorsque c'est nécessaire le type associé à une valeur, il est alors possible de sérialiser conjointement une valeur et son type, puis lors de la désérialisation de comparer le type reçu avec le type attendu. Furuse et Weis [2000]; Billings *et al.* [2006] ont étudié cette idée dans le cadre du langage OCaml. La principale difficulté qu'ils pointent vient de la représentation externe des types, en particulier des types abstraits par le système de modules : lors de l'écriture d'un GC, les types utilisés sont spécifiques au programme en cours d'exécution. Dans un mécanisme de communication, il faut que ces noms aient un sens global. Pour cela, les premiers introduisent un système de nommage des types basé sur un mécanisme de hachage de la définition du type ; les seconds étendent ce système de nommage aux types abstraits, en étendant la mécanisme de hachage à l'arbre de syntaxe du module définissant un type abstrait et à celui des modules dont il dépend.

1.2 Conserver partiellement les types statiques à l'exécution

Cette section regroupe deux compilateurs de langages typés statiquement ne conservant que partiellement l'information de typage statique dans le code généré et pour lesquels la sûreté d'exécution des primitives de communication repose sur quelques vérifications dynamiques de typage. Le première est le compilateur actuel du langage Java. Le second est le compilateur du langage Typed Scheme [Tobin-Hochstadt et Felleisen, 2008]. Ce langage ajoute un système de types statique au langage Scheme [Sperber *et al.*, 2009] usuellement typé dynamiquement. Il est possible de faire inter-opérer des programmes écrits dans les deux modes de typage. Lorsqu'un fragment de programme typé statiquement reçoit une valeur provenant d'un fragment typé dynamiquement, il est alors nécessaire de vérifier que la valeur reçue est compatible avec le type statique attendu. Cette problématique est identique à celle de la vérification de type lors de la désérialisation.

1.2.1 La sérialisation en Java 5

Depuis la version 5 du langage Java, la compilation du polymorphisme paramétrique est réalisée à l'aide d'un mécanisme de partage de code par effacement du paramètre de type (Bracha *et al.* [1998]). Ce choix interdit toutes les opérations basées sur le type dynamique d'un paramètre de type. Par exemple, il est impossible d'instancier un nouvel objet à partir d'un paramètre de type : `new A()` où `A` est un paramètre de type. Ou encore, il est impossible d'effectuer un *downcast* vers une classe paramétrée. La

sûreté du mécanisme de désérialisation reposant en partie sur le mécanisme de *downcast*, celle-ci semble à première vue mis à mal.

Néanmoins, une des contraintes fortes lors de l'ajout du polymorphisme paramétrique au langage Java était de conserver la possibilité d'inter-opérer avec du code écrit dans les versions précédentes du langage. En particulier, il est utile de permettre à du code ancien d'utiliser les nouvelles versions des classes de la bibliothèque de base du langage, sans pour autant distribuer, et maintenir à jour, deux versions de cette bibliothèque : l'une sans paramètre de type, l'autre avec paramètre de type. Les modifications apportées au compilateur Java pour satisfaire cette contrainte vont conjointement permettre de conserver la sûreté d'exécution lors de la désérialisation d'un objet appartenant à une classe paramétrée.

Le choix de ne pas modifier la JVM et d'utiliser une technique de compilation d'effacement du paramètre de type permet facilement à du code ancien, c'est-à-dire sans polymorphisme paramétrique, d'utiliser des objets dont la classe d'origine est paramétrée. Par exemple, dans le cas d'un conteneur générique comme la classe `Stack<E>` représentant une pile contenant des objets de type `E` — c'est-à-dire dont les méthodes d'empilement et de dépilement ont respectivement les signatures `void push(E)` et `E pop()` — le code produit par le compilateur sera une classe `Stack`, appelée *classe de base* de la classe `Stack<E>`, et dont les méthodes auront les signatures : `void push(Object)` et `Object pop()`. Du point de vue de la JVM, rien ne s'oppose alors à ce qu'un objet alloué par un code récent puisse être manipulé par un code ancien et réciproquement : dans les deux cas, la classe manipulée par la JVM sera la classe de base. Ce mécanisme permet de ne maintenir qu'une version « paramétrée » de la bibliothèque de base du langage. Le mécanisme d'effacement permet alors à du code ancien d'utiliser une version récente de la bibliothèque de base.

Pour assurer une inter-opérabilité complète entre les deux versions du langage, il est aussi nécessaire qu'un objet puisse être partagé entre des fragments de code anciens et récents. Si nous venons de voir que cette inter-opérabilité est techniquement possible du point de vue de la JVM, elle pose quelques difficultés supplémentaires au mécanisme de typage. Par exemple, un objet de type `Stack<Integer>` sera vu par du code ancien comme ayant le type `Stack` ; la signature de la méthode d'empilement étant alors `void push(Object)` rien ne s'oppose à ce que du code ancien empile des objets incompatibles avec le type `Integer`, cassant ainsi un des invariants attendus par le fragment de code récent.

Pour garantir la sûreté d'exécution d'un code récent inter-opérant avec du code ancien, le compilateur Java va insérer un *downcast* à chaque utilisation d'un paramètre de type. Par exemple, lors d'un appel à la méthode `E pop()` sur un objet de type statique `Stack<Integer>`, le compilateur insère un test dynamique pour vérifier que l'objet renvoyé par la méthode `pop` appartient effectivement au type `Integer`. Dans un programme écrit entièrement en Java avec polymorphisme paramétrique, ce test réussira systématiquement. Dans le cas d'interaction entre du code récent et du code plus ancien, ce test permettra de détecter lors d'une opération d'accès à l'objet de type `Stack<Integer>` par le code récent, une erreur qui ne pouvait pas être détectée par

l'opération d'écriture effectuée par le code ancien dans le même objet mais vu avec le type `Stack`.

Ce mécanisme permet également de conserver la sûreté d'exécution du mécanisme de désérialisation en présence de types paramétrés. Pour cela le *downcast* vers le type paramétré attendu, inséré immédiatement après la fonction de désérialisation, détectera dans un premier temps une incohérence entre la classe de base attendue et celle reçue, mais ne détectera pas d'incohérence sur les paramètres de type. Celles-ci seront détectées ultérieurement, de la même manière qu'au paragraphe précédent. En ce sens, cette approche revient à ajouter une dose supplémentaire de typage dynamique dans le langage.

1.2.2 Inter-opérabilité entre Scheme et Typed Scheme

Le langage Scheme [Sperber *et al.*, 2009] est un des dialectes actuel de LISP : c'est donc un langage fonctionnel à typage dynamique, mais où la liaison des variables est statique par défaut. Typed Scheme [Tobin-Hochstadt et Felleisen, 2008] est une extension statiquement typée de Scheme. Elle a été conçue pour permettre la conversion progressive de *scripts* écrits en Scheme vers des *programmes* typés statiquement et structurés à l'aide de modules [Tobin-Hochstadt et Felleisen, 2006]. Afin d'éviter une réécriture complète, la démarche proposée consiste à effectuer une migration progressive du code par ajout d'annotations de types dans le code original. Classiquement, ce système de types permet de manipuler les types de base usuels, des types enregistrements, des types fonctionnels, du polymorphisme paramétrique. De plus, pour permettre de typer les programmes Scheme basés sur les prédicats de typage dynamique, ce système comprend des *vrais* types unions[3] et permet de lier un prédicat de vérification dynamique de type à un type statique. Ces deux mécanismes permettent par exemple de typer la conditionnelle suivante avec le type `Number` :

```
(if (number? x)
  x
  (car x))
```

dans un contexte où la variable `x` est associée à un type union regroupant les types : `Number` ; `(Pair Number Number)` et `(Pair Number Char)`. Le prédicat de vérification dynamique de type `number?` étant lié au type statique `Number`, le type de `x` peut être restreint au singleton `Number` dans la première branche de la conditionnelle. Le type de `x` est alors restreint à son complémentaire dans la seconde branche, c'est-à-dire le type union `(Pair Number Char)` et `(Pair Number Number)`.

Dans le schéma de migration entre script et programme proposé par Tobin-Hochstadt et Felleisen [2006], la distinction entre code typé dynamiquement et code typé statiquement se fait module par module et il est nécessaire de faire inter-opérer des fragments de code écrit dans les deux modes de typage. Notamment un module typé dynamiquement doit pouvoir faire appel à une fonction définie dans un module typé statiquement. Pour

3. pour reprendre le vocabulaire des auteurs, par opposition aux types unions *discriminés* de ML.

garantir la sûreté d'exécution de cette fonction typée statiquement, il faut s'assurer que chacune de ses utilisations depuis un contexte typé dynamiquement sera respectueuse du type des arguments. Réciproquement, lors de l'utilisation dans un contexte typé statiquement d'une fonction typée dynamiquement, il faut s'assurer de la compatibilité de la valeur renvoyée avec le type attendu. Pour permettre ce test, un module typé statiquement doit expliciter le type supposé des fonctions dynamiques qu'il utilise[4].

Ces vérifications lors de la communication entre les deux modes de typage reposent sur les primitives de typage dynamique de Scheme et sur le mécanisme de *contrat* d'ordre supérieur décrit par Findler et Felleisen [2002]. L'une des difficulté est la vérification d'appartenance d'une argument fonctionnel typé dynamiquement à un type fonctionnel. Ce test ne peut être réalisé par une simple analyse de la valeur fonctionnelle, car cela est équivalent à typer statiquement la fonction originale. Pour résoudre cette difficulté, la vérification doit être reportée au moment de l'application effective de la fonction et doit être effectuée à chaque exécution de cette application. Pour réaliser ce mécanisme, une fonction typée dynamiquement et passée en argument à une fonction typée statiquement, est encapsulée dans une fonction vérifiant la compatibilité de la valeur renvoyée avec le type attendu ; et réciproquement lors de l'application d'une fonction typée statiquement à une fonction typée dynamiquement, celle-ci est encapsulée dans une fonction vérifiant la compatibilité des arguments effectifs.

Dans ce modèle, le partage d'une donnée modifiable entre les deux modes de typage est plus simple que dans le modèle d'inter-opérabilité de Java 5. En effet, la modification d'une valeur est systématiquement encapsulée dans une fonction de modification. La modification incorrecte d'une valeur typée statiquement par un fragment de code typé dynamiquement est donc interdite par le mécanisme plus général d'appel de fonction.

Si le système de types de Typed Scheme permet le polymorphisme paramétrique, la version actuelle du compilateur ne permet pas de vérifier la compatibilité d'une valeur avec un type paramétré, il est alors interdit de manipuler depuis un code typé dynamiquement une valeur dont le type statique est paramétré. Cette limitation impose la réécriture d'une version typée de la bibliothèque de base du langage.

1.3 Fonctions spécifiques de sérialisation

La première partie de ce chapitre décrit quelques systèmes exploitant les informations de type statique conservées par l'environnement d'exécution pour typer le mécanisme de désérialisation. Dans ces systèmes les fonctions de sérialisation et de désérialisation sont des fonctions génériques basées sur un parcours du graphe mémoire. Ces fonctions sont rarement écrites directement dans le langage de programmation qui va les utiliser, mais dans le langage de programmation servant à écrire l'environnement d'exécution. Ce second langage est rarement un langage avec une discipline stricte de typage statique, mais plus souvent un langage avec *coercion* de type non sûre et permettant de manipuler explicitement la mémoire, tel que langage C.

4. Dans les premières versions, le compilateur de Typed Scheme essayait de déterminer automatiquement le type des fonctions dynamiques utilisées par un module typé statiquement.

Cette section détaille d'autres approches visant à décrire directement dans le langage hôte les fonctions de sérialisation ou au moins à typer statiquement le langage servant de support à l'écriture de l'environnement d'exécution. Les deux premières parties décrivent des mécanismes basés sur l'écriture de fonctions spécifiques pour chacun des types de données sérialisables, la seconde partie discutant en particulier sur la génération automatique de ces fonctions. La troisième partie présente des systèmes de types statiques permettant de manipuler explicitement l'information de typage.

1.3.1 Approche *ad hoc*

À la place d'une seule paire de fonctions génériques de (dé)sérialisation, la technique décrite ici repose sur des fonctions spécifiques pour chaque type de données. L'environnement d'exécution ne contient alors qu'un ensemble restreint de primitives de (dé)sérialisation opérant sur les types de base et les fonctions spécifiques aux types de données d'un programme sont ensuite écrites par le programmeur à partir de ces primitives.

Cette approche permet un plus grand contrôle par le programmeur de la représentation externe de ses données, celle-ci n'étant plus directement liée à leur représentation interne. En particulier, quelque soit le système de types du langage utilisé, il n'est pas nécessaire d'ajouter d'information de type dans la représentation externe : les fonctions de désérialisation étant directement écrites dans le langage de programmation hôte, son système de types garantit que les valeurs qu'elles construisent sont cohérentes avec leur type de retour. Si les données lues par une fonction de désérialisation ne permettent pas de reconstruire une telle valeur, la fonction peut échouer sans risque pour la sûreté d'exécution du programme. Néanmoins, même si aucune information n'est nécessaire dans la représentation externe, en conserver un petit peu peut permettre une gestion un peu plus documentée des cas d'erreurs et éviter certaines situations sans risque pour la sûreté d'exécution mais tout de même incorrectes, par exemple lorsque deux valeurs dont les types sont distincts partagent la même représentation mémoire.

L'indépendance entre les représentations internes et externes est nécessaire lors de communication entre programmes écrits dans des langages différents et utilisant des représentations internes différentes. Ces mécanismes de (dé)sérialisation *ad hoc* se retrouvent donc dans des systèmes d'appels de fonctions à distance tels que XDR/RPC [RFC 5531, 2009; RFC 4506, 2006], CORBA [Object Management Group, 1995], etc. Les types de données manipulées dans ces systèmes sont souvent monomorphes, mais l'approche *ad hoc* peut aussi s'appliquer aux systèmes de types avec polymorphisme paramétrique. Il faut pour cela que le langage de programmation permette de paramétrer la fonction de (dé)sérialisation d'un type paramétré, par les fonctions de (dé)sérialisation de ses paramètres de type.

L'approche *ad hoc* peut se révéler répétitive dans son écriture et peut être une source supplémentaire d'erreurs de programmation. Pour rendre cette technique un peu plus générique, les fonctions de (dé)sérialisation associées à un type peuvent parfois être générées à partir de la définition d'un type ou d'une description dans un *langage de*

description d'interface (IDL, comme acronyme anglophone), comme par exemple dans CORBA. La section suivante s'attarde sur un mécanisme d'aide à l'écriture de fonctions de sérialisation spécifiques, conçu à l'origine pour être utilisé dans le compilateur SML.NET Benton *et al.* [2006]. Cette technique est applicable dans la plupart des langages de programmation fonctionnelle.

1.3.2 Combinateurs de sérialisation

Pour faciliter l'écriture de fonctions spécifiques de (dé)sérialisation, Kennedy [2004] et Elsman [2005] proposent, respectivement dans le cadre des langages fonctionnels Haskell et SML, un ensemble de primitives, pour les types de base, et un ensemble de combinateurs, pour les types structurés, permettant de définir conjointement et succinctement les fonctions de sérialisation et de désérialisation spécifiques à un type. Ils fournissent, par exemple, un combinateur permettant de définir les fonctions de (dé)sérialisation d'une liste à partir des fonctions de (dé)sérialisation des éléments de la liste. Ils fournissent aussi des combinateurs facilitant l'écriture de telles fonctions pour les types sommes et enregistrements.

Construire les fonctions de sérialisation et de désérialisation conjointement évite les incohérences entre les deux mécanismes et garantit que toute valeur sérialisée sera désérialisable vers une valeur « équivalente » à la valeur sérialisée. Un autre avantage pour le programmeur qui se restreint à l'ensemble des primitives et combinateurs fournis pour construire ses fonctions de (dé)sérialisation *ad hoc* est qu'il suffit alors de changer l'implémentation des combinateurs pour changer de format de sortie. Il est inutile de réécrire les fonctions spécifiques écrites à partir de ces combinateurs.

En plus des combinateurs permettant d'écrire des fonctions de (dé)sérialisation pour les types structurés et les types récursifs, Kennedy [2004] et Elsman [2005] proposent chacun des combinateurs permettant de prendre en compte une certaine forme de *partage* dans les données sérialisées. Ils permettent typiquement d'éviter les répétitions dans la représentation externe, et d'en réduire la taille. Néanmoins, leurs notions de partage sont différentes. Le premier demande au programmeur de préciser pour chaque type, la notion d'égalité entre valeurs à utiliser lors de la sérialisation, le second se base sur une notion d'égalité polymorphe de SML. De la même manière, ils diffèrent aussi dans la technique utilisée pour mémoriser dans un *dictionnaire* les valeurs déjà sérialisées : le premier demande au programmeur de fournir pour chaque type, la structure de données servant de dictionnaire, alors que le second fournit un mécanisme générique basé sur du typage dynamique (voir section 1.4.1) et optimisé à l'aide d'un mécanisme de *hash-consing*. De plus, le mécanisme proposé par Elsmann permet de gérer la sérialisation des références de SML.

Sutou et Sumii [2007] étendent l'ensemble de combinateurs de sérialisation avec un combinateur permettant de gérer les abstractions de types construites par un système de modules. Ce combinateur utilise un mécanisme de chiffrement : à chaque type abstrait est associé une clé servant à chiffrer le résultat de la sérialisation de la représentation concrète du type.

Si ces combinateurs permettent au programmeur d'avoir le contrôle sur la représentation externe des données qu'il manipule, il est aussi possible dans de nombreux cas de définir de manière systématique une représentation externe et de *dériver* les fonctions de (dé)sérialisation à partir de la définition d'un type. Dans le cadre du langage Haskell, cette génération automatique des fonctions *ad hoc* peut être couplée au mécanisme de classes de types pour surcharger le nom des fonctions spécifiques et retrouver l'impression d'un unique couple de fonctions de (dé)sérialisation.

Cette approche à base de combinateurs peut s'appliquer dans d'autres langages de programmation que les deux langages décrits ici. Néanmoins cette approche ne semble pas s'étendre à la sérialisation de données associées à du code, telles que des fermetures ou des objets. Par exemple, cette distinction se retrouve explicitement dans le langage Scala [Odersky, 2009]. Scala est un langage mélangeant programmation par objets et fonctionnelle qui est compilé vers la machine virtuelle Java. Il contient deux mécanismes de sérialisation distincts. Le premier est une réutilisation directe du mécanisme générique apporté par la JVM. Il permet la sérialisation des objets. Le second mécanisme de sérialisation permet simplement la sérialisation de données structurées, indépendamment du code qui les manipule. Il est basé sur les combinateurs décrits dans cette section.

1.3.3 Manipuler explicitement les types

Le mécanisme de compilation du polymorphisme paramétrique par partage de code et représentation uniforme des données permet de générer un code compact. Le choix de ce mécanisme simplifie aussi l'écriture d'un compilateur : le générateur de code est relativement indépendant du mécanisme de typage et, par exemple, il est possible de faire évoluer ce dernier sans trop de conséquences sur les étapes suivantes de la compilation.

Cette approche peut néanmoins avoir un coût à l'exécution, notamment à cause des indirections nécessaires lorsqu'une donnée ne peut pas être représentée dans un mot-mémoire, ou par un gaspillage de l'espace mémoire : par défaut un booléen est représenté à l'aide d'un mot-mémoire et un tableau de booléens est représenté par une séquence de mots-mémoire au lieu d'une séquence de bits.

Pour notamment faciliter l'écriture de compilateurs utilisant une représentation spécialisée de la mémoire, Harper et Morrisett [1995] proposent un langage intermédiaire permettant de manipuler explicitement les types. Pour cela, ils introduisent dans ce langage une construction `typerec`, permettant une analyse récursive et par cas sur la structure des types, notamment pour des types algébriques et des types fonctionnels. Ils définissent ensuite un système de types pour ce langage intermédiaire, garantissant sa bonne évaluation.

Ces travaux ont été utilisés pour l'écriture de plusieurs compilateurs pour le langage SML : les projets TIL [Tarditi *et al.*, 1996, 2004] et MLton [Weeks *et al.*, 2007] utilisent ces informations de type pour diverses optimisations dans le code généré. Le premier projet utilise une représentation spécialisée des données, mais compile tout de même le polymorphisme paramétrique dans un modèle de partage de code : lorsque l'information statique de type est insuffisante, le code généré utilise l'information dynamique de type pour connaître la taille des données qu'il manipule, de la même manière que les

gestionnaires de mémoire présentés à la section 1.1.3.

Weirich [2002] utilise l'idée de cette construction `typerec` pour définir un langage de programmation dit *polytypic*, c'est-à-dire pouvant manipuler et analyser explicitement des expressions de type durant l'exécution. Weirich [2006] a ensuite montré comment réaliser cette construction dans un langage utilisant un mécanisme de compilation ne préservant pas – implicitement – les informations de type. Pour représenter explicitement dans le programme source les expressions de type à conserver, elle introduit, à la manière de Yang [1998] et Danvy [1998], un type singleton $\texttt{Ty}(\tau)$, dont l'unique valeur est une représentation du type τ [5].

L'exemple canonique utilisé par Weirich [2006] pour illustrer la construction `typerec` est une fonction générique de sérialisation, décrite par analyse récursive de la structure du type de la valeur à sérialiser. De la même manière, il est possible de décrire une fonction générique de désérialisation. Comme les fonctions de (dé)sérialisation *ad hoc* présentées précédemment, cette fonction générique est entièrement typée dans le langage de programmation hôte et préserve les garanties apportées par le système de types.

De plus, le langage de type considéré par Weirich [2006] permet de représenter les types existentiels. Il est possible alors d'y encoder les types abstraits et de définir un mécanisme de sérialisation d'une valeur d'un type abstrait en retrouvant son type concret. Il est aussi imaginable d'y appliquer ensuite un traitement particulier, comme par exemple le mécanisme de chiffrement proposé par Sutou et Sumii [2007] et décrit à la section précédente.

1.4 La sérialisation en OCaml

La version actuelle du compilateur OCaml [Leroy *et al.*, 2010] utilise une *représentation uniforme* des données et un mécanisme de partage de code pour réaliser efficacement le mécanisme de polymorphisme paramétrique du langage. Pour la bonne exécution du gestionnaire de mémoire les données sont *taggées* à l'aide d'un bit permettant de discriminer les adresses-mémoire des valeurs immédiates. Les valeurs allouées sur le tas sont précédées d'un entête décrivant notamment la taille de la zone mémoire. Ces entêtes contiennent un autre *tag*, représenté par un petit entier, permettant de discriminer quelques bloc mémoire spécialisés (flottant, chaîne de caractères, fermeture, ...) ou de discriminer les différents constructeurs d'un type somme.

Ces informations sur la structure d'une valeur en mémoire permettent aussi de réaliser une fonction générique de sérialisation à l'aide d'un simple parcours de leur graphe mémoire. De la même façon, il est possible d'écrire une fonction de désérialisation qui reconstruit le graphe mémoire à partir de sa représentation sérialisée. Dans sa version de base, cette fonction générique de désérialisation ne vérifie pas la compatibilité de la valeur lue avec le type attendu et il est possible de mettre en erreur un programme accepté par

5. Plus précisément, dans le prototype Haskell décrit par Weirich [2006], le type `Ty(_)` est nommé `R` et doit aussi être paramétré par le type de retour de la construction `typerec :: Theta c -> R c a -> c a` où `Theta c` est un enregistrement représentant les différents cas d'analyse du type.

le typeur. La figure 1.1 présente un exemple d'utilisation incorrecte du mécanisme de sérialisation.

FIGURE 1.1 Désérialisation non sûre

Le module `Marshal` de la bibliothèque de base distribué avec le compilateur OCaml exporte des fonctions de sérialisation et de désérialisation. Leur type s'apparente aux types suivants :

```
val to_string: 'a -> string
val from_string: string -> 'a
```

Ainsi, il est possible d'écrire une fonction `unsafe_copy`, dupliquant la représentation mémoire d'une valeur et associant un type statique arbitraire à la nouvelle valeur :

```
val unsafe_copy: 'a -> 'b
let unsafe_copy x = from_string (to_string x)
```

Le programme suivant est accepté par le typeur, mais provoquera un accès mémoire invalide à l'exécution :

```
let tab : int array = unsafe_copy 0 in
print_int tab.(0)
```

En effet, l'entier `0` n'est pas une représentation acceptable pour un tableau et l'accès à la première case de ce tableau va tenter de déréférencer cet entier ne correspondant pas à une adresse mémoire valide.

La suite de cette section regroupe en trois catégories les différentes approches recensées pour typer le mécanisme de sérialisation dans le cadre du compilateur OCaml. La première se base sur l'ajout de quelques constructions de typage dynamique au langage. La seconde se base la génération de fonctions *ad hoc* à l'aide de CamlP4 et la dernière se base sur une notion de compatibilité d'une représentation mémoire avec un type donné.

1.4.1 Type dynamique

Leroy et Mauny [1993] et Abadi *et al.* [1995] ont étudié l'ajout de primitives de typage dynamique dans les langages typés statiquement en présence de polymorphisme paramétrique. Leur approche consiste à étendre le langage avec un type `dyn` – permettant d'encapsuler dans un couple une valeur et son type statique – et de permettre l'analyse d'une valeur de type `dyn` par cas selon le type encapsulé.

Pour permettre l'ajout de ce mécanisme dans un compilateur procédant par effacement de l'information de type, la fonction de construction d'une valeur de type `dyn` reçoit explicitement en paramètre le schéma de type clos de la valeur à encapsuler.

Il est possible d'utiliser ce mécanisme pour réaliser un mécanisme sûr de désérialisation, en restreignant le type statique des valeurs sérialisées au type `dyn`, puis en analysant le type encapsulé dans la valeur désérialisée. La figure 1.2 montre un exemple d'utilisation.

Dans la mise en œuvre du mécanisme proposé par Leroy et Mauny [1993], les noms des types sont représentés durant l'exécution par un entier calculé au moment de la

compilation du programme. Cet entier est spécifique à un exécutable donné et peut être différent dans une version différente du même programme. Ainsi, ce mécanisme ne permet la sérialisation sûre qu'entre différentes exécutions d'un même programme. Pour permettre un mécanisme plus général, il faut alors envisager un mécanisme global de nommage des types, comme celui proposé par Billings *et al.* [2006].

1.4.2 Fonctions *ad hoc*

Suivant le principe décrit dans la section 1.3, il est possible de générer automatiquement des fonctions de sérialisation spécifiques pour les types de données algébriques. Sutou et Sumii [2007] utilisent pour cela le pré-processeur CamlP4 [De Rauglaudre, 2003] et repose sur des combinateurs similaires à ceux décrits précédemment (section 1.3.2). Yallop [2007] utilise un principe similaire pour permettre un mécanisme de communication sûre entre un serveur et un client non fiable.

1.4.3 Compatibilité de la représentation mémoire

De la même façon que les gestionnaires de mémoire guidés par les types (cf. la section 1.1.3) il est possible de parcourir la représentation mémoire d'une valeur pour vérifier sa compatibilité avec un type donné. Ce type de parcours en parallèle d'un graphe mémoire et de son type est déjà employé dans le compilateur OCaml, par exemple, pour réaliser la fonction générique d'affichage de la boucle d'interaction. Dans cette fonction générique, le type de la valeur permet de retrouver le nom d'un constructeur ou le nom des champs d'un enregistrement [6].

Ce type de parcours est aussi utilisé dans le cadre du logiciel Frama-C [Cuoq et Signoles, 2009]. Ses auteurs réimplémentent la fonction générique de désérialisation

6. Il est à noter que ce type de parcours ne peut être écrit en OCaml à l'aide d'un programme correctement typé.

Figure 1.2 Désérialisation et type dynamique

Le type des fonctions de (dé)sérialisation soit être restreint au valeur de type `dyn` :

```
val to_string: dyn -> string
val from_string: string -> dyn

val safe_copy: dyn -> dyn
let safe_copy x = from_string (to_string x)
```

Le programme proposé dans l'exemple 1.1 doit être adapté pour ajouter une analyse explicite du type de la valeur désérialisée :

```
let dyn_tab : dyn = safe_copy (dynamic (0 : int)) in
let tab = match dyn_tab with
  | dynamic (tab: int array) -> tab
  | _ -> failwith "Unexpected type" in
print_int tab.(0)
```

d'OCaml pour permettre d'appliquer des pré-traitements en cours de la désérialisation, notamment pour ne pas dupliquer de valeur en mémoire et substituer une valeur désérialisée à la valeur déjà existante en mémoire. Pour cela l'algorithme est guidé par une description du type de la valeur et les traitements éventuels à effectuer pour chacun des types.

1.5 Propositions

Le mécanisme de parcours du graphe mémoire d'une valeur pour vérifier sa compatibilité avec un type, décrit à la section 1.4.3 permet sans changer le mécanisme générique de (dé)sérialisation d'OCaml d'obtenir un mécanisme sûre de désérialisation : il suffit de vérifier *a posteriori* la compatibilité d'une valeur désérialisée avec le type attendu. Cette vérification est triviale en présence des types de base usuels et de types algébriques. L'objet initial de cette thèse est de voir sous quelles conditions cette notion de compatibilité peut s'étendre à l'ensemble du noyau fonctionnel et impératif du langage OCaml tout en conservant une efficacité raisonnable. La suite de cette section détaille brièvement la problématique et l'approche choisie successivement en présence de partage et de cycles dans la représentation mémoire, de valeurs modifiables, puis de fermetures.

Partage et valeurs modifiables Dans un modèle de programmation fonctionnelle où les structures de données sont non modifiables, il est important pour obtenir des programmes efficaces de ne pas dupliquer inutilement la représentation mémoire d'une valeur. Lorsque ce modèle est étendu avec des données modifiables, cette notion de partage devient indispensable : il devient sémantiquement incorrect de dupliquer une valeur modifiable. Pour ces deux raisons, le mécanisme de sérialisation d'OCaml permet de conserver le partage présent dans une représentation mémoire.

En l'absence de valeurs modifiables, il semble possible d'ignorer le partage lors du parcours de vérification de compatibilité d'un graphe mémoire avec un type. Mais cette approche a dans le pire cas une complexité exponentielle en la taille de la représentation mémoire.

En présence de valeurs modifiables, il devient indispensable de prendre en compte le partage : le système de types impose aux valeurs modifiables un type monomorphe [Leroy, 1992] ; ignorer le partage lors de la désérialisation risque de rompre cet invariant, en associant des types distincts aux différentes occurrences d'une donnée modifiable initialement partagée.

Une possibilité pour retrouver un parcours linéaire et contraindre l'utilisation de type monomorphe pour les valeurs modifiables est de conserver le type avec lequel un nœud du graphe mémoire a déjà été vérifié. Si ce nœud est partagé, il suffira la seconde fois de comparer le type attendu avec le type ayant servi à parcourir la valeur la première fois. Cependant cette approche impose des types monomorphes à chacune des valeurs partagées, y compris pour les valeurs non modifiables. En présence de polymorphisme, cette approche n'est pas suffisante : une valeur non modifiable peut avoir plusieurs types (voir l'exemple 1.5.1).

Exemple 1.5.1. *Soit le type d'arbre binaire suivant paramétré le type des étiquettes associées aux nœuds et dont les feuilles sont étiquetées par des chaînes de caractères :*

```
type 'a t =
  | Leaf of string
  | Node of 'a t * 'a * 'a t
```

Une même feuille peut être partagée entre un arbre dont les nœuds sont étiquetés par des entiers et par un arbre dont les nœuds sont étiquetés par des flottants. Lors d'un parcours de vérification de type, il est alors nécessaire de vérifier que la représentation mémoire de cette feuille est compatible avec ces deux types.

L'approche proposée dans cette thèse consiste en un parcours topologique du graphe mémoire, qui permet de collecter l'ensemble des types attendus pour une valeur et de les anti-unifier [Plotkin, 1970; Huet, 1976] avant de la vérifier. Ce choix permet, dans un premier temps en l'absence de fermeture et de type cyclique, de pouvoir typer correctement la désérialisation de toutes les valeurs sérialisables. Une partie de cette proposition a déjà été étudiée dans [Henry *et al.*, 2007].

Fonctions Le mécanisme de sérialisation d'OCaml permet la sérialisation de données fonctionnelles. La fermeture représentant une telle donnée est sérialisée directement comme un couple contenant un pointeur de code et un environnement d'évaluation. Pour que le pointeur de code ait une signification dans le programme effectuant la désérialisation, celle-ci ne peut actuellement avoir lieu que dans une instance du même exécutable. Cependant, le mécanisme de vérification proposé ici est indépendant de la représentation externe des fermetures et il resterait vraisemblablement utilisable avec un mécanisme de désérialisation utilisant de la liaison dynamique ou de chargement dynamiquement de code.

Dans le mécanisme actuel de sérialisation d'OCaml, le nombre de pointeurs de code pouvant apparaître dans une fermeture pour un exécutable donné est fixe. Le type du code ayant servi à la création de chacun de ces pointeurs de code est une information statique, qui peut être conservée à l'exécution sans introduire de surcoût autre qu'une légère augmentation de l'empreinte mémoire initiale. Cette information peut être utilisée par le mécanisme de désérialisation pour comparer le type attendu pour une fermeture, avec le type associé par le compilateur au pointeur de code désérialisé. Si le type associé au code est monomorphe et si l'on suppose que la valeur désérialisée n'a pas été forgée par un attaquant malveillant, cette vérification est suffisante. Si l'on ne fait pas confiance à la valeur désérialisée, il faut aussi vérifier que l'environnement de la fermeture est compatible avec l'environnement de typage associé au pointeur de code.

Si le type associé au pointeur de code d'une fermeture est polymorphe, il est aussi nécessaire de vérifier que le type original de la fonction sérialisée et le type attendu soient la même instanciation, ou une instanciation compatible, du type associé au pointeur de code. Or, dans un mécanisme de compilation ne préservant pas les types à l'exécution, l'instanciation exacte d'une fermeture n'est pas directement disponible. Goldberg et Gloger [1992] ont montré, dans le cadre d'un gestionnaire de mémoire, que cette instanciation peut être généralement retrouvée en explorant l'environnement de la fermeture.

Ces propositions vont être détaillées et formalisées dans les deux chapitres suivants.

Chapitre 2

Mini-ML et environnements d'exécution

Le chapitre précédent a décrit les grandes lignes d'un mécanisme de désérialisation sûre dans le cadre d'un langage à la ML, où les valeurs adoptent une représentation uniforme et *tagguée*. Ce chapitre présente une définition formelle des éléments utiles pour mettre en œuvre ce principe : le langage source et son système de types ; le mécanisme d'évaluation ; le langage des valeurs et son système de types. En particulier, nous verrons comment définir les règles de typage et d'évaluation des primitives de (dé)sérialisation. Ce chapitre se conclut par la démonstration de la correction du système de types par rapport au mécanisme d'évaluation.

2.1 Syntaxe

L'étude formelle se restreint dans un premier temps à un sous-ensemble du langage OCaml proche du langage mini-ML [Clément *et al.*, 1986] : un λ-calcul simplement typé, étendu avec des booléens, des entiers, des types algébriques (paires, listes, options), des références et des fonctions récursives. La syntaxe abstraite du langage de programmation étudié est définie à l'aide de la grammaire ci-dessous. On y distinguera quatre sous-ensembles de constructions syntaxiques : le noyau fonctionnel ; les traits impératifs ; les constructeurs de données avec les primitives et les structures de contrôle associées ; l'égalité polymorphe.

t ::=	x	*variable*
	$\mid \lambda x.t$	*fonction*
	$\mid \texttt{fix}\ f = \lambda\, x.t$	*fonction récursive*
	$\mid t\ t$	*application*
	$\mid \texttt{let}\ x = t\ \texttt{in}\ t$	*définition locale*
	$\mid \texttt{ref}(t)$	*allocation de référence*
	$\mid !(t)$	*déréférencement*
	$\mid t := t$	*affectation*
	$\mid t\,;\, t$	*séquence*
	$\mid \texttt{true} \mid \texttt{false}$	*booléen*
	$\mid i$	*entier*
	$\mid t\ \texttt{op}\ t$	*opérations entières et booléennes*
	$\mid \texttt{if}\ t\ \texttt{then}\ t\ \texttt{else}\ t$	*conditionnelle*
	$\mid (t,\, t)$	*constructeur de paires*
	$\mid \texttt{fst}(t) \mid \texttt{snd}(t)$	*projections*
	$\mid [] \mid t :: t$	*constructeurs de listes*
	$\mid (\texttt{case}\ t\ \texttt{of}\ [] \to t \mid x :: xs \to t)$	*filtrage de listes*
	$\mid \texttt{None} \mid \texttt{Some}(t)$	*constructeurs d'options*
	$\mid (\texttt{case}\ t\ \texttt{of None} \to t \mid \texttt{Some}(x) \to t)$	*filtrage d'options*
	$\mid t = t$	*égalité polymorphe*

L'ensemble des types algébriques choisi ici est arbitraire. Il contient un type produit (les paires), un type somme simple (les options) et un type somme récursif (les listes). Ces trois cas suffisent à illustrer l'ensemble des mécanismes nécessaires dans le cas général.

Exemple 2.1.1. *Une fonction calculant la longueur d'une liste peut s'écrire :*

$$\texttt{fix}\ \mathit{length} = \lambda\, \mathit{xs}.\texttt{case}\ \mathit{xs}\ \texttt{of}\ [] \to 0 \mid y :: \mathit{ys} \to 1 + \mathit{length}\ \mathit{ys}$$

2.2 Sémantique opérationnelle

Cette section reproduit une sémantique opérationnelle classique pour ce langage, dite sémantique à grand pas en appel par valeurs. Historiquement cette présentation a été étudiée par Clément, Despeyroux, Despeyroux, et Kahn [1986]. Classiquement, les *valeurs* résultant de l'*évaluation* d'un programme sont :

- soit des données de base ou structurées : entiers, booléens, listes de valeurs, paires de valeurs, valeurs optionnelles ;
- soit des *fermetures*, c'est-à-dire des fonctions auxquelles ont été explicitement attaché l'environnement dans lequel leur corps devra être évalué. Ces *environnements d'évaluation* capturés par les fermetures associent une valeur à chacune des variables libres de la fonction ;
- soit des *étiquettes*, représentant l'adresse mémoire d'une valeur modifiable.

Dans les descriptions usuelles de la sémantique opérationnelle de mini-ML, le langage de valeurs est un sous-ensemble du langage source, avec une notation particulière pour les fermetures. Dans le cadre de l'étude de la désérialisation, il sera plus pratique de définir un langage de valeurs plus proche de la représentation mémoire utilisée réellement.

2.2.1 Valeurs, environnement et tas

Le langage des valeurs est proche de la représentation mémoire utilisé par la version actuelle du compilateur OCaml. En plus des *entiers-mémoire* et des *blocs* utiles pour représenter les données simples et les données structurées, il permet de distinguer des fermetures et des étiquettes.

$$\begin{array}{lll} v & ::= & i & \textit{entier-mémoire} \\ & \mid & \texttt{Blk}(v_1,\, v_2) & \textit{bloc} \\ & \mid & \langle \lambda x.t, \rho \rangle & \textit{fermeture} \\ & \mid & \langle\!\langle f = \lambda x.t, \rho \rangle\!\rangle & \textit{fermeture récursive} \\ & \mid & \ell & \textit{étiquette} \end{array}$$

Les entiers-mémoire servent à représenter les constructeurs constants du langage : les entiers, les booléens, la liste vide et le constructeur `None`. Les blocs servent à représenter les constructeurs de paires et de listes et le constructeur `Some(_)`. Par exemple, la valeur résultant de l'évaluation de la liste $[1;2]$ sera représentée par $\texttt{Blk}(1,\ \texttt{Blk}(2,\ 0))$, où 0 représente la liste vide. Pour simplifier la présentation, tous les blocs seront de taille 2 ; ainsi la valeur résultant de l'évaluation du constructeur $\texttt{Some}(t)$ sera la même que

celle d'une liste à un élément, c'est-à-dire : $\texttt{Blk}(v, 0)$ où v est la valeur résultant de l'évaluation de t. Le second élément étant inutile, il est initialisée avec la valeur 0.

On distingue les fermetures simples des fermetures *récursives*, produites par l'évaluation d'une définition récursive $\texttt{fix}\ f = \lambda x.t$. L'environnement d'évaluation ρ capturé par les fermetures est défini comme une fonction partielle de l'ensemble des variables vers l'ensemble des valeurs. Les environnements d'évaluation seront parfois notés explicitement : $\{ x_1 \mapsto v_1 ; \ldots ; x_n \mapsto v_n \}$. Le domaine de définition d'un environnement est noté $dom(\rho)$. La notation $\rho_1 \oplus \rho_2$ représente un nouvel environnement tel que $dom(\rho_1 \oplus \rho_2) = dom(\rho_1) \bigcup dom(\rho_2)$ et :

$$(\rho_1 \oplus \rho_2)(x) = \begin{cases} \rho_2(x) & \text{si } x \in dom(\rho_2) \\ \rho_1(x) & \text{sinon} \end{cases}$$

Les étiquettes servent à représenter les références. Le contenu d'une référence est stocké dans un *tas*. Un tas μ est représenté par une fonction partielle de l'ensemble $\mathcal{L}$ des étiquettes vers l'ensemble des valeurs. L'allocation d'une valeur est notée $\mu \oplus \{\ell \mapsto v\}$ avec $\ell \notin dom(\mu)$. Dans cette formalisation, on suppose qu'il est toujours possible d'allouer une nouvelle valeur sur un tas ; pour cela, l'ensemble $\mathcal{L}$ des étiquettes est infini et le domaine d'un tas est toujours fini. La modification d'un tas est notée $\mu \oplus \{\ell \mapsto v\}$ avec $\ell \in dom(\mu)$.

2.2.2 Règles de dérivation

La valeur d'un programme est définie à l'aide d'un système de règles de dérivation dont les conclusions sont de la forme $\mu, \rho \vdash t \Rightarrow v/\mu'$ et signifient « dans l'environnement d'évaluation ρ, le programme t s'évalue vers la valeur v et transforme le tas μ en un tas μ'. » Plus formellement, ces règles servent à construire inductivement une relation entre des triplets (tas, environnement, terme) et des couples (valeur, tas). Cette relation est construite pour que l'évaluation soit définie de manière *déterministe* : pour un tas μ, un environnement ρ et un terme t donnés il existera au plus une valeur v et un tas μ' telle que $\mu, \rho \vdash t \Rightarrow v/\mu'$. La proposition 2.2.3 établira cette propriété. Lorsqu'il n'existera aucune valeur v telle que $\mu, \rho \vdash t \Rightarrow v/\mu'$ on notera $\mu, \rho \vdash t \not\Rightarrow$. La section 2.2.3 reviendra sur ces cas particuliers.

Noyau fonctionnel La plus simple des règles de dérivation est celle des variables. Elle se lit : « si la variable x appartient au domaine de l'environnement d'évaluation ρ, alors le programme x s'évalue vers la valeur $\rho(x)$. Le tas μ n'est pas modifié. »

$$\frac{\begin{array}{c}[\textsc{Eval-Var}] \\ x \in dom(\rho)\end{array}}{\mu, \rho \vdash x \Rightarrow \rho(x)/\mu}$$

Par opposition, si la variable x n'appartient par au domaine de ρ, le programme x ne peut pas s'évaluer.

Le résultat de l'évaluation d'une fonction est une fermeture capturant l'environnement courant. une fermeture ne sera ré-ouverte que par la règle d'évaluation de l'application. Dans un compilateur réaliste, le domaine de l'environnement capturé par une fermeture est restreint au minimum utile, c'est-à-dire à l'ensemble des variables libres du corps de la fonction ; pour simplifier, dans cette présentation l'environnement complet est capturé.

$$\frac{}{\mu, \rho \vdash \lambda x.t \Rightarrow \langle \lambda x.t, \rho \rangle / \mu} \text{[Eval-Clos]}$$

Une application $t_1\ t_2$ s'évalue uniquement dans les cas où dans l'environnement courant t_1 s'évalue vers une fermeture $\langle \lambda x.t', \rho' \rangle$ et où ensuite t_2 s'évalue vers une valeur v'. Le résultat de cette application est alors le même que celui de l'évaluation du corps t' de la fonction, dans son environnement de définition original — c'est-à-dire celui capturé dans la fermeture — étendu par la valeur v' associée à son argument x. Si l'une de ces prémisses ne peut être établie, l'application ne peut pas s'évaluer.

$$\frac{\mu, \rho \vdash t_1 \Rightarrow \langle \lambda x.t', \rho' \rangle / \mu' \qquad \mu', \rho \vdash t_2 \Rightarrow v_2 / \mu'' \qquad \mu'', \rho' \oplus \{x \mapsto v_2\} \vdash t' \Rightarrow v / \mu'''}{\mu, \rho \vdash t_1\ t_2 \Rightarrow v / \mu'''} \text{[Eval-App]}$$

Cette règle d'évaluation ne modifie pas explicitement le tas. Néanmoins l'évaluation de la valeur fonctionnelle t_1, celle de l'argument t_2 et celle du corps de la fonction t' peuvent le modifier. La stratégie d'évaluation est fixée : le tas produit par l'évaluation de t_1 est utilisé pour évaluer t_2, celui produit par l'évaluation de t_2 est utilisé pour évaluer le corps de la fonction t' et finalement le tas produit finalement par la règle d'évaluation est celui produit par l'évaluation du corps de la fonction. L'ordre d'évaluation de t_1 et de t_2 aurait pu être inversé.

L'évaluation d'une fonction récursive produit une fermeture récursive. Par opposition aux fermetures simples, l'environnement capturé par une fermeture récursive contient implicitement la fermeture elle-même. Pour rendre explicite cette « capture récursive », la règle d'application devra être spécialisée au cas des fermetures récursives.

$$\frac{}{\mu, \rho \vdash \texttt{fix}\ f = \lambda\, x.t \Rightarrow \langle\!\langle f = \lambda x.t, \rho \rangle\!\rangle / \mu} \text{[Eval-RecClos]}$$

Ainsi, lorque le terme de gauche d'une application s'évalue vers une fermeture récursive $\langle\!\langle f = \lambda x.t', \rho' \rangle\!\rangle$, la seule différence avec la règle d'application en présence de fermeture simple sera l'ajout dans l'environnement nécessaire à l'évaluation du corps t' d'une association $f \mapsto \langle\!\langle f = \lambda x.t', \rho' \rangle\!\rangle$, en plus de l'association de l'argument x au

résultat de l'évaluation de t_2.

$$\frac{\begin{array}{l}\mu, \rho \vdash t_1 \Rightarrow \langle\!\langle f = \lambda x.t', \rho' \rangle\!\rangle / \mu' \qquad \mu', \rho \vdash t_2 \Rightarrow v_2 / \mu'' \\ \mu'', \rho' \oplus \{f \mapsto \langle\!\langle f = \lambda x.t', \rho' \rangle\!\rangle \,;\, x \mapsto v_2\} \vdash t' \Rightarrow v / \mu'''\end{array}}{\mu, \rho \vdash t_1 \; t_2 \Rightarrow v / \mu'''} \; [\textsc{Eval-RecApp}]$$

Lors de l'évaluation d'une définition locale $\texttt{let}\ x = t_x\ \texttt{in}\ t$, le terme t_x, définissant la variable x, est évalué dans l'environnement initial. Puis la valeur obtenue est utilisée pour étendre cet environnement et permettre l'évaluation de la suite du programme. Ainsi, du point de vue de l'évaluation, les termes $\texttt{let}\ x = t_x\ \texttt{in}\ t$ et $(\lambda x.t)\ t_x$ sont équivalents.

$$\frac{\mu, \rho \vdash t_x \Rightarrow v_x / \mu' \qquad \mu', \rho \oplus \{x \mapsto v_x\} \vdash t \Rightarrow v / \mu''}{\mu, \rho \vdash \texttt{let}\ x = t_x\ \texttt{in}\ t \Rightarrow v / \mu''} \; [\textsc{Eval-LetIn}]$$

Traits impératifs Pour allouer une nouvelle référence $\texttt{ref}(t)$, on commence par évaluer l'argument t. Puis la valeur obtenue est stockée dans un nouvel emplacement du tas. Le résultat de l'évaluation est l'étiquette correspondant à ce nouvel emplacement.

$$\frac{\mu, \rho \vdash t \Rightarrow v / \mu' \qquad \ell \notin dom(\mu')}{\mu, \rho \vdash \texttt{ref}(t) : \ell / \mu' \oplus \{\ell \mapsto v\}} \; [\textsc{Eval-Ref}]$$

Ainsi pour accéder à une référence $!(t)$, le terme t doit s'évaluer vers une étiquette appartenant effectivement au tas. Il suffit ensuite de lire la valeur correspondante. De la même manière, pour modifier une référence $t_1 := t_2$, le terme t_1 doit s'évaluer vers une étiquette. L'emplacement correspondant dans le tas est ensuite modifié avec la valeur résultant de l'évaluation de t_2.

$$\frac{\mu, \rho \vdash t \Rightarrow \ell / \mu' \qquad \ell \in dom(\mu')}{\mu, \rho \vdash !(t) \Rightarrow \mu'(\ell) / \mu'} \; [\textsc{Eval-Access}]$$

$$\frac{\mu, \rho \vdash t_1 \Rightarrow \ell / \mu' \qquad \ell \in dom(\mu') \qquad \mu', \rho \vdash t_2 \Rightarrow v / \mu''}{\mu, \rho \vdash t_1 := t_2 \Rightarrow 0 / \mu'' \oplus \{\ell \mapsto v\}} \; [\textsc{Eval-Modify}]$$

L'évaluation d'une séquence $t_1 \,;\, t_2$ ignore la valeur produite par l'évaluation de t_1 et renvoie la valeur produite par l'évaluation de t_2. Néanmoins, les modifications apportées au tas par l'évaluation de t_1 ne sont pas ignorées.

$$\frac{\mu, \rho \vdash t_1 \Rightarrow v_1 / \mu' \qquad \mu', \rho \vdash t_2 \Rightarrow v_2 / \mu''}{\mu, \rho \vdash t_1 \,;\, t_2 \Rightarrow v_2 / \mu''} \; [\textsc{Eval-Seq}]$$

Données, accesseurs et structures de contrôle Les booléens `false` et `true` sont représentés respectivement par les entiers mémoire 0 et 1.

$$\text{[Eval-False]}\ \frac{}{\mu, \rho \vdash \texttt{false} \Rightarrow 0/\mu} \qquad \text{[Eval-True]}\ \frac{}{\mu, \rho \vdash \texttt{true} \Rightarrow 1/\mu}$$

$$\text{[Eval-And]}\ \frac{\mu, \rho \vdash t_1 \Rightarrow 1/\mu' \qquad \mu', \rho \vdash t_2 \Rightarrow v/\mu''}{\mu, \rho \vdash t_1\ \texttt{\&\&}\ t_2 \Rightarrow v/\mu''} \qquad \text{[Eval-And-False]}\ \frac{\mu, \rho \vdash t_1 \Rightarrow 0/\mu'}{\mu, \rho \vdash t_1\ \texttt{\&\&}\ t_2 \Rightarrow 0/\mu'}$$

$$\text{[Eval-Or]}\ \frac{\mu, \rho \vdash t_1 \Rightarrow 0/\mu' \qquad \mu', \rho \vdash t_2 \Rightarrow v/\mu''}{\mu, \rho \vdash t_1\ \texttt{||}\ t_2 \Rightarrow v/\mu''} \qquad \text{[Eval-Or-True]}\ \frac{\mu, \rho \vdash t_1 \Rightarrow 1/\mu'}{\mu, \rho \vdash t_1\ \texttt{||}\ t_2 \Rightarrow 1/\mu'}$$

$$\text{[Eval-IfTrue]}\ \frac{\mu, \rho \vdash t_c \Rightarrow 1/\mu' \qquad \mu', \rho \vdash t_1 \Rightarrow v_1/\mu''}{\mu, \rho \vdash \texttt{if}\ t_c\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \Rightarrow v_1/\mu''} \qquad \text{[Eval-IfFalse]}\ \frac{\mu, \rho \vdash t_c \Rightarrow 0/\mu' \qquad \mu', \rho \vdash t_2 \Rightarrow v_2/\mu''}{\mu, \rho \vdash \texttt{if}\ t_c\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \Rightarrow v_2/\mu''}$$

Les entiers sont directement représentés par un entier-mémoire. La notation $\widehat{\texttt{op}}$ représente l'opération arithmétique entière correspondant à la construction syntaxique `op`.

$$\text{[Eval-Int]}\ \frac{}{\mu, \rho \vdash i \Rightarrow i/\mu}$$

$$\text{[Eval-Arith]}\ \frac{\mu, \rho \vdash t_1 \Rightarrow i_1/\mu' \qquad \mu', \rho \vdash t_2 \Rightarrow i_2/\mu''}{\mu, \rho \vdash t_1\ \texttt{op}\ t_2 \Rightarrow i_1\ \widehat{\texttt{op}}\ i_2/\mu''}$$

Le produit cartésien est représenté par un bloc.

$$\text{[Eval-Prod]}\ \frac{\mu, \rho \vdash t_1 \Rightarrow v_1/\mu' \qquad \mu', \rho \vdash t_2 \Rightarrow v_2/\mu''}{\mu, \rho \vdash (t_1,\ t_2) \Rightarrow \texttt{Blk}(v_1,\ v_2)/\mu''}$$

$$\text{[Eval-Fst]}\ \frac{\mu, \rho \vdash t \Rightarrow \texttt{Blk}(v_1,\ v_2)/\mu'}{\mu, \rho \vdash \texttt{fst}(t) \Rightarrow v_1/\mu'} \qquad \text{[Eval-Snd]}\ \frac{\mu, \rho \vdash t \Rightarrow \texttt{Blk}(v_1,\ v_2)/\mu'}{\mu, \rho \vdash \texttt{snd}(t) \Rightarrow v_2/\mu'}$$

La liste vide est représentée par l'entier 0 et le constructeur de listes est représenté par un bloc.

$$\text{[Eval-Nil]}\ \frac{}{\mu, \rho \vdash [] \Rightarrow 0/\mu} \qquad \text{[Eval-Cons]}\ \frac{\mu, \rho \vdash t_1 \Rightarrow v_1/\mu' \qquad \mu', \rho \vdash t_2 \Rightarrow v_2/\mu''}{\mu, \rho \vdash t_1 :: t_2 \Rightarrow \texttt{Blk}(v_1,\ v_2)/\mu''}$$

$$\text{[Eval-Case-Nil]}\quad \frac{\mu, \rho \vdash t \Rightarrow 0/\mu' \qquad \mu', \rho \vdash t_1 \Rightarrow v_1/\mu''}{\mu, \rho \vdash (\texttt{case}\ t\ \texttt{of}\ [] \to t_1 \mid x :: xs \to t_2) \Rightarrow v_1/\mu''}$$

$$\text{[Eval-Case-Cons]}\quad \frac{\mu, \rho \vdash t \Rightarrow \texttt{Blk}(v_1, v_2)/\mu' \qquad \mu', \rho \oplus \{x \mapsto v_1; xs \mapsto v_2\} \vdash t_2 \Rightarrow v_3/\mu''}{\mu, \rho \vdash (\texttt{case}\ t\ \texttt{of}\ [] \to t_1 \mid x :: xs \to t_2) \Rightarrow v_3/\mu''}$$

Le constructeur `None` est représentée par l'entier 0, et le constructeur `Some`(_) est représenté par un bloc contenant la valeur encapsulée et l'entier 0. Dans un environnement d'exécution réel, un compilateur créera un bloc de taille 1.

$$\text{[Eval-None]}\quad \frac{}{\mu, \rho \vdash \texttt{None} \Rightarrow 0/\mu} \qquad \text{[Eval-Some]}\quad \frac{\mu, \rho \vdash t_1 \Rightarrow v_1/\mu'}{\mu, \rho \vdash \texttt{Some}(t_1) \Rightarrow \texttt{Blk}(v_1, 0)/\mu'}$$

$$\text{[Eval-CaseOpt-None]}\quad \frac{\mu, \rho \vdash t \Rightarrow 0/\mu' \qquad \mu', \rho \vdash t_1 \Rightarrow v_1/\mu''}{\mu, \rho \vdash (\texttt{case}\ t\ \texttt{of None} \to t_1 \mid \texttt{Some}(x) \to t_2) \Rightarrow v_1/\mu''}$$

$$\text{[Eval-CaseOpt-Some]}\quad \frac{\mu, \rho \vdash t \Rightarrow \texttt{Blk}(v_1, 0)/\mu' \qquad \mu', \rho \oplus \{x \mapsto v_1\} \vdash t_2 \Rightarrow v_2/\mu''}{\mu, \rho \vdash (\texttt{case}\ t\ \texttt{of None} \to t_1 \mid \texttt{Some}(x) \to t_2) \Rightarrow v_2/\mu''}$$

Égalité polymorphe Pour définir la sémantique opérationnelle de l'égalité polymorphe, on définit préalablement une notion d'égalité sur les valeurs, notée $\_ \doteq \_$. Ainsi, l'égalité sur les entiers est l'égalité usuelle ; deux blocs sont égaux si leurs arguments sont égaux ; deux références sont égales si elles sont représentées par la même étiquette et deux fermetures (récursives ou non) sont toujours distinctes. La relation complémentaire, l'inégalité, sera noté $\dot{\neq}$.

$$\frac{}{i \doteq i} \qquad \frac{v_1 \doteq v_1' \qquad v_2 \doteq v_2'}{\texttt{Blk}(v_1, v_2) \doteq \texttt{Blk}(v_1', v_2')} \qquad \frac{}{\ell \doteq \ell}$$

Cette définition de l'égalité est une simplification de la fonction d'égalité polymorphe utilisée par OCaml. En particulier, notre système utilise pour le cas des références une notion d'égalité physique, alors que l'égalité d'OCaml compare le contenu des références. Ainsi, dans notre système le programme ci-dessous s'évaluera vers la valeur `false`, alors que le programme OCaml équivalent s'évalue vers `true`.

$$\texttt{ref}(3) = \texttt{ref}(3)$$

Ce choix permet de simplifier la définition de l'égalité polymorphe en présence de cycles dans le tas.

$$\frac{\mu, \rho \vdash t_1 \Rightarrow v_1/\mu' \qquad \mu', \rho \vdash t_2 \Rightarrow v_2/\mu'' \qquad v_1 \doteq v_2}{\mu, \rho \vdash t_1 = t_2 \Rightarrow 1/\mu''} \text{[Eval-Eq]}$$

$$\frac{\mu, \rho \vdash t_1 \Rightarrow v_1/\mu' \qquad \mu', \rho \vdash t_2 \Rightarrow v_2/\mu'' \qquad v_1 \dot{\neq} v_2}{\mu, \rho \vdash t_1 = t_2 \Rightarrow 0/\mu''} \text{[Eval-NotEq]}$$

2.2.3 Évaluation infinie

Dans la sémantique opérationnelle définie à la section précédente, certains programmes ne peuvent pas s'évaluer, ou autrement dit, il existe certains termes t, environnements ρ et tas μ pour lesquels il n'existe aucun couple valeur/tas v/μ' vérifiant $\mu, \rho \vdash t \Rightarrow v/\mu'$. C'est le cas notamment de programmes mal formés comme par exemple l'addition entière d'un couple et d'une liste vide : $(1, 2) + []$; l'application d'une projection à un entier : $\texttt{fst}(3)$; ou encore l'application d'un terme non fonctionnel : 4 5. C'est aussi le cas pour les programmes dont l'évaluation ne termine jamais comme $\Omega \equiv \delta\ \delta$ où $\delta \equiv \lambda x.(x\ x)$. La proposition 2.2.2 démontre à l'aide du lemme 2.2.1 que ce terme Ω ne peut s'évaluer dans le système d'évaluation qui vient d'être défini.

Lemme 2.2.1. *Soit deux environnements d'exécution ρ et ρ_δ, deux tas μ et μ', un terme t, une valeur v et une variable x. Si $\mu, \rho \oplus \{x \mapsto \langle \delta, \rho_\delta \rangle\} \vdash t \Rightarrow v/\mu'$ alors $t \not\equiv x\ x$.*

Preuve La preuve se fait par induction sur l'évaluation de t. On nomme ρ_x l'environnement $\rho \oplus \{x \mapsto \langle \lambda x.(x\ x), \rho_\delta \rangle\}$. Si t n'est pas une application, le résultat est immédiat. Sinon, on pose $t \equiv t_1\ t_2$ et la dérivation du jugement d'évaluation se termine par la règle [Eval-App] ou [Eval-RecApp]. Dans le second cas, on vérifie facilement $t_1 \not\equiv x$. Dans le premier cas, la dérivation peut se décomposer ainsi :

$$\frac{\begin{array}{c}\mu, \rho_x \vdash t_1 \Rightarrow \langle \lambda y.t' \rho' \rangle/\mu_1 \qquad \mu_1, \rho_x \vdash t_2 \Rightarrow v_2/\mu_2 \\ \mu_2, \rho' \oplus \{y \mapsto v_2\} \vdash t' \Rightarrow v/\mu'\end{array}}{\mu, \rho_x \vdash t_1\ t_2 \Rightarrow v/\mu'} \text{[Eval-App]}$$

Si $t_1 \equiv x$, la première prémisse implique $t' \equiv y\ y$. D'un autre côté, si $t_2 \equiv x$, alors $v_2 \equiv \langle \delta, \rho_\delta \rangle$ et en appliquant l'hypothèse d'induction à la troisième prémisse, on obtient $t' \not\equiv y\ y$. On ne peut donc à la fois avoir $t_1 \equiv x$ et $t_2 \equiv x$. On conclut $t \not\equiv x\ x$. □

Proposition 2.2.2. *Soit un environnement d'exécution ρ, deux tas μ et μ', un terme t et une valeur v. Si $\mu, \rho \vdash t \Rightarrow v/\mu'$ alors $t \not\equiv \Omega$.*

Preuve Comme dans la preuve du lemme précédent, le cas intéressant est $t \equiv t_1\ t_2$ et la dérivation d'évaluation a la forme suivante :

$$\frac{\begin{array}{c}[\textsc{Eval-App}] \\ \mu, \rho \vdash t_1 \Rightarrow \langle \lambda x.t', \rho' \rangle / \mu' \qquad \mu', \rho \vdash t_2 \Rightarrow v_2/\mu'' \\ \mu'', \rho' \oplus \{x \mapsto v_2\} \vdash t' \Rightarrow v/\mu'''\end{array}}{\mu, \rho \vdash t_1\ t_2 \Rightarrow v/\mu'''}$$

Si $t_1 \equiv \delta$, alors $t' \equiv x\ x$. D'un autre côté, si $t_2 = \delta$, alors une application directe du lemme précédent permet d'affirmer $t' \not\equiv x\ x$. On en conlut $t \not\equiv \Omega$. □

Lorsque le langage de programmation sera étendu avec des primitives de communication, certains programmes qui bouclent, comme par exemple le programme d'un serveur réseau, seront des programmes intéressants à étudier. Il sera alors nécessaire de distinguer les programmes mal formés des programmes qui bouclent. Pour préparer cette distinction, on définit ici un second système de dérivation pour décrire les programmes qui bouclent. Ce système définit coinductivement un ensemble de triplets (tas, environnement, terme), notés $\mu, \rho \vdash t \overset{\infty}{\Rightarrow}$ ce qui signifie : « dans l'environnement ρ et le tas initial μ, l'évaluation du terme t ne termine jamais. » Les éléments n'appartenant pas à cet ensemble sont notés : $\mu, \rho \vdash t \not\overset{\infty}{\Rightarrow}$. Cet ensemble est construit pour être disjoint du domaine de la relation d'évaluation, ainsi, si $\mu, \rho \vdash t \overset{\infty}{\Rightarrow}$ alors $\mu, \rho \vdash t \not\Rightarrow$ (propriétés 2.2.4 et 2.2.5).

Pour suivre les notations de Leroy et Grall [2009], les règles d'inférence à interpréter coinductivement seront notées à l'aide d'une double barre. Ce système de règles est construit à partir du système précédent, par une transformation systématique. À chacune des règles du premier système va correspondre autant de règles dans le nouveau système que la règle a de prémisses de la forme $\mu, \rho \vdash t \Rightarrow v/\mu'$; chacune de ces prémisses fournissant une possibilité de boucle. Pour rendre déterministe le système, lorsqu'une règle contient plusieurs prémisses de cette sorte, leur ordre et les conditions annexes sont préservées. Par exemple dans le cas de l'application, on peut boucler dans trois cas mutuellement exclusifs : si le terme de gauche boucle ; s'il ne boucle pas et si le second terme boucle ; si aucun de ces sous-termes ne boucle et si l'évaluation elle-même boucle. Il en est de même ensuite pour le reste des règles.

$$\frac{\begin{array}{c}[\textsc{EvalInf-App-Left}] \\ \mu, \rho \vdash t_1 \overset{\infty}{\Rightarrow}\end{array}}{\mu, \rho \vdash t_1\ t_2 \overset{\infty}{\Rightarrow}} \qquad \frac{\begin{array}{c}[\textsc{EvalInf-App-Right}] \\ \mu, \rho \vdash t_1 \Rightarrow v_1/\mu' \qquad \mu', \rho \vdash t_2 \overset{\infty}{\Rightarrow}\end{array}}{\mu, \rho \vdash t_1\ t_2 \overset{\infty}{\Rightarrow}}$$

$$\frac{\begin{array}{c}[\textsc{EvalInf-App}] \\ \mu, \rho \vdash t_1 \Rightarrow \langle \lambda x.t', \rho' \rangle / \mu' \qquad \mu', \rho \vdash t_2 \Rightarrow v_2/\mu'' \\ \mu'', \rho' \oplus \{x \mapsto v_2\} \vdash t' \overset{\infty}{\Rightarrow}\end{array}}{\mu, \rho \vdash t_1\ t_2 \overset{\infty}{\Rightarrow}}$$

$$\frac{\mu, \rho \vdash t_1 \Rightarrow \langle\!\langle f = \lambda x.t', \rho' \rangle\!\rangle / \mu' \qquad \mu', \rho \vdash t_2 \Rightarrow v_2 / \mu'' \qquad \mu'', \rho' \oplus \{f \mapsto \langle\!\langle f = \lambda x.t', \rho' \rangle\!\rangle; x \mapsto v_2\} \vdash t' \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash t_1\ t_2 \overset{\infty}{\Rightarrow}} \text{[EvalInf-RecApp]}$$

$$\frac{\mu, \rho \vdash t_x \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash \texttt{let}\ x = t_x\ \texttt{in}\ t \overset{\infty}{\Rightarrow}} \text{[EvalInf-LetIn-Left]} \qquad \frac{\mu, \rho \vdash t_x \Rightarrow v_x / \mu' \qquad \mu', \rho \oplus \{x \mapsto v_x\} \vdash t \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash \texttt{let}\ x = t_x\ \texttt{in}\ t \overset{\infty}{\Rightarrow}} \text{[EvalInf-LetIn-Right]}$$

$$\frac{\mu, \rho \vdash t_1 \Rightarrow v_1 / \mu' \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash t_1 :: t_2 \overset{\infty}{\Rightarrow}} \text{[EvalInf-ConsLeft]} \qquad \frac{\mu, \rho \vdash t_1 \Rightarrow v_1 / \mu' \qquad \mu', \rho \vdash t_2 \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash t_1 :: t_2 \overset{\infty}{\Rightarrow}} \text{[EvalInf-ConsRight]}$$

$$\frac{\mu, \rho \vdash t_1 \Rightarrow v_1 / \mu' \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash (t_1, t_2) \overset{\infty}{\Rightarrow}} \text{[EvalInf-ProdLeft]} \qquad \frac{\mu, \rho \vdash t_1 \Rightarrow v_1 / \mu' \qquad \mu', \rho \vdash t_2 \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash (t_1, t_2) \overset{\infty}{\Rightarrow}} \text{[EvalInf-ProdRight]}$$

$$\frac{\mu, \rho \vdash t \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash \texttt{Some}(t) \overset{\infty}{\Rightarrow}} \text{[EvalInf-Some]}$$

$$\frac{\mu, \rho \vdash t_1 \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash t_1 \texttt{op} t_2 \overset{\infty}{\Rightarrow}} \text{[EvalInf-Plus-Left]} \qquad \frac{\mu, \rho \vdash t_1 \Rightarrow v_1 / \mu' \qquad \mu', \rho \vdash t_2 \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash t_1 \texttt{op} t_2 \overset{\infty}{\Rightarrow}} \text{[EvalInf-Plus-Right]}$$

$$\frac{\mu, \rho \vdash t_c \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash \texttt{if}\ t_c\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \overset{\infty}{\Rightarrow}} \text{[EvalInf-If-Cond]} \qquad \frac{\mu, \rho \vdash t_c \Rightarrow 1 / \mu' \qquad \mu', \rho \vdash t_1 \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash \texttt{if}\ t_c\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \overset{\infty}{\Rightarrow}} \text{[EvalInf-If-True]}$$

$$\frac{\mu, \rho \vdash t_c \Rightarrow 0 / \mu' \qquad \mu', \rho \vdash t_2 \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash \texttt{if}\ t_c\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \overset{\infty}{\Rightarrow}} \text{[EvalInf-If-False]}$$

$$\frac{\mu, \rho \vdash t \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash \texttt{fst}(t) \overset{\infty}{\Rightarrow}} \text{[EvalInf-Fst]} \qquad \frac{\mu, \rho \vdash t \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash \texttt{snd}(t) \overset{\infty}{\Rightarrow}} \text{[EvalInf-Snd]}$$

[EvalInf-Case]
$$\frac{\mu, \rho \vdash t \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash \texttt{case}\ t\ \texttt{of}\ [] \to t_1 \mid x :: xs \to t_2 \overset{\infty}{\Rightarrow}}$$

[EvalInf-Case-Nil]
$$\frac{\mu, \rho \vdash t \Rightarrow 0/\mu' \qquad \mu', \rho \vdash t_1 \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash \texttt{case}\ t\ \texttt{of}\ [] \to t_1 \mid x :: xs \to t_2 \overset{\infty}{\Rightarrow}}$$

[EvalInf-Case-Cons]
$$\frac{\mu, \rho \vdash t \Rightarrow \texttt{Blk}(v_1,\, v_2)/\mu' \qquad \mu', \rho \oplus \{x \mapsto v_1; xs \mapsto v_2\} \vdash t_2 \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash \texttt{case}\ t\ \texttt{of}\ [] \to t_1 \mid x :: xs \to t_2 \overset{\infty}{\Rightarrow}}$$

[EvalInf-CaseOpt]
$$\frac{\mu, \rho \vdash t \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash \texttt{case}\ t\ \texttt{of None} \to t_1 \mid \texttt{Some}(x) \to t_2 \overset{\infty}{\Rightarrow}}$$

[EvalInf-CaseOpt-None]
$$\frac{\mu, \rho \vdash t \Rightarrow 0/\mu' \qquad \mu', \rho \vdash t_1 \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash \texttt{case}\ t\ \texttt{of None} \to t_1 \mid \texttt{Some}(x) \to t_2 \overset{\infty}{\Rightarrow}}$$

[EvalInf-CaseOpt-Some]
$$\frac{\mu, \rho \vdash t \Rightarrow \texttt{Blk}(v,\, 0)/\mu' \qquad \mu', \rho \oplus \{x \mapsto v\} \vdash t_2 \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash \texttt{case}\ t\ \texttt{of None} \to t_1 \mid \texttt{Some}(x) \to t_2 \overset{\infty}{\Rightarrow}}$$

[EvalInf-Access]
$$\frac{\mu, \rho \vdash t \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash !(t) \overset{\infty}{\Rightarrow}}$$

[EvalInf-Modify-Left]
$$\frac{\mu, \rho \vdash t_1 \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash t_1 := t_2 \overset{\infty}{\Rightarrow}}$$

[EvalInf-Modify-Right]
$$\frac{\mu, \rho \vdash t_1 \Rightarrow v_1/\mu' \qquad \mu', \rho \vdash t_2 \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash t_1 := t_2 \overset{\infty}{\Rightarrow}}$$

[EvalInf-Seq-Left]
$$\frac{\mu, \rho \vdash t_1 \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash t_1\,;\, t_2 \overset{\infty}{\Rightarrow}}$$

[EvalInf-Seq-Right]
$$\frac{\mu, \rho \vdash t_1 \Rightarrow v_1/\mu' \qquad \mu', \rho \vdash t_2 \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash t_1\,;\, t_2 \overset{\infty}{\Rightarrow}}$$

[EvalInf-Eq-Left]
$$\frac{\mu, \rho \vdash t_1 \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash t_1 = t_2 \overset{\infty}{\Rightarrow}}$$

[EvalInf-Eq-Right]
$$\frac{\mu, \rho \vdash t_1 \Rightarrow v_1/\mu' \qquad \mu', \rho \vdash t_2 \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash t_1 = t_2 \overset{\infty}{\Rightarrow}}$$

Exemple 2.2.1. *Dans ce système de dérivation coinductif, il est possible de montrer* $\emptyset, \emptyset \vdash \Omega \overset{\infty}{\Rightarrow}$. *La dérivation se construit en deux temps :*

$$\Delta \equiv \frac{\begin{array}{c}[\textsc{EvalInf-App}] \\ \emptyset, \emptyset \vdash x \Rightarrow \langle \lambda x.x\ x, \emptyset \rangle \qquad \emptyset, \emptyset \vdash x \Rightarrow \langle \lambda x.x\ x, \emptyset \rangle \qquad \Delta\end{array}}{\emptyset, \{x \mapsto \langle \lambda x.x\ x, \emptyset \rangle\} \vdash x\ x \overset{\infty}{\Rightarrow}}$$

$$\frac{\begin{array}{c}[\textsc{EvalInf-App}] \\ \emptyset, \emptyset \vdash \delta \Rightarrow \langle \lambda x.x\ x, \emptyset \rangle \qquad \emptyset, \emptyset \vdash \delta \Rightarrow \langle \lambda x.x\ x, \emptyset \rangle \qquad \Delta\end{array}}{\emptyset, \emptyset \vdash \delta\ \delta \overset{\infty}{\Rightarrow}}$$

2.2.4 Déterminisme et cas d'erreur

Une fois définis les deux systèmes de sémantique opérationnelle, évaluation simple et évaluation infinie, il est possible de vérifier d'une part que le premier système est déterministe : pour un triplet (μ, ρ, t) donné, il existe au plus un couple v/μ et une dérivation telle que $\mu, \rho \vdash t \Rightarrow v/\mu$ (proposition 2.2.3) – d'autre part, que les deux systèmes ne se chevauchent pas : s'il existe une valeur v telle que $\mu, \rho \vdash t \Rightarrow v/\mu$, alors $\mu, \rho \vdash t \overset{\infty}{\not\Rightarrow}$ (proposition 2.2.4) et, par contraposition, si $\mu, \rho \vdash t \overset{\infty}{\Rightarrow}$ alors $\mu, \rho \vdash t \not\Rightarrow$ (proposition 2.2.5). Cette section explicite le principe de ces preuves relativement systématiques.

Lorsque pour un terme t, un tas μ et un environnement ρ donnés, on a $\mu, \rho \vdash t \not\Rightarrow$ et $\mu, \rho \vdash t \overset{\infty}{\not\Rightarrow}$, on dit alors que le terme t s'évalue vers une erreur et on note :

$$\mu, \rho \vdash t \Rightarrow \texttt{err}$$

Proposition 2.2.3. *Soit un tas* μ, *un environnement* ρ, *un terme* t. *Il existe au plus un tas* μ', *une valeur* v *et une dérivation menant au jugement* $\mu, \rho \vdash t \Rightarrow v/\mu'$.

Preuve Par induction sur la structure du terme t. Pour les constructions syntaxiques n'admettant qu'une règle de dérivation, la propriété se montre généralement par application directe de l'hypothèse d'induction aux sous-termes. Dans le cas particulier du constructeur `ref(_)`, il est aussi nécessaire de supposer que le choix d'une nouvelle étiquette est déterministe. Pour les constructions syntaxiques admettant plusieurs règles de dérivation, comme l'application, la conditionnelle, le filtrage de listes et l'égalité, il est nécessaire de vérifier que les différentes règles ont des conditions d'application mutuellement exclusives. Par exemple, la première règle de filtrage de listes s'applique lorsque la valeur filtrée s'évalue vers l'entier 0, alors que la seconde règle s'applique lorsqu'elle s'évalue en un bloc. □

Proposition 2.2.4. *Soit deux tas* μ *et* μ', *un environnement* ρ, *un terme* t *et une valeur* v. *Si* $\mu, \rho \vdash t \Rightarrow v/\mu'$ *alors* $\mu, \rho \vdash t \overset{\infty}{\not\Rightarrow}$.

Preuve Par induction sur la dérivation d'évaluation. La technique systématique de construction du système coinductif suffit pour garantir cette propriété. □

Proposition 2.2.5. *Soit un tas μ, un environnement ρ, un terme t. Si $\mu, \rho \vdash t \overset{\infty}{\Rightarrow}$ alors $\mu, \rho \vdash t \not\Rightarrow$.*

Preuve Par contraposition de la proposition 2.2.4. □

Les trois propositions précédentes donnent le théorème suivant :

Théorème 2.2.6. *Soit un tas μ, un environnement ρ et un terme t. On a alors de manière exclusive l'une des propositions suivantes :*
- *il existe une unique valeur v et un tas μ', tels que $\mu, \rho \vdash t \Rightarrow v/\mu'$*
- *$\mu, \rho \vdash t \overset{\infty}{\Rightarrow}$*
- *$\mu, \rho \vdash t \Rightarrow$ `err`*

2.3 Système de types

L'objectif de cette section est de définir un système de types permettant de garantir la bonne évaluation d'un programme. Ainsi le théorème 2.5.4, dit de correction du typage, montrera que les termes typés ne s'évaluent pas en `err`.

Le système de types choisi ici est basé sur le système avec polymorphisme paramétrique de Hindley [1969] et Milner [1978] qui est classiquement associé aux langages de la famille ML. Le mécanisme de typage des références suit le principe de la *value restriction* de Wright [1995].

Cette section détaille dans un premier temps le langage de types utilisé, puis détaille les règles de typage des programmes. La seconde partie de cette section décrit un système de types pour les valeurs. Ce second système de types est notamment nécessaire à la démonstration du théorème de correction du typage ; il sera aussi utilisé dans la suite pour étendre le langage avec une primitive de désérialisation sûre.

Langage de types L'ensemble des types considérés est construit à partir de la grammaire suivante :

τ ::=	$\mathtt{T}(\tau, \ldots, \tau)$	*type paramétré, avec* $\mathtt{T} \in \mathcal{Q}_\tau$
$\mid$	α	*variable de type*
$\mid$	$\tau \to \tau$	*type fonctionnel*

L'ensemble des constructeurs de types, noté $\mathcal{Q}_\tau$, contient les booléens `Bool`, les entiers `Int`, le type singleton `Unit`, le type `Option(_)`, le type `List(_)`, les références `Ref(_)` et le produit cartésien $(\_ \times \_)$. Les constructeurs de listes et de références sont d'arité 1. Celui de produit cartésien est d'arité 2 et il est noté de manière infixe.

Substitution et schéma de types Une *substitution* θ sera une fonction partielle de l'ensemble des variables de type vers les types. Par extension, la notation $\theta(\tau)$ sera utilisée pour désigner le type τ dont les variables appartenant au domaine de θ ont été remplacées par leur image dans θ. Lorsque pour deux types τ et τ', il existe une

substitution θ telle que $\theta(\tau) = \tau'$, on dira quel le type τ est *plus général* que le type τ', ou que le type τ' est une *instance du type* τ. On notera alors $\tau \preccurlyeq \tau'$, ou $\tau \preccurlyeq_\theta \tau'$ lorsqu'il sera utile d'expliciter la substitution.

Un *schéma de types* est un type dont certaines variables ont été quantifiées universellement. Ils seront notés $\sigma = \forall \bar{\alpha}.\, \tau$ où $\bar{\alpha}$ est un sous-ensemble, éventuellement vide, des variables de types apparaissant dans τ. On appellera variables libres d'un schéma de types $\forall \bar{\alpha}.\, \tau$ les variables de types apparaissant dans τ qui n'appartiennent pas à $\bar{\alpha}$. On notera cet ensemble de variables libres $ftv(\sigma)$. Un type τ sera dit *instance d'un schéma de types* $\sigma = \forall \bar{\alpha}.\, \tau'$, s'il existe une substitution θ de domaine $\bar{\alpha}$ telle que $\tau = \theta(\tau')$. On notera $\sigma \preccurlyeq \tau$ ou $\sigma \preccurlyeq_\theta \tau$ lorsqu'il sera utile d'expliciter la substitution.

Un *environnement de typage* est une fonction partielle de l'ensemble des variables des programmes vers l'ensemble des schémas de type. Ils sont usuellement notés Γ. La fonction $ftv(\_)$ s'étend naturellement à un environnement de typage comme l'union des variables de type libres de son image.

2.3.1 Typage des termes

Le système de types est défini à l'aide d'un ensemble de règles de dérivation. Les jugements de ce système sont de la forme $\Gamma \vdash t : \tau$, ce qui signifie « dans un environnement de typage Γ, le terme t a le type τ. » Dans cette présentation, il y a exactement une règle d'inférence par construction syntaxique du langage source. Une variable x peut être typée avec un type τ si et seulement si τ est une instance de $\Gamma(x)$.

$$\frac{\Gamma(x) \preccurlyeq \tau}{\Gamma \vdash x : \tau} \text{[ML-Var]}$$

Une fonction $\lambda x.t$ sera correctement typée $\tau_1 \to \tau_2$ dans un environnement Γ, si dans l'environnement étendu avec l'association $\{x \mapsto \tau_1\}$, le corps t de la fonction est typable avec le type τ_2.

$$\frac{\Gamma \oplus \{x \mapsto \tau_1\} \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2} \text{[ML-Abs]}$$

De même une fonction récursive $\texttt{fix}\ f = \lambda x.t$ sera correctement typée $\tau_1 \to \tau_2$ dans un environnement Γ, si dans l'environnement étendu à la fois avec l'association $\{f \mapsto (\tau_1 \to \tau_2)\}$ et l'association $\{x \mapsto \tau_1\}$, le corps t de la fonction est correctement typé τ_2.

$$\frac{\Gamma \oplus \{f \mapsto (\tau_1 \to \tau_2)\,;\, x \mapsto \tau_1\} \vdash t : \tau_2}{\Gamma \vdash \texttt{fix}\ f = \lambda x.t : \tau_1 \to \tau_2} \text{[ML-RecAbs]}$$

Une application $t_1\ t_2$ sera correctement typée τ, s'il existe un type τ' tel que t_1 ait le

type fonctionnel $\tau' \to \tau$, et tel que l'argument effectif t_2 ait le type τ'.

$$\text{[ML-App]}\quad \frac{\Gamma \vdash t_1 : \tau' \to \tau \qquad \Gamma \vdash t_2 : \tau'}{\Gamma \vdash t_1\ t_2 : \tau}$$

Données, accesseurs et structures de contrôle Le typage des différents types de données et des structures de contrôle est classique.

$$\text{[ML-True]}\quad \frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}} \qquad \text{[ML-False]}\quad \frac{}{\Gamma \vdash \texttt{false} : \texttt{Bool}}$$

$$\text{[ML-And]}\quad \frac{\Gamma \vdash t_1 : \texttt{Bool} \qquad \Gamma \vdash t_2 : \texttt{Bool}}{\Gamma \vdash t_1\ \texttt{\&\&}\ t_2 : \texttt{Bool}} \qquad \text{[ML-Or]}\quad \frac{\Gamma \vdash t_1 : \texttt{Bool} \qquad \Gamma \vdash t_2 : \texttt{Bool}}{\Gamma \vdash t_1\ \texttt{||}\ t_2 : \texttt{Bool}}$$

$$\text{[ML-If]}\quad \frac{\Gamma \vdash t_1 : \texttt{Bool} \qquad \Gamma \vdash t_2 : \tau \qquad \Gamma \vdash t_3 : \tau}{\Gamma \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 : \tau}$$

$$\text{[ML-Int]}\quad \frac{}{\Gamma \vdash i : \texttt{Int}} \qquad \text{[ML-Arith]}\quad \frac{\Gamma \vdash t_1 : \texttt{Int} \qquad \Gamma \vdash t_2 : \texttt{Int}}{\Gamma \vdash t_1\ \texttt{op}\ t_2 : \texttt{Int}}$$

$$\text{[ML-Prod]}\quad \frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1, t_2) : (\tau_1 \times \tau_2)} \qquad \text{[ML-Fst]}\quad \frac{\Gamma \vdash t : (\tau_1 \times \tau_2)}{\Gamma \vdash \texttt{fst}(t) : \tau_1} \qquad \text{[ML-Snd]}\quad \frac{\Gamma \vdash t : (\tau_1 \times \tau_2)}{\Gamma \vdash \texttt{snd}(t) : \tau_2}$$

$$\text{[ML-Nil]}\quad \frac{}{\Gamma \vdash [] : \texttt{List}(\tau)} \qquad \text{[ML-Cons]}\quad \frac{\Gamma \vdash t_1 : \tau \qquad \Gamma \vdash t_2 : \texttt{List}(\tau)}{\Gamma \vdash t_1 :: t_2 : \texttt{List}(\tau)}$$

$$\text{[ML-Case]}\quad \frac{\Gamma \vdash t_1 : \texttt{List}(\tau') \qquad \Gamma \vdash t_2 : \tau \qquad \Gamma \oplus \{x : \tau'; xs : \texttt{List}(\tau')\} \vdash t_3 : \tau}{\Gamma \vdash (\texttt{case}\ t_1\ \texttt{of}\ [] \to t_2 \mid x :: xs \to t_3) : \tau}$$

$$\text{[ML-None]}\quad \frac{}{\Gamma \vdash \texttt{None} : \texttt{Option}(\tau)} \qquad \text{[ML-Some]}\quad \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \texttt{Some}(t) : \texttt{Option}(\tau)}$$

$$\text{[ML-CaseOpt]}\quad \frac{\Gamma \vdash t_1 : \texttt{Option}(\tau') \qquad \Gamma \vdash t_2 : \tau \qquad \Gamma \oplus \{x : \tau'\} \vdash t_3 : \tau}{\Gamma \vdash (\texttt{case}\ t_1\ \texttt{of None} \to t_2 \mid \texttt{Some}(x) \to t_3) : \tau}$$

$$\frac{\Gamma \vdash t_1 : \tau \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 = t_2 : \texttt{Bool}} \text{[ML-Eq]}$$

Références Le type $\texttt{Ref}(\tau)$ va servir à typer une référence sur une valeur de type τ ; et le type `Unit` est utilisé comme type de retour de l'opération d'affectation.

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash \texttt{ref}(t) : \texttt{Ref}(\tau)} \text{[ML-Ref]}$$

$$\frac{\Gamma \vdash t : \texttt{Ref}(\tau)}{\Gamma \vdash !(t) : \tau} \text{[ML-Access]} \qquad \frac{\Gamma \vdash t_1 : \texttt{Ref}(\tau) \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 := t_2 : \texttt{Unit}} \text{[ML-Modify]}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Unit} \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 ; t_2 : \tau} \text{[ML-Seq]}$$

Généralisation Il ne reste plus qu'à définir la règle de typage associée à la construction $\texttt{let}\ x = t_x\ \texttt{in}\ t$. Cette règle va permettre d'associer à la variable x, dans l'environnement de typage utilisé pour typer t, un schéma de types représentant l'ensemble des types acceptables pour le terme t_x.

Par exemple, la liste vide admet les types $\texttt{List}(\texttt{Int})$ et $\texttt{List}(\texttt{Bool})$. Elle admet aussi le type $\texttt{List}(\alpha)$ et tout les types de la forme $\texttt{List}(\tau)$. De même, la fonction $\lambda x.(x, x)$, qui construit un couple contenant deux fois son argument, admet le type $\alpha \to (\alpha \times \alpha)$ et tout les types de la forme $\tau \to (\tau \times \tau)$. Ainsi, le terme t du programme $\texttt{let}\ f = \lambda x.(x, x)\ \texttt{in}\ t$, peut être typé dans un environnement où la variable f est associée au schéma de types $\forall \alpha.\, \alpha \to (\alpha \times \alpha)$. En ce sens, cette fonction est dite polymorphe, elle pourra être appliquée à des arguments de types différents.

De manière plus générale, le lemme 2.3.1 permettra notamment de montrer pour un environnement de typage Γ et un terme t donnés que si $\Gamma \vdash t : \tau$ alors $\Gamma \vdash t : \theta(\tau)$ pour toute substitution θ telle que $dom(\theta) \# ftv(\Gamma)$, autrement dit, telle $dom(\theta) \cap ftv(\Gamma) = \emptyset$. Suivant cette idée, pour typer une définition locale $\Gamma \vdash \texttt{let}\ x = t_x\ \texttt{in}\ t : \tau$, lorsque $\Gamma \vdash t_x : \tau_x$, il est alors possible de typer t dans un environnement étendu avec l'association $\{x : gen(\Gamma, \tau)\}$ où $gen(\Gamma, \tau) = \forall \bar{\alpha}.\, \tau$ pour $\bar{\alpha} = ftv(\tau) \setminus ftv(\Gamma)$.

Dans un langage du style de mini-ML sans effet de bord, il possible de généraliser systématiquement le type d'une variable introduite par une définition. Ce n'est plus le cas en présence de références. Par exemple, dans le programme ci-dessous, bien que la valeur $\texttt{ref}(\lambda x.x)$ admette le type $\texttt{Ref}(\alpha \to \alpha)$, il serait incorrect d'associer à la variable id le schéma de types $\forall \alpha.\, \texttt{Ref}(\alpha \to \alpha)$.

$$\begin{array}{l}\texttt{let}\ id = \texttt{ref}(\lambda x.x)\ \texttt{in}\\ f := (\lambda x.x + 1)\,;\\ !(f)\ (3,1)\end{array}$$

En effet, avec une telle association ce programme serait typable ; les deux occurrences de f admettant alors les type $\texttt{Ref}(\texttt{Int} \to \texttt{Int})$ et $\texttt{Ref}((\texttt{Int} \times \texttt{Int}) \to (\texttt{Int} \times \texttt{Int}))$. Mais l'évaluation de ce programme produit une erreur, en effet l'application $!(f)\ (3,1)$ va tenter d'additionner un couple à un entier.

Suivant le principe de *value restriction* proposé par Wright [1995], le typage d'une définition locale $\texttt{let}\ x = t_x\ \texttt{in}\ t$ sera différent selon la nature de t_x. Il ne sera possible de généraliser le type de x uniquement lorsque t_x appartient au sous-ensemble ci-dessous.

$$\begin{array}{rll} t^v & ::= & x & \textit{variable}\\ & & \mid \lambda x.t & \textit{fonction}\\ & & \mid \texttt{fix}\ f = \lambda\, x.t & \textit{fonction récursive}\\ & & \mid \texttt{true} \mid \texttt{false} & \textit{booléen}\\ & & \mid i & \textit{entier}\\ & & \mid (t^v,\, t^v) & \textit{constructeur de paires}\\ & & \mid [] \mid t^v :: t^v & \textit{constructeurs de listes}\\ & & \mid \texttt{None} \mid \texttt{Some}(t^v) & \textit{constructeurs d'options}\end{array}$$

Comme le montrera le lemme 2.5.1 l'évaluation des termes de ce sous-ensemble possède la bonne propriété de ne pas modifier le tas.

$$\frac{\Gamma \vdash t_x^v : \tau_x \qquad \Gamma \oplus \{x \mapsto gen(\Gamma, \tau_x)\} \vdash t : \tau}{\Gamma \vdash \texttt{let}\ x = t_x^v\ \texttt{in}\ t : \tau}\ [\text{ML-LetV}] \qquad \frac{\Gamma \vdash t_x : \tau_x \qquad \Gamma \oplus \{x \mapsto \tau_x\} \vdash t : \tau}{\Gamma \vdash \texttt{let}\ x = t_x\ \texttt{in}\ t : \tau}\ [\text{ML-Let}]$$

Algorithme d'inférence Il est possible de construire à partir ce système de types un algorithme d'inférence permettant de calculer pour un programme donné son *type principal*. Le type principal d'un programme est un schéma de types σ tel que pour tout type τ vérifiant $\emptyset \vdash t : \tau$ alors τ est une instance de $\sigma \preccurlyeq \tau$. Le principe de construction d'un tel algorithme ne sera pas détaillé ici, mais une synthèse des techniques mise en œuvre dans les algorithmes d'inférence de types existant pour ML a été réalisée par Pottier et Rémy [2005]. Pour la suite de ce document, seul le lemme suivant, moins général, sera utile.

Lemme 2.3.1 (Instanciation)**.** *Étant donné un environnement de typage Γ, un type τ, un terme t et une substitution θ. Si $\Gamma \vdash t : \tau$, alors $\theta(\Gamma) \vdash t : \theta(\tau)$.*

Preuve Par induction sur la structure du terme t. □

Autrement dit, le système de types décrit ici est stable par substitution des variables de types. Ou encore : lorsqu'il possible de typer un terme t avec le type τ, il possible de typer ce terme avec toute les instances de $gen(\Gamma, \tau)$.

2.3.2 Typage des valeurs : relation de compatibilité

Afin de pouvoir établir à la section 2.5 du système de type par rapport au mécanisme d'évaluation — autrement dit, tout terme bien typé ne s'évalue par vers une erreur — cette section définit un système de types pour les valeurs. Ce système est construit de telle sorte que la valeur résultant de l'évaluation d'un programme de type τ s'évalue vers une valeur du même type (cf lemme 2.5.2, dit lemme de préservation du typage). Ce système de type des valeurs sera aussi utile à la section 2.4 pour caractériser l'ensemble des valeurs acceptables par une fonction de désérialisation typée.

Contrairement au système de types des programmes, il n'est pas possible de définir un système de types pour les valeurs possédant une notion de type principal en utilisant le même langage de types que les programmes. En effet, l'entier-mémoire 0 peut être le résultat de l'évaluation de l'entier 0, ou des constructeurs `false` et `None`. Il doit donc pouvoir être typable aussi bien avec le type `Int`, qu'avec le type `Bool` ou `Option`(α). Il est clair alors qu'il n'existe pas de type principal pour cette valeur : le seul schéma de types plus général que ces trois types $\forall\alpha.\,\alpha$ n'est pas acceptable. En ce sens, au lieu de chercher à caractériser une notion de type principal pour une valeur, nous définissons une notion de *compatibilité* d'une valeur avec un type. On dira par exemple que l'entier-mémoire 0 est compatible avec les types `Int`, `Bool` ou `Option`(α).

Notations Les jugements de typage des valeurs seront notés $\Psi \vdash v : \tau$ où Ψ représente un *environnement de typage du tas*, c'est-à-dire une fonction partielle de l'ensemble des étiquettes vers les types *clos*. On dira qu'un tas μ est correctement typé avec Ψ et on notera $\mu : \Psi$ lorsque pour toutes les étiquettes $\ell \in dom(\mu)$ on vérifie $\Psi \vdash \mu(\ell) : \Psi(l)$. On notera $\mu : \Psi \vdash v : \tau$ pour signifier à la fois $\mu : \Psi$ et $\Psi \vdash v : \tau$. On notera $\Psi \vdash v : \sigma$ pour signifier que la valeur v est typable avec toutes les instances de σ. Le lemme 2.3.6 démontrera que ce système de types est stable par substitution des variables de τ ; ainsi, pour montrer que $\Psi \vdash v : \forall\bar{\alpha}.\,\tau$, il suffira de montrer $\Psi \vdash v : \tau$. Avec cette notation, les jugements de typage des valeurs s'étendent naturellement aux environnements d'évaluation : on dira qu'un environnement d'évaluation ρ est compatible avec un environnement de typage Γ dans un environnement de typage du tas Ψ, ce que l'on notera $\Psi \vdash \rho : \Gamma$, lorsque pour toute variable $x \in dom(\rho) = dom(\Gamma)$, on vérifie $\Psi \vdash \rho(x) : \Gamma(x)$. On notera $\mu : \Psi \vdash \rho : \Gamma$ pour signifier à la fois $\mu : \Psi$ et $\Psi \vdash \rho : \Gamma$.

Règles de dérivation Pour n'introduire a priori qu'une règle de typage par construction syntaxique dans la définition des valeurs et simplifier certaines preuves, il est possible de définir deux relations Δ et ∇ servant respectivement à regrouper les cas des entiers et des blocs. La première permet d'associer à un type les entiers-mémoire qui en sont une représentation acceptable. Elle est définie par cas :

$$
\begin{array}{rcll}
0 & \triangle & \texttt{Unit} & (\triangle\text{-Unit}) \\
0 & \triangle & \texttt{Bool} & (\triangle\text{-False}) \\
1 & \triangle & \texttt{Bool} & (\triangle\text{-True}) \\
0 & \triangle & \texttt{List}(\tau) & (\triangle\text{-Nil}) \\
0 & \triangle & \texttt{Option}(\tau) & (\triangle\text{-None}) \\
i & \triangle & \texttt{Int} & (\triangle\text{-Int})
\end{array}
$$

Cette relation permet de définir la règle de typage des entiers-mémoire :

$$\frac{i \triangle \tau}{\Psi \vdash i : \tau} \quad [\text{V-Int}]$$

La seconde relation permet d'associer à un type τ les types $[\tau_1 ; \tau_2]$ que doivent respectivement vérifier les composants (v_1, v_2) d'un bloc $\texttt{Blk}(v_1, v_2)$ pour que ce bloc soit compatible avec le type τ. Elle est aussi définie par cas :

$$
\begin{array}{rcll}
[\tau\,;\texttt{List}(\tau)] & \nabla & \texttt{List}(\tau) & (\nabla\text{-Cons}) \\
[\tau_1\,;\tau_1] & \nabla & (\tau_1 \times \tau_2) & (\nabla\text{-Prod}) \\
[\tau\,;\texttt{Unit}] & \nabla & \texttt{Option}(\tau) & (\nabla\text{-Some})
\end{array}
$$

$$\frac{[\tau_1 ; \tau_2] \quad \nabla \quad \tau \qquad \Psi \vdash v_1 : \tau_1 \qquad \Psi \vdash v_2 : \tau_2}{\Psi \vdash \texttt{Blk}(v_1, v_2) : \tau} \quad [\text{V-Blk}]$$

Le type d'une étiquette est le type des références, paramétré par le type associé à l'étiquette dans l'environnement de typage du tas.

$$\frac{}{\Psi \vdash \ell : \texttt{Ref}(\Psi(\ell))} \quad [\text{V-Ref}]$$

Une fermeture $\langle \lambda x.t, \rho \rangle$ aura le type $\tau_1 \to \tau_2$ s'il existe un environnement de typage Γ permettant de typer la fonction $\lambda x.t$ avec $\tau_1 \to \tau_2$ et tel que l'environnement d'évaluation ρ soit compatible avec Γ. La règle de typage de la fermeture récursive suit le même principe.

$$\frac{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2 \qquad \Psi \vdash \rho : \Gamma}{\Psi \vdash \langle \lambda x.t, \rho \rangle : \tau_1 \to \tau_2} \quad [\text{V-Clos}]$$

$$\frac{\Gamma \vdash \texttt{fix}\ f = \lambda\, x.t : \tau_1 \to \tau_2 \qquad \Psi \vdash \rho : \Gamma}{\Psi \vdash \langle\!\langle f = \lambda x.t, \rho \rangle\!\rangle : \tau_1 \to \tau_2} \quad [\text{V-RecClos}]$$

Par construction l'environnement de typage nécessaire pour typer une fermeture sera une instance de l'environnement ayant servi à typer la fonction dans le programme original. Il est à noter qu'il peut exister plusieurs instanciations différentes de cet environnement menant à un même jugement $\Psi \vdash \langle \lambda x.t, \rho \rangle : \tau_1 \to \tau_2$. Par exemple, le programme ci-dessous a pour type principal $\forall \alpha.\, \alpha \to \texttt{Bool}$:

$$
\begin{array}{l}
\texttt{let}\ g = \lambda x.\lambda y.\, \texttt{fst}(x)\ \texttt{in} \\
g\ (\texttt{true}, \texttt{false})
\end{array}
$$

En particulier, dans la dérivation de type principale de ce programme, le jugement de typage concernant l'expression $\lambda y.\texttt{fst}(x)$ est :

$$\begin{array}{rl} & \Gamma \vdash \lambda y.\,\texttt{fst}(x) : \alpha \to \beta \\ \text{où} & \Gamma \equiv \{x : (\beta \times \gamma)\} \end{array}$$

Par ailleurs, l'évaluation du couple (`true`, `false`) produit le bloc $\texttt{Blk}(1, 0)$ et l'évaluation du programme complet produit la fermeture :

$$\begin{array}{rl} & \langle \lambda y.\,\texttt{fst}(x),\, \rho \rangle \\ \text{où} & \rho \equiv \{x \mapsto \texttt{Blk}(1, 0)\} \end{array}$$

Comme le lemme de préservation du typage par évaluation (lemme 2.5.2) le montrera dans le cas général, cette fermeture est compatible avec le type principal du programme : $\forall\alpha.\,\alpha \to \texttt{Bool}$. En effet, en utilisant l'environnement de typage $\{x : (\texttt{Bool} \times \texttt{Bool})\}$, on obtient la dérivation de typage suivante :

$$\dfrac{\dfrac{\vdots}{\{x : (\texttt{Bool} \times \texttt{Bool})\} \vdash \lambda y.\,\texttt{fst}(x) : \alpha \to \texttt{Bool}} \qquad \dfrac{\Delta}{\emptyset \vdash \texttt{Blk}(1, 0) : (\texttt{Bool} \times \texttt{Bool})}}{\emptyset \vdash \langle \lambda y.\,\texttt{fst}(x),\, \rho \rangle : \alpha \to \texttt{Bool}}$$

$$\Delta \equiv \dfrac{[\texttt{Bool}\,;\texttt{Bool}] \bigtriangledown (\texttt{Bool} \times \texttt{Bool}) \qquad \dfrac{1 \mathrel{\Delta} \texttt{Bool}}{\emptyset \vdash 1 : \texttt{Bool}} \qquad \dfrac{0 \mathrel{\Delta} \texttt{Bool}}{\emptyset \vdash 0 : \texttt{Bool}}}{\emptyset \vdash \texttt{Blk}(1, 0) : (\texttt{Bool} \times \texttt{Bool})}$$

Mais, la seconde valeur du couple n'étant jamais utilisée, il est aussi possible pour typer cette fermeture avec le type $\alpha \to \texttt{Bool}$ d'utiliser l'environnement $\{x : (\texttt{Bool} \times \texttt{Option}(\texttt{Int}))\}$. De manière générale, il est possible d'utiliser toutes les instanciations $\theta(\Gamma)$ de l'environnement de typage Γ — utilisé pour calculer le type principal $\alpha \to \beta$ de $\lambda y.\,\texttt{fst}(x)$ — qui vérifient à la fois $\theta(\alpha \to \beta) \equiv \alpha \to \texttt{Bool}$ et $0 \mathrel{\Delta} \theta(\gamma)$. La première condition permet, avec le lemme 2.3.1 de stabilité du typage, de vérifier $\theta(\Gamma) \vdash \lambda y.\,\texttt{fst}(x) : \alpha \to \texttt{Bool}$; le seconde condition permet de typer l'environnement d'évaluation $\rho : \theta(\Gamma)$.

Relation de décomposition Les relations Δ et ∇ définies ci-dessus ont été introduite pour factoriser les règles de typage portant respectivement sur les entiers-mémoire et les blocs. Ces relations sont stables par substitution (propriétés 2.3.2 et 2.3.3) et la décomposition d'un type par la relation ∇ est unique (propriété 2.3.4). Si le détail des cas de définition de ces relations est important pour établir la preuve de correction du typage, seules ces trois propriétés et le lemme 3.4.6 seront nécessaires au chapitre 3 pour établir les propriétés de l'algorithme de vérification de type.

Propriété 2.3.2. *Soit un entier i, un type τ et une substitution θ. Si $i \mathrel{\Delta} \tau$ alors $i \mathrel{\Delta} \theta(\tau)$.*

Preuve Par cas de définition.

Propriété 2.3.3. *Soit trois types τ, τ' et τ'' et une substitution θ. Si $[\tau' ; \tau''] \ \triangledown \ \tau$ alors $[\theta(\tau') ; \theta(\tau'')] \ \triangledown \ \theta(\tau)$.*

Preuve Par cas de définition.

Propriété 2.3.4. *Soit cinq types τ_1, τ_2, τ_1', τ_2' et τ. Si $[\tau_1 ; \tau_2] \ \triangledown \ \tau$ et $[\tau_1' ; \tau_2'] \ \triangledown \ \tau$ alors $\tau_1 = \tau_1'$ et $\tau_2 = \tau_2'$.*

Preuve La relation $\triangledown$ est définie par cas sur le constructeur de types de tête du type τ.

Type universel Le système de types des valeurs caractérise l'ensemble des valeurs compatibles avec un type donné et il va permettre à la section 2.4 de caractériser l'ensemble des valeurs acceptables par une primitive de désérialisation typée. Le chapitre 3 construira à partir de cette spécification un algorithme vérifiant la compatibilité d'une valeur avec un type. Cet algorithme ne sera pas toujours capable de reconstruire le type de toutes les sous-valeurs. Ces valeurs résiduelles seront alors typées avec un type universel, noté $\star$, qui permet de typer toutes les valeurs.

$$\frac{}{\Psi \vdash v : \star} \quad [\text{V-}\star]$$

L'emploi de ce type universel ne met pas en danger la sûreté d'exécution des programmes utilisant des primitives de désérialisation, dans la mesure où l'on s'apercevra que ces valeurs ne sont pas utiles à l'évaluation du reste du programme. Par exemple, pour vérifier la compatiblité avec le type $\alpha \to \texttt{Bool}$ de la fermeture $\langle \lambda y.\, \texttt{fst}(x), \{x \mapsto \texttt{Blk}(1, 0)\}\rangle$ — utilisée ci-dessus pour illustrer la règle de typage des fermetures — l'algorithme proposé reconstruira un environnement de typage $\{x : (\texttt{Bool} \times \star)\}$. Le second élément du bloc est clairement inutile à la bonne exécution du programme.

Type vide Comme dans le cas du système de typage des programmes, le typage des valeurs est stable par substitution (lemme 2.3.6). Mais, ce lemme n'a pas tout-à-fait la même signification que le lemme équivalent pour les programmes (lemme 2.3.1). Dans le système de types des programmes, une variable de type apparaissant dans l'environnemment de typage représente un type encore inconnu.

$$\frac{\{x : \alpha\} \vdash x : \alpha}{\emptyset \vdash \lambda x.x : \alpha \to \alpha}$$

Ainsi, la variable x de la fonction identité a le type α. Durant l'évaluation du corps de cette fonction, la variable x pourra, selon les contextes d'utilisation, être liée à toutes les valeurs existantes. Dans le système de types des valeurs où les environnement de typage sont clos et en l'absence de types fonctionnels, une variable de type représente une valeur appartenant à tous les types *à la fois*. Comme il n'existe pas de telle valeur, une variable de type représente ici systématiquement un type vide : il n'existe aucune valeur compatible avec le type α.

Propriété 2.3.5. *Soit un environnement de typage du tas Ψ, une valeur v et un type τ. Si $\Psi \vdash v : \tau$ alors $\tau \neq \alpha$.*

Preuve Dans tous les jugement de typage $\Psi \vdash v : \tau$ produits par les règles de dérivation [V-. . .], le type τ est de la forme $\mathtt{T}(\tau_1 , \ldots , \tau_n)$. □

Lemme 2.3.6 (Instanciation)**.** *Étant donné un type τ, une valeur v, un environnement de typage du tas Ψ et une substitution θ. Si $\Psi \vdash v : \tau$, alors $\Psi \vdash v : \theta(\tau)$.*

Preuve Par induction sur la structure de v. Dans le cas des blocs, la propriété 2.3.3 est nécessaire en plus de l'hypothèse d'induction. Le cas des fermetures utilise le lemme 2.3.1 d'instanciation du système de types des programmes. Dans le cas des références, il est à noter que l'environnement Ψ est clos par définition. Ainsi, pour toute substitution θ, on vérifie trivialement $\Psi(\ell) = \theta(\Psi(\ell))$.

2.4 Primitives de (dé)sérialisation

Avant d'énoncer et de vérifier les théorèmes classiques de correction du système de types, cette section étend le langage étudié avec des primitives de sérialisation et de désérialisation.

Sérialisation La syntaxe des programmes est d'abord étendue avec une primitive de sérialisation.

t ::= ...
| `marshall` t *primitive de sérialisation*

Cette primitive accepte en argument toutes les valeurs du langage et les renvoie sous une forme sérialisée. Lorsque une valeur sérialisé contiendra des références à un tas μ, la partie "utile" du tas sera elle aussi sérialisée : le fragment à sérialiser du tas pour une valeur v sera noté $\mu_{|v}$, il correspond à la restriction du tas μ aux étiquettes récursivement accessibles depuis v. Cette restriction peut être définie comme la plus petite solution de l'équation ci-dessous, où $fl(v)$ représente l'ensemble des étiquettes apparaissant dans la valeur v.

$$S = fl(v) \cup \left(\bigcup_{l \in S} fl(\mu(l)) \right)$$

La représentation sérialisée d'une valeur v et d'un tas μ sera noté $\overline{[v\,;\mu]}$. Intuitivement, cela correspond à une séquence de valeurs sérialisées $[\overline{v}\,;\,\overline{\mu}(\ell_1)\,;\,\ldots\,;\,\overline{\mu}(\ell_n)]$: c'est-à-dire la valeur v suivi des valeurs du tas, où les étiquettes ont été remplacées par des indices dans la séquence de valeurs engendrée. La sérialisation d'une valeur v dans la tas μ produit la séquence $\overline{[v\,;\mu_{|v}]}$.

Exemple 2.4.1. *Soit le tas μ ci-dessous :*

$$\mu \quad \equiv \quad \{\ell_1 \mapsto \texttt{Blk}(23, \texttt{Blk}(\ell_3, 0))\,;\, \ell_2 \mapsto 0\,;\, \ell_3 \mapsto \texttt{Blk}(\ell_1, 0)\}$$

La sérialisation de la valeur $v \equiv \texttt{Blk}(\ell_1, 0)$ produit alors la séquence :

$$\overline{[v\,;\mu_{|v}]} \quad \equiv \quad [\texttt{Blk}(\#1, 0)\,;\, \texttt{Blk}(23, \texttt{Blk}(\#2, 0))\,;\, \texttt{Blk}(\#1, 0)]$$

Le domaine de $\mu_{|v}$ ne contient pas l'étiquette ℓ_2 et la séquence des valeurs sérialisées est $[\overline{v}\,;\,\overline{\mu}(\ell_1)\,;\,\overline{\mu}(\ell_3)]$. Ainsi, les indices $\#1$ et $\#2$ représentent respectivement la sérialisation des étiquettes ℓ_1 et ℓ_3.

Avant de pouvoir écrire la règle d'évaluation de la primitive de sérialisation, la représentation des valeurs sérialisées est ajoutée à la syntaxe des valeurs :

$$\begin{array}{rcll} v & ::= & \ldots & \\ & | & \overline{[v\,;\mu]} & \textit{représentation externe de la valeur } v \textit{ et du tas } \mu \end{array}$$

La règle d'évaluation de la primitive `marshall` t consiste alors à évaluer l'argument t puis à sérialiser la valeur obtenue, en extrayant la partie utile du tas. La règle d'évaluation infinie en découle immédiatement : si l'évaluation de l'argument boucle, l'évaluation de la primitive boucle.

$$\frac{\mu, \rho \vdash t \Rightarrow v/\mu'}{\mu, \rho \vdash \texttt{marshall}\ t \Rightarrow \overline{[v\,;\mu'_{|v}]}/\mu'}\ [\textsc{Eval-Marshal}] \qquad \frac{\mu, \rho \vdash t \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash \texttt{marshall}\ t \overset{\infty}{\Rightarrow}}\ [\textsc{EvalInf-Marshal}]$$

Dans un langage réaliste, une valeur sérialisée est représentée par une suite d'octets. Par exemple, la fonction de sérialisation d'OCaml a le type : $\forall\alpha.\ \alpha \to \texttt{String}$. Dans notre mini-langage sans chaîne de caractères, on ajoute à l'ensemble des constructeurs de types le constructeur `Serial` d'arité 0 et unique type pour toutes les valeurs sérialisées.

$$\Psi \vdash \overline{[v\,;\mu]} : \texttt{Serial} \quad [\text{V-}\textsc{Serial}]$$

Ainsi, le type de retour de la primitive de sérialisation sera systématiquement `Serial`, indépendamment du type de son argument.

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash \texttt{marshall}\ t : \texttt{Serial}}\ [\text{ML-}\textsc{Marshal}]$$

Désérialisation non sûre L'opération de désérialisation réalise l'opération inverse. Une primitive de désérialisation non sûre pourrait être typée :

$$\frac{\Gamma \vdash t : \texttt{Serial}}{\Gamma \vdash \texttt{unsafe\_unmarshall}\ t : \tau}$$

Et la règle d'évaluation correspondante consisterait simplement à reconstruire la valeur sérialisée, notamment en réallouant dans le tas les références sérialisées.

$$\frac{\mu, \rho \vdash t \Rightarrow \overline{[v\,;\mu']}/\mu'' \qquad dom(\mu'') \# dom(\mu')}{\mu, \rho \vdash \texttt{unsafe\_unmarshall}\ t \Rightarrow v/(\mu'' \oplus \mu')}$$

où $dom(\mu'') \# dom(\mu')$ signifie que les domaines de μ'' et μ' sont disjoints ; ainsi, la notation $(\mu'' \oplus \mu')$ représente bien la réallocation dans le tas μ'' de toutes les références sérialisées dans $\overline{[v\,;\mu']}$. Ceci correspond au comportement des fonctions de (dé)sérialisation d'OCaml, où lorsqu'une référence est sérialisée puis désérialisée dans le même programme, elle est dupliquée dans le tas. Mais, comme l'illustre l'exemple ci-dessous, cette primitive non sûre permet également d'écrire un programme bien typé ne s'évaluant pas.

Exemple 2.4.2. *Le terme ci-dessous est correctement typé avec le type* `Int`*, mais il s'évalue vers une erreur.*

$$\begin{array}{l}\texttt{let}\ s = \texttt{marshall}\ []\ \texttt{in}\\ \texttt{let}\ x = \texttt{unsafe\_unmarshall}\ s\ \texttt{in}\\ !(x) + 1\end{array}$$

En effet, la variable s a le type `Serial` *et la variable x est contrainte par son contexte d'utilisation* $!(\_) + 1$ *à être de type* `Ref(Int)`*. Au cours de l'évaluation la valeur liée à x sera l'entier 0, représentant la liste vide. Or, cette valeur n'est pas compatible avec le type de x :* `Ref(Int)` *car les seules valeurs compatibles avec le constructeur de types* `Ref(_)` *sont les étiquettes, voir les règles* [V-Ref] *et* [Eval-Access]*. La variable x ne peut pas être déréférencée et l'évaluation du programme produit une erreur.*

Désérialisation sûre Comme l'illustre l'exemple précédent, une incompatibilité entre la valeur désérialisée et le type attendu ne peut pas être détecté uniquement par le système de types. Il est nécessaire d'effectuer au moment de la désérialisation un test de compatibilité. Pour permettre au programmeur de prendre en compte le résultat de ce test, une fonction de désérialisation sûre peut par exemple renvoyer une valeur appartenant au type `Option(_)` : en cas d'échec du test de compatibilité la valeur `None` est renvoyée, en cas de réussite la valeur désérialisée est encapsulée dans le constructeur `Some(_)`.

D'un autre côté, pour faire le lien entre le type de retour associé par le système de types à la fonction de désérialisation et le type utilisé par le test de compatibilité, la fonction générique de désérialisation doit être paramétrée, d'une manière ou d'une autre, par le type attendu. Dans le formalisme choisi ici, cela pourrait être représenté par une nouvelle construction syntaxique :

$$\begin{array}{rcl} t & ::= & \ldots \\ & & \mid \texttt{safe\_unmarshall}\ \tau\ t \end{array}$$

Cette construction pourrait alors être typée à l'aide de la règle suivante, où $gen(\tau)$ est une abréviation pour $gen(\emptyset, \tau) = \forall \bar{\alpha}.\, \tau$ avec $\bar{\alpha} = ftv(\tau)$:

$$\frac{\Gamma \vdash t : \texttt{Serial} \qquad gen(\tau) \preccurlyeq \tau'}{\Gamma \vdash \texttt{safe\_unmarshall}\ \tau\ t : \texttt{Option}(\tau')}$$

Cette généralisation permet de découpler les variables du monde statique, utilisées par le système de types, des variables du monde dynamique, utilisées par le test de compatibilité. Ce découplage permet préserver la stabilité du typage par substitution des variables de types (lemme 2.3.1). En utilisant une contrainte d'égalité à la place d'une contrainte d'instanciation, il aurait été nécessaire de substituer les variables de types apparaissant dans les programmes pour conserver un lemme équivalent ; cela introduirait une complexité supplémentaire, sans doute inutile.

Pour distinguer les cas d'échec des cas de réussite du test de compatibilité de la valeur désérialisée avec le type attendu, l'évaluation de la primitive de désérialisation est représentée par deux règles ci-dessous, où l'expression $\mu' : \Psi \vdash v : \tau$ représente le test de compatibilité.

$$\frac{\mu, \rho \vdash t \Rightarrow \overline{[v\,;\mu']}/\mu'' \qquad not(\mu' : \Psi \vdash v : \tau)}{\mu, \rho \vdash \texttt{safe\_unmarshall}\ \tau\ t \Rightarrow 0/\mu''}$$

$$\frac{\mu, \rho \vdash t \Rightarrow \overline{[v\,;\mu']}/\mu'' \qquad \mu' : \Psi \vdash v : \tau \qquad dom(\mu'') \,\#\, dom(\mu')}{\mu, \rho \vdash \texttt{safe\_unmarshall}\ \tau\ t \Rightarrow \texttt{Blk}(v, 0)/(\mu'' \oplus \mu')}$$

En cas d'échec du test de compatibilité, la valeur renvoyée représente le constructeur `None`. En cas de réussite, la valeur désérialisée est encapsulée dans un bloc représentant le constructeur `Some(_)` et comme dans la règle d'évaluation de la primitive de désérialisation non sûre, les références désérialisées sont réallouées sur le tas.

Type explicite Pour réaliser ce mécanisme de désérialisation sûre dans le compilateur OCaml, nous avons choisi de représenter explicitement les expressions de types dans le langage, à la manière du type `ty(_)` décrit à la section 1.3.3. Pour mimer ce choix, les règles d'évaluation et de typage de la primitive de désérialisation décrites ci-dessus doivent être légèrement adaptées et les syntaxes des termes et des valeurs doivent être étendues avec une construction représentant explicitement les types [1]. Pour la distinguer de la primitive `safe_unmarshall` τ t recevant syntaxiquement un type en premier argument, la primitive de désérialisation est simplement nommée `unmarshall` t t.

t ::= ...
| `unmarshall` t t — *primitive de désérialisation*
| `ty`(τ) — *représentation explicite des types*

t^v ::= ...
| `ty`(τ) — *représentation explicite des types*

v ::= ...
| `ty`(τ) — *représentation explicite des types*

1. Dans la mise en œuvre de ce mécanisme, les valeurs représentant des types seront projetées vers un type algébrique ne nécessitant pas l'ajout d'une représentation explicite pour l'environnement d'exécution.

Pour typer les termes et les valeurs de type, l'ensemble des constructeurs de types est étendu avec le constructeur $\texttt{Ty}(\_)$ d'arité 1. Ce constructeur permet d'un côté d'étendre le système de type des termes :

$$\frac{gen(\tau) \preccurlyeq \tau'}{\Gamma \vdash \texttt{ty}(\tau) : \texttt{Ty}(\tau')} \quad [\text{ML-Ty}]$$

et d'un autre côté d'étendre de système de types des valeurs :

$$\frac{gen(\tau) \preccurlyeq \tau'}{\Psi \vdash \texttt{ty}(\tau) : \texttt{Ty}(\tau')} \quad [\text{V-Ty}]$$

L'évaluation des termes représentant un type est immédiate.

$$\frac{}{\mu, \rho \vdash \texttt{ty}(\tau) \Rightarrow \texttt{ty}(\tau)/\mu} \quad [\text{Eval-Ty}]$$

Avec cette représentation explicite des types, la règle de typage de la primitive de désérialisation doit être légèrement adaptée :

$$\frac{\Gamma \vdash t_1 : \texttt{Ty}(\tau) \qquad \Gamma \vdash t_2 : \texttt{Serial}}{\Gamma \vdash \texttt{unmarshall}\ t_1\ t_2 : \texttt{Option}(\tau)} \quad [\text{ML-Unmarshal}]$$

Et pour finir, les règles d'évaluation proposées précédemment sont légèrement adaptées : le premier argument de la primitive doit s'évaluer vers une valeur représentant un type.

$$\frac{\mu, \rho \vdash t_1 \Rightarrow \texttt{ty}(\tau')/\mu' \qquad \mu', \rho \vdash t_2 \Rightarrow \overline{[v''\,;\mu'']}/\mu''' \qquad \mu'' : \Psi'' \vdash v'' : \tau' \qquad dom(\mu''') \# dom(\mu'')}{\mu, \rho \vdash \texttt{unmarshall}\ t_1\ t_2 \Rightarrow \texttt{Blk}(v'', 0)/(\mu''' \oplus \mu'')} \quad [\text{Eval-Unmarshall-Ok}]$$

$$\frac{\mu, \rho \vdash t_1 \Rightarrow \texttt{ty}(\tau')/\mu' \qquad \mu', \rho \vdash t_2 \Rightarrow \overline{[v''\,;\mu'']}/\mu''' \qquad \neg(\mu'' : \Psi'' \vdash v'' : \tau')}{\mu, \rho \vdash \texttt{unmarshall}\ t_1\ t_2 \Rightarrow 0/\mu'''} \quad [\text{Eval-Unmarshall-Fail}]$$

Les règles d'évaluation infinie sont construites selon le principe systématique décrit à la section 2.2.3.

$$\frac{\mu, \rho \vdash t_1 \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash \texttt{unmarshall}\ t_1\ t_2 \overset{\infty}{\Rightarrow}} \quad [\text{EvalInf-Unmarshall-Left}]$$

$$\frac{\mu, \rho \vdash t_1 \Rightarrow v/\mu' \qquad \mu', \rho \vdash t_2 \overset{\infty}{\Rightarrow}}{\mu, \rho \vdash \texttt{unmarshall}\ t_1\ t_2 \overset{\infty}{\Rightarrow}} \quad [\text{EvalInf-Unmarshall-Right}]$$

Primitives de communication Il est à noter que le simple ajout de primitives de (dé)sérialisation au langage étudié, permet la sérialisation et la désérialisation d'une donnée uniquement à l'intérieur d'un programme. Pour obtenir un système plus réaliste, il est nécessaire d'étendre le langage avec des primitives de communication, par exemple `read` : `Unit` → `Serial` et `write` : `Serial` → `Unit`, permettant d'échanger des valeurs sérialisées avec l'extérieur. Puisque toute la sûreté du mécanisme de sérialisation repose sur la vérification de compatibilité effectuée par primitive de désérialisation, il suffit que les primitives `read` et `write` vérifient les types proposés ci-dessus pour conserver la propriété de sûreté d'exécution. Il est inutile de formaliser ici la sémantique opérationnelle de ces primitives.

2.5 Correction du système de types

Cette section démontre la correction de ce système de types par rapport à la sémantique opérationnelle : tout programme bien typé ne peut pas s'évaluer vers une erreur. Cette démonstration se fait en deux étapes. Dans un premier temps, les lemmes 2.5.1 et 2.5.2, dits lemmes de préservation du typage, montrent que la valeur produite par l'évaluation d'un programme bien typé produit une valeur du même type que le programme. Dans un second temps, le lemme 2.5.3, dit lemme de progrès, montre que tous les programmes bien typés ne s'évaluant pas vers une valeur v s'évaluent à l'infini.

Le premier lemme de préservation du typage s'appliquera uniquement aux termes appartenant au sous-ensemble t^v. Il montrera conjointement que l'évaluation de ces termes ne modifie pas le tas. Le second lemme s'appliquera dans le cas général, il nécessite de montrer conjointement que le type des valeurs stockées dans le tas est préservé par l'évaluation. De plus, pour renforcer l'idée qu'il est suffisant de typer le tas avec des types clos, ce second lemme suppose que l'environnement de typage Γ et le type τ utilisés par l'hypothèse de typage $\Gamma \vdash t : \tau$, sont clos.

Les démonstrations des deux lemmes de préservation présentent peu d'originalité et les preuves ci-dessous détaillent principalement le cas des fermetures et des primitives de (dé)sérialisation. La preuve du lemme de progrès en présence de règles d'évaluation infinie définies par coinduction est une idée plus récente, ce document reproduit la preuve de Leroy et Grall [2009]. L'extension de cette preuve aux primitives de (dé)sérialisation est immédiate et ne présente aucune originalité.

Lemme 2.5.1 (Préservation (valeurs))**.** *Étant donné un environnement de typage des termes Γ, un terme t^v et un type τ tels que $\Gamma \vdash t^v : \tau$, étant donné un tas μ, un environnement de typage du tas Ψ tels que $\mu : \Psi$, étant donné un environnement d'évaluation ρ tel que $\Psi \vdash \rho : \Gamma$, alors il existe une valeur v telle que $\mu, \rho \vdash t \Rightarrow v/\mu$ et $\Psi \vdash v : \tau$.*

Preuve Par induction sur la structure du terme t^v.

Cas $t^v \equiv x$**.** En décomposant l'hypothèse de typage $\Gamma \vdash x : \tau$ par la règle [ML-Var], on obtient $x \in dom(\Gamma)$ et $\Gamma(x) \preccurlyeq \tau$. On peut alors appliquer la règle d'évaluation [Eval-Var] et vérifier $\mu, \rho \vdash t \Rightarrow \rho(x)/\mu$. En combinant ce résultat avec

l'hypothèse de typage de l'environnement $\Psi \vdash \rho : \Gamma$, on obtient en particulier $\Psi \vdash \rho(x) : \tau$.

Cas $t^v \equiv \lambda x.t$**.** La règle d'évaluation [Eval-Clos] s'applique et on vérifie $\mu, \rho \vdash t^v \Rightarrow \langle \lambda x.t, \rho \rangle / \mu$. L'hypothèse de typage $\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2$ et l'hypothèse de typage de l'environnement $\Psi \vdash \rho : \Gamma$ permettent directement d'appliquer la règle [V-Clos] pour conclure $\Psi \vdash \langle \lambda x.t, \rho \rangle : \tau_1 \to \tau_2$.

Cas $t^v \equiv \texttt{fix}\ f = \lambda\, x.t$**.** Similaire au cas des fonctions.

Cas des constructeurs constants : `true`, `false`, i, [] **et** `None`**.**
En suivant respectivement les règles [Eval-True], [Eval-False], [Eval-Int], [Eval-Nil] et [Eval-None], l'évaluation est immédiate vers un entier-mémoire. La règle de typage des entiers [V-Int] et respectivement les cas (Δ-True), (Δ-False), (Δ-Int), (Δ-Nil)et (Δ-None)de la relation Δ permettent de vérifier la préservation du typage.

Cas des constructeurs non constants : $(t_1^v,\, t_2^v)$**,** $t_1^v :: t_2^v$ **et** $\texttt{Some}(t_1^v)$**.**
Les trois cas étant similaires, seul le cas du couple sera détaillé. La décomposition de l'hypothèse de typage $\Gamma \vdash (t_1^v,\, t_2^v) : (\tau_1 \times \tau_2)$, permet de vérifier $\Gamma \vdash t_1^v : \tau_1$ et $\Gamma \vdash t_2^v : \tau_2$. Il est alors possible d'appliquer l'hypothèse d'induction aux sous-termes t_1^v et t_2^v et vérifier l'existence de valeurs v_1 et v_2 telles que d'une part $\mu, \rho \vdash t_1^v \Rightarrow v_1/\mu$ et $\mu, \rho \vdash t_2^v \Rightarrow v_2/\mu$ et d'autre part $\Psi \vdash v_1 : \tau_1$ et $\Psi \vdash v_2 : \tau_2$. La règle d'évaluation [Eval-Prod] et la règle de typage [V-Blk] associé au cas (∇-Prod) permettent alors de conclure $\mu, \rho \vdash (t_1^v,\, t_2^v) \Rightarrow \texttt{Blk}(v_1,\, v_2)/\mu$ et $\Psi \vdash \texttt{Blk}(v_1,\, v_2) : (\tau_1 \times \tau_2)$.

Cas des expressions de types : $\texttt{ty}(\tau')$**.**
En suivant la règle [Eval-Ty], l'évaluation est immédiate vers une valeur $\texttt{ty}(\tau')$. Les règles de typage [ML-Ty] et [V-Ty] étant similaires, la préservation du typage est immédiate.

Lemme 2.5.2 (Préservation)**.** *Étant donné un environnement de typage des termes* Γ *clos, un terme* t *et un type* τ *clos tels que* $\Gamma \vdash t : \tau$*, étant donné un tas* μ*, un environnement de typage du tas* Ψ *tels que* $\mu : \Psi$*, étant donné un environnement d'évaluation* ρ *tel que* $\Psi \vdash \rho : \Gamma$*, s'il existe une valeur* v *et un tas* μ' *tels que* $\mu, \rho \vdash t \Rightarrow v/\mu'$ *alors il existe un environnement* Ψ' *contenant* Ψ *et tel que* $\mu' : \Psi'$ *et* $\Psi' \vdash v : \tau$*.*

Preuve Par induction sur la dérivation d'évaluation.

Cas [Eval-Marshal]
$$\frac{\mu, \rho \vdash t \Rightarrow v/\mu'}{\mu, \rho \vdash \texttt{marshall}\ t \Rightarrow \overline{[v\,;\mu'_{|v}]}/\mu'}$$
L'hypothèse d'induction permet d'obtenir un environnement Ψ' tel que $\Psi \subseteq \Psi'$ et $\mu' : \Psi'$. La règle de typage [V-Serial] permet par ailleurs d'obtenir $\Psi' \vdash \overline{[v\,;\mu'_{|v}]} : \texttt{Serial}$.

Cas [Eval-Unmarshall-Ok]
$$\frac{\mu, \rho \vdash t_1 \Rightarrow \texttt{ty}(\tau')/\mu' \qquad \mu', \rho \vdash t_2 \Rightarrow \overline{[v''\,;\mu'']}/\mu''' \qquad \mu'' : \Psi'' \vdash v'' : \tau' \qquad dom(\mu''') \# dom(\mu'')}{\mu, \rho \vdash \texttt{unmarshall}\ t_1\ t_2 \Rightarrow \texttt{Blk}(v'', 0)/(\mu''' \oplus \mu'')}$$

En décomposant l'hypothèse de typage $\Gamma \vdash \texttt{unmarshall}\ t_1\ t_2 : \texttt{Option}(\tau)$, on obtient $\Gamma \vdash t_1 : \texttt{Ty}(\tau)$. L'hypothèse d'induction appliquée à la première prémisse $\mu, \rho \vdash t_1 \Rightarrow \texttt{ty}(\tau')/\mu'$ permet alors d'obtenir un environnement Ψ' tel que $\Psi \subseteq \Psi'$ et $\mu' : \Psi'$ et de vérifier $\Psi' \vdash \texttt{ty}(\tau') : \texttt{Ty}(\tau)$. La décomposition de ce jugement par la règle [V-Ty] permet d'obtenir une substitution θ tel que $\theta(\tau') = \tau$. Le test de compatibilité $\mu'' : \Psi'' \vdash v'' : \tau'$ et le lemme d'instanciation (lemme 2.3.6) permettent alors de vérifier $\Psi'' \vdash v'' : \tau$. Les règles [V-Blk] et (∇-Some) permettent ensuite d'étendre ce résultat en $\Psi'' \vdash \texttt{Blk}(v'', 0) : \texttt{Option}(\tau)$. Par ailleurs, l'hypothèse d'induction appliquée à la deuxième prémisse $\mu', \rho \vdash t_2 : \overline{[v'' ; \mu'']}/\mu'''$ permet d'obtenir un environnement Ψ''' tel que $\Psi \subseteq \Psi' \subseteq \Psi'''$ et $\mu''' : \Psi'''$. Les domaines de μ'' et de μ''' étant disjoints, on peut poser $\Psi'''' = \Psi''' \cup \Psi''$ et conclure facilement que $\Psi \subseteq \Psi''''$, $(\mu''' \oplus \mu'') : \Psi''''$ et $\Psi'''' \vdash \texttt{Blk}(v, 0) : \texttt{Option}(\tau)$.

[Eval-Unmarshall-Fail]

Cas $$\frac{\mu, \rho \vdash t_1 \Rightarrow \texttt{ty}(\tau')/\mu' \qquad \mu', \rho \vdash t_2 \Rightarrow \overline{[v'' ; \mu'']}/\mu''' \qquad \neg(\mu'' : \Psi'' \vdash v'' : \tau')}{\mu, \rho \vdash \texttt{unmarshall}\ t_1\ t_2 \Rightarrow 0/\mu'''}$$

L'hypothèse d'induction appliquée successivement à $\mu, \rho \vdash t_1 \Rightarrow \texttt{ty}(\tau')/\mu'$ et $\mu', \rho \vdash t_2 : \overline{[v'' ; \mu'']}/\mu'''$ permet d'obtenir un environnement Ψ''' tel que $\Psi \subseteq \Psi'''$ et $\mu''' : \Psi'''$. Les règles [V-Int] et (Δ-None) permettent par ailleurs d'obtenir $\Psi''' \vdash 0 : \texttt{Option}(\tau)$.

[Eval-Var]

Cas $$\frac{x \in dom(\rho)}{\mu, \rho \vdash x \Rightarrow \rho(x)/\mu}$$

En décomposant l'hypothèse de typage $\Gamma \vdash x : \tau$ par la règle [ML-Var], on obtient $\Gamma(x) \preccurlyeq \tau$; en combinant ce résultat avec l'hypothèse de typage de l'environnement $\Psi \vdash \rho : \Gamma$, on obtient en particulier $\Psi \vdash \rho(x) : \tau$.

[Eval-Clos]

Cas $$\frac{}{\mu, \rho \vdash \lambda x.t \Rightarrow \langle \lambda x.t, \rho \rangle/\mu}$$

L'hypothèse de typage $\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2$ et l'hypothèse de typage de l'environnement $\Psi \vdash \rho : \Gamma$ permettent directement d'appliquer la règle [V-Clos] pour conclure $\Psi \vdash \langle \lambda x.t, \rho \rangle : \tau_1 \to \tau_2$.

[Eval-App]

Cas $$\frac{\mu, \rho \vdash t_1 \Rightarrow \langle \lambda x.t', \rho' \rangle/\mu' \qquad \mu', \rho \vdash t_2 \Rightarrow v_2/\mu'' \qquad \mu'', \rho' \oplus \{x \mapsto v_2\} \vdash t' \Rightarrow v/\mu'''}{\mu, \rho \vdash t_1\ t_2 \Rightarrow v/\mu'''}$$

En décomposant l'hypothèse de typage $\Gamma \vdash t_1\ t_2 : \tau$ par la règle [ML-App], on obtient un type τ' tel que $\Gamma \vdash t_1 : \tau' \to \tau$ et $\Gamma \vdash t_2 : \tau'$. Si ce type n'est pas clos et puisque l'environnement de typage Γ est clos par hypothèse, le lemme 2.3.1 permet de le clore. Comme dans le cas de la primitive de désérialisation, l'application successive de l'hypothèse d'induction aux deux premières prémisses, permet

d'obtenir un environnement de typage du tas Ψ'' tel que $\Psi \subseteq \Psi''$ et $\mu'' : \Psi''$; et de vérifier $\Psi'' \vdash \langle \lambda x.t', \rho' \rangle : \tau' \to \tau$ et $\Psi'' \vdash v_2 : \tau'$. La décomposition par la règle [V-Clos] du premier de ces jugements de typage permet d'obtenir un environnement de typage Γ' tel que $\Gamma' \oplus \{x \mapsto \tau'\} \vdash t' : \tau$ et $\Psi'' \vdash \rho' : \Gamma'$. Les types τ et τ' étant clos, on peut appliquer les lemmes 2.3.1 et 2.3.6 pour clore l'environnement Γ'. Ceci permet d'appliquer l'hypothèse d'induction à la troisième prémisse de la règle d'évaluation et de conclure l'existence d'un environnement de typage du tas Ψ''' tel que $\Psi \subseteq \Psi'''$, $\mu''' : \Psi'''$ et $\Psi''' \vdash v : \tau$.

Cas [Eval-RecClos]
$$\frac{}{\mu, \rho \vdash \texttt{fix}\ f = \lambda\, x.t \Rightarrow \langle\!\langle f = \lambda x.t, \rho \rangle\!\rangle / \mu}$$

et [Eval-RecApp]
$$\frac{\mu, \rho \vdash t_1 \Rightarrow \langle\!\langle f = \lambda x.t', \rho' \rangle\!\rangle / \mu' \qquad \mu', \rho \vdash t_2 \Rightarrow v_2 / \mu'' \qquad \mu'', \rho' \oplus \{f \mapsto \langle\!\langle f = \lambda x.t', \rho' \rangle\!\rangle\,;\, x \mapsto v_2\} \vdash t' \Rightarrow v / \mu'''}{\mu, \rho \vdash t_1\ t_2 \Rightarrow v / \mu'''}$$

Ces deux cas se montrent de manière similaire aux cas non récursifs.

Cas [Eval-LetIn]
$$\frac{\mu, \rho \vdash t_x \Rightarrow v_x / \mu' \qquad \mu', \rho \oplus \{x \mapsto v_x\} \vdash t \Rightarrow v / \mu''}{\mu, \rho \vdash \texttt{let}\ x = t_x\ \texttt{in}\ t \Rightarrow v / \mu''}$$

On distingue deux cas selon la règle de typage utilisée.

Cas [ML-LetV]
$$\frac{\Gamma \vdash t_x^v : \tau_x \qquad \Gamma \oplus \{x \mapsto gen(\Gamma, \tau_x)\} \vdash t : \tau}{\Gamma \vdash \texttt{let}\ x = t_x^v\ \texttt{in}\ t : \tau}$$

Le lemme 2.5.1 permet de vérifier $\mu' \equiv \mu$ et $\Psi \vdash v_x : \tau_x$. On peut alors appliquer le lemme 2.3.6 pour généraliser le type de v_x et vérifier $\rho \oplus \{x \mapsto v_x\} : \Gamma \oplus \{x : gen(\Gamma, \tau_x)\}$. D'autre part, par hypothèse, l'environnement de typage Γ est clos, on a alors $gen(\Gamma, \tau_x) = gen(\emptyset, \tau_x)$ et l'environnement $\Gamma \oplus \{x : gen(\Gamma, \tau_x)\}$ est lui aussi clos. On peut maintenant appliquer l'hypothèse d'induction à la seconde prémisse de la règle d'évaluation $\mu', \rho \oplus \{x \mapsto v_x\} \vdash t : v / \mu''$ pour conclure à l'existence d'un environnement Ψ'' contenant Ψ et tel que $\mu'' : \Psi''$ et $\Psi'' \vdash v : \tau$.

Cas [ML-Let]
$$\frac{\Gamma \vdash t_x : \tau_x \qquad \Gamma \oplus \{x \mapsto \tau_x\} \vdash t : \tau}{\Gamma \vdash \texttt{let}\ x = t_x\ \texttt{in}\ t : \tau}$$

Comme dans le cas de l'application et puisque l'environnement de typage Γ est clos par hypothèse, si le type τ_x n'est pas clos, le lemme 2.3.1 permet de le clore. Il est alors possible d'appliquer l'hypothèse d'induction à la prémisse de la règle d'évaluation $\mu, \rho \vdash t_x : v_x / \mu'$. On obtient un environnement Ψ' tel que $\Psi \subseteq \Psi'$ et $\mu' : \Psi'$ et tel que $\Psi' \vdash v_x : \tau_x$. On vérifie immédiatement $\Psi' \vdash \rho \oplus \{x \mapsto v_x\} : \Gamma \oplus \{x : \tau_x\}$. On peut alors appliquer l'hypothèse d'induction à la seconde prémisse $\mu', \rho \oplus \{x \mapsto v_x\} \vdash t : v / \mu''$ pour conclure à l'existence d'un environnement Ψ'' contenant Ψ et tel que $\mu'' : \Psi''$ et $\Psi'' \vdash v : \tau$.

Cas $\dfrac{\mu, \rho \vdash t \Rightarrow v/\mu' \qquad \ell \notin dom(\mu')}{\mu, \rho \vdash \texttt{ref}(t) : \ell/\mu' \oplus \{\ell \mapsto v\}}$ [Eval-Ref]

Le type $\texttt{Ref}(\tau)$ est clos par hypothèse. Par hypothèse d'induction, on obtient un tas Ψ' contenant Ψ et tel que $\mu' : \Psi'$ et $\Psi' \vdash v : \tau$. On peut alors étendre Ψ' avec l'association $\{\ell : \tau\}$ pour typer correctement le nouveau tas et la valeur résultante.

Cas des constructeurs de données.

Dans le cas des constructeurs non constants, l'hypothèse d'induction doit être successivement appliquée à chacun des arguments pour obtenir l'environnement de typage nécessaire pour typer le tas résultant. Ensuite, et comme dans la preuve du lemme 2.5.1, il faut vérifier qu'à chaque règle d'évaluation correspond un cas dans le système de types des valeurs, permettant de typer correctement la valeur résultante.

Autre cas.

Encore une fois l'hypothèse d'induction s'applique successivement aux différentes prémisses pour obtenir l'environnement de typage nécessaire pour typer le tas résultant. La seule règle modifiant le tas est le cas [Eval-Modify] ; l'hypothèse de typage garantit alors que cette modification du tas n'en modifie pas le type. Dans le cas des règles d'évaluation des opérations arithmétiques [Eval-Arith] et de l'égalité [Eval-Eq] et [Eval-NotEq], il est ensuite immédiat de vérifier que les valeurs renvoyées sont respectivement entière et booléennes. Pour les règles d'évaluation de la conditionnelle : [Eval-IfTrue] et [Eval-IfFalse], de filtrage de listes : [Eval-Case-Nil] et [Eval-Case-Cons], et de la lecture d'une référence [Eval-Access], l'hypothèse d'induction appliquée à la dernière prémisse suffit pour garantir la correction du type de la valeur renvoyée. Dans le cas des projections [Eval-Fst] et [Eval-Snd], il est en plus nécessaire de décomposer la règle de typage $\Psi \vdash \texttt{Blk}(v_1, v_2) : (\tau_1 \times \tau_2)$.

□

Lemme 2.5.3 (Progrès). *Étant donné un environnement de typage des termes Γ, un terme t et un type τ tels que $\Gamma \vdash t : \tau$, étant donné un tas μ, un environnement de typage du tas Ψ tels que $\mu : \Psi$, étant donné un environnement d'évaluation ρ tel que $\Psi \vdash \rho : \Gamma$, si $\mu, \rho \vdash t \not\Rightarrow$ alors $\mu, \rho \vdash t \overset{\infty}{\Rightarrow}$.*

Preuve Par coinduction sur la structure de terme t.

Cas $t \equiv x$ Le typage garantit l'appartenance de la variable x à l'environnement ρ. L'évaluation est donc systématique et le programme ne boucle pas. L'hypothèse $\mu, \rho \vdash t \not\Rightarrow$ n'est alors jamais satisfaite.

Cas $t \equiv \lambda x.t'$ **et** $t \equiv \texttt{ty}(\tau)$ De la même manière : une fonction s'évalue systématiquement vers une fermeture et une expression de type s'évalue immédiatement ; leur évaluation ne peut pas boucler.

Cas $t \equiv t_1\ t_2$

- Si $\mu, \rho \vdash t_1 \not\Rightarrow$, alors par hypothèse de coinduction $\mu, \rho \vdash t_1 \overset{\infty}{\Rightarrow}$. La règle [EvalInf-App-Left] permet d'en déduire $\mu, \rho \vdash t_1\ t_2 \overset{\infty}{\Rightarrow}$. De même, si $\mu, \rho \vdash t_1 \Rightarrow v_1/\mu'$ et $\mu', \rho \vdash t_2 \not\Rightarrow$, on applique l'hypothèse de coinduction à la seconde condition et la règle [EvalInf-App-Right], permet d'en déduire $\mu, \rho \vdash t_1\ t_2 \overset{\infty}{\Rightarrow}$.
- Dans les autres cas, les deux sous-termes s'évaluent : $\mu, \rho \vdash t_1 \Rightarrow v_1/\mu'$ et $\mu', \rho \vdash t_2 \Rightarrow v_2/\mu''$. Par décomposition par la règle [ML-App] de l'hypothèse de typage $\Gamma \vdash t_1\ t_2 : \tau$, il existe alors un type τ' tel que $\Gamma \vdash t_1 : \tau' \to \tau$ et tel que $\Gamma : t_2 : \tau'$. Le lemme de correction du typage permet alors de vérifier l'existence d'un environnement de typage Ψ'', tel que $\mu'' : \Psi''$ et de vérifier $\Psi'' \vdash v_1 : \tau' \to \tau$ et $\Psi'' \vdash v_2 : \tau'$. Le système de types des valeurs impose alors que la valeur v_1 est une fermeture, récursive ou non. Le raisonnement étant similaire dans les deux cas, on suppose dans la suite que $v_1 \equiv \langle \lambda x.t', \rho' \rangle$.
  - S'il existe une valeur v telle que $\mu'', \rho' \oplus \{x \mapsto v_2\} \vdash t' \Rightarrow v$, alors l'application $t_1\ t_2$ s'évalue correctement et l'hypothèse $\mu, \rho \vdash t \not\Rightarrow$ n'est pas satisfaite.
  - Sinon, on peut appliquer l'hypothèse de coinduction pour en déduire $\mu'', \rho' \oplus \{x \mapsto v_2\} \vdash t' \overset{\infty}{\Rightarrow}$ et la règle [EvalInf-App] permet de conclure $\mu, \rho \vdash t_1\ t_2 \overset{\infty}{\Rightarrow}$.

Autres cas. Les autres cas se démontrent de la même manière que le cas de l'application : si toutes les prémisses s'évaluent correctement, le programme ne boucle pas. Si l'une des prémisses ne s'évalue pas, l'hypothèse de coinduction permet de démontrer que le programme boucle. Lorsqu'une construction syntaxique admet plusieurs règles de dérivation, il est aussi nécessaire de vérifier que l'ensemble des valeurs autorisées par le typage pour un des arguments est pris en compte : par exemple dans le cas de la conditionnelle, les valeurs autorisées par le typage pour le premier argument, de type `Bool`, sont 0 et 1 ; ces deux valeurs correspondent respectivement aux règles d'évaluations [Eval-IfFalse] et [Eval-IfTrue]. Dans de la fonction de désérialisation, il est aussi nécessaire de vérifier que le test de compatibilité de la valeur désérialisée avec le type attendu est décidable ; le chapitre 3 traite en détail cette question et propose un algorithme de décision.

□

Les lemmes de déterminisme, de préservation du typage et de progrès permettent alors de démontrer le théorème suivant.

Théorème 2.5.4 (Correction du typage)**.** *Étant donné un environnement de typage des termes Γ, un terme t et un type τ tels que $\Gamma \vdash t : \tau$, étant donné un tas μ, un environnement de typage du tas Ψ tels que $\mu : \Psi$, étant donné un environnement d'évaluation ρ tel que $\Psi \vdash \rho : \Gamma$, on a alors de manière exclusive :*

- *il existe un unique couple v/μ', tel que $v/\mu' : \tau$ et $\mu', \rho \vdash t \Rightarrow v/\mu'$;*
- *$\mu, \rho \vdash t \overset{\infty}{\Rightarrow}$.*

Le théorème 2.2.6 permet d'en déduire que si un programme t est correctement typé, son évaluation ne provoque pas d'erreur : $\emptyset, \emptyset \vdash t \not\Rightarrow \texttt{err}$.

Type universel Il est à noter que la présence ou non du type universel $\star$ dans le système de types des valeurs ne modifie en rien les énoncés et les preuves des lemmes et théorèmes de cette section.

Chapitre 3

Graphes-mémoire compatibles

Le chapitre précédent a décrit la sémantique opérationnelle et le typage d'un langage de programmation fonctionnel et impératif. Il a montré ensuite que le typage garantit la sûreté d'exécution. Le langage de valeurs utilisé pour décrire la sémantique opérationnelle se veut proche de la représentation mémoire du compilateur OCaml : les constructeurs de données y sont représentés par des entiers ou des blocs. Pour montrer la correction du typage, l'ensemble des valeurs compatibles avec un type τ a été caractérisé à l'aide d'un système de types. Cette caractérisation a aussi permis d'étendre le langage avec une primitive de désérialisation sûre. Elle est réalisée par les deux opérations suivantes : la reconstruction du graphe mémoire de la valeur, en particulier la réallocation des références sérialisées, puis la vérification de l'appartenance de la valeur reconstruite à l'ensemble des types compatibles avec le type attendu.

Nous nous attachons dans ce chapitre a décrire un algorithme réalisant cette vérification. Pour cela, le langage de valeurs utilisé pour décrire la sémantique opérationnelle n'est pas encore assez réaliste. Car contrairement à la représentation mémoire utilisée par le compilateur OCaml, ce langage de valeurs distingue explicitement les valeurs modifiables des valeurs non modifiables et n'explicite pas le partage de valeurs introduit par la construction `let _ = _ in _` ou par l'abstraction fonctionnelle. Ce chapitre commence par définir un langage de valeurs plus proche de la notion de graphe mémoire, qui servira comme donnée d'entrée pour notre algorithme de vérification de type. En cas d'ambiguïté, le langage de valeurs utilisé pour définir la sémantique opérationnelle sera appelé *langage de valeurs* v et le langage plus réaliste utilisé pour décrire l'algorithme de vérification sera appelé *langage de valeurs* w [1].

Le langage de valeurs w est lui aussi muni d'un système de types permettant de caractériser les valeurs acceptables par la fonction de désérialisation. Pour démontrer la correction de ce système de types, il est possible de redéfinir toute la sémantique opérationnelle du langage de programmation directement à l'aide de graphes-mémoire. Ceci dit, cette présentation est plus verbeuse que celle choisie au chapitre précédent et introduirait une complexité vraisemblablement inutile du point de vue du problème de la désérialisation. La correction de ce système de types est justifiée partiellement en définissant une fonction d'expansion d'une valeur w bien typée vers une valeur v de même type. Cette fonction d'expansion est telle que si un programme s'évalue en pratique vers une valeur w, alors ce programme s'évalue dans la sémantique opérationnelle du chapitre précédent vers une valeur v correspondant à l'expansion de w. Cette fonction est évidemment non injective : plusieurs valeurs w peuvant représenter une même valeur v selon le degré de partage explicitement conservé par le compilateur.

La première partie de ce chapitre définit le langage w, son système de types et la fonction d'expansion. Les parties suivantes décrivent la formalisation d'un l'algorithme de vérification de type comme un système de réécriture muni d'une stratégie d'application. Le système de réécriture sera décrit par étapes : initialement en présence uniquement de blocs non modifiables et de partage sans cycle, puis la gestion des fermetures, celle du partage avec cycles et celle des références seront ajoutées successivement. Dans certains cas, il sera impossible de garantir la terminaison du système de réécriture sans renoncer à

1. w à la fois comme successeur alphabétique de v et comme l'initiale de l'anglais *word*.

sa complétude par rapport au système de types du langage w. La fin du chapitre étudiera des restrictions de ce système de types et leurs liens avec le système de types du langage source, pour lesquelles il sera possible de garantir terminaison et complétude.

3.1 Graphe-mémoire

Le langage de valeurs v contenait une représentation de toutes les valeurs nécessaires à la sémantique opérationnelle : des entiers-mémoire, des blocs et des fermetures, ainsi que des étiquettes désignant des éléments dans un tas et utiles pour représenter les références. Cette formalisation est trop abstraite pour décrire l'algorithme de vérification de types. Le langage de valeurs w défini ci-dessous se rapproche de la réalité en représentant explicitement l'allocation des valeurs autres que les entiers-mémoire.

$$
\begin{array}{rcll}
w & ::= & i & \textit{entier-mémoire} \\
 & | & \ell & \textit{adresse-mémoire} \\
h & ::= & \texttt{Blk}(w,\, w) & \textit{bloc} \\
 & | & \langle \lambda x.t, \rho \rangle & \textit{fermeture} \\
 & | & \langle\!\langle f = \lambda x.t, \rho \rangle\!\rangle & \textit{fermeture récursive}
\end{array}
$$

Dans ce formalisme, les environnements d'évaluation ρ contiennent des valeurs w et les tas μ contiennent des valeurs allouées h. Pour distinguer les tas manipulés par les langages v et w, les symboles ℓ du premier langage seront toujours appelés étiquettes, ceux du second langage seront appelés adresses-mémoire.

Dans ce langage de valeurs, les références sont encapsulées dans des blocs. Autrement dit, une référence sur une valeur w est représentée par une adresse-mémoire ℓ telle que $\mu(\ell) = \texttt{Blk}(w,\, 0)$ [2]. La seule opération de modification du tas à être alors autorisée est la modification du premier élément d'un bloc. Le système de types garantit que seuls les blocs représentant une référence pourront être modifiés par l'évaluation d'un programme.

L'allocation des blocs dans le tas permet de conserver le partage de valeurs introduit par la construction `let _ = _ in _` et par l'abstraction fonctionnelle : lorsque la valeur liée à une variable dans un environnement d'évaluation est une adresse-mémoire, seule cette adresse est dupliquée à chaque utilisation de la variable.

Exemple 3.1.1. *Le programme suivant :*

$$
\begin{array}{l}
\texttt{let}\, r = \texttt{ref}(1)\, \texttt{in} \\
\texttt{let}\, l = \texttt{None} :: [] \,\texttt{in} \\
(\texttt{Some}(r) :: l, \texttt{Some}(2) :: l)
\end{array}
$$

s'évalue dans le langage de valeurs v vers le couple v/μ :

$$
\begin{array}{rcl}
v & \equiv & \texttt{Blk}(\texttt{Blk}(\texttt{Blk}(\ell,\, 0), \texttt{Blk}(0,\, 0)),\, \texttt{Blk}(\texttt{Blk}(2,\, 0), \texttt{Blk}(0,\, 0))) \\
\mu & \equiv & \{\ell \mapsto 1\}
\end{array}
$$

2. Pour simplifier la présentation et comme dans le cas du constructeur `Some(_)`, on suppose que tous les blocs sont de taille deux.

Dans le langage de valeurs w, le résultat de l'évaluation pourrait être le couple w/μ' ci-dessous, où l'adresse-mémoire ℓ_0 représente l'encapsulation de la référence r dans un bloc ; l'adresse-mémoire ℓ_1 représente la variable l ; et les adresses-mémoire ℓ_3 et ℓ_5 représentent respectivement l'évaluation de $\texttt{Some}(r) :: l$ *et* $\texttt{Some}(2) :: l$.

$$
\begin{array}{lcll}
w & \equiv & \ell_6 & \\
\mu' & \equiv & \{ \quad \ell_0 & \mapsto \quad \texttt{Blk}(1,\,0) \quad ; \\
 & & \ell_1 & \mapsto \quad \texttt{Blk}(0,\,0) \quad ; \\
 & & \ell_2 & \mapsto \quad \texttt{Blk}(\ell_0,\,0) \quad ; \\
 & & \ell_3 & \mapsto \quad \texttt{Blk}(\ell_2,\,\ell_1) \quad ; \\
 & & \ell_4 & \mapsto \quad \texttt{Blk}(2,\,0) \quad ; \\
 & & \ell_5 & \mapsto \quad \texttt{Blk}(\ell_4,\,\ell_1) \quad ; \\
 & & \ell_6 & \mapsto \quad \texttt{Blk}(\ell_3,\,\ell_5) \quad \}
\end{array}
$$

3.1.1 Typage d'un graphe-mémoire

Le langage de valeurs w est lui aussi muni d'un système de types. Ce système est construit de telle sorte que tout couple w/μ bien typé puisse être projeté vers un couple v/μ de même type. Le chapitre précédent a montré que dans le langage v où le tas contient uniquement des valeurs modifiables, il suffit d'utiliser un environnement de typage associant des types monomorphes à chacune des étiquettes. Dans le langage w, le tas peut aussi contenir des valeurs non modifiables et polymorphes et il est alors nécessaire d'utiliser des environnements de typage du tas plus généraux.

Dans un compilateur réaliste, le partage est introduit uniquement par les différentes occurrences d'une même variable ; le système de types garantissant alors que toutes les utilisations d'une variable sont typées avec des instances d'un même schéma de type. Pour pouvoir typer l'ensemble des graphes-mémoire des programmes issus d'un tel compilateur, il est donc suffisant que le système de types du langage w accepte de typer une valeur partagée à l'aide d'un schéma de type. De la même manière qu'il est suffisant dans le langage v de typer les valeurs modifiables avec un type monomorphe, il est suffisant que ces schémas de type soient clos. Ainsi, les environnements de typage du tas dans le langage w associeront à chaque adresse-mémoire ℓ un type τ, dont toutes les variables libres sont implicitement quantifiées universellement. Ces environnements de typage seront notés Ψ.

Règles de dérivation Les jugements de typage seront notés $\Psi \vdash w : \tau$ et $\Psi \vdash h : \tau$. Les règles de typage des valeurs w seront nommées [W-...] et celles des valeurs allouées h seront nommées [H-...]. La règle de typage des entiers est similaire à la règle de typage des entiers du langage de valeurs v. Celle des adresses-mémoire vérifie que le type attendu pour une adresse ℓ est une instance du type contenu dans l'environnement de typage.

$$\frac{i \vartriangle \tau}{\Psi \vdash i : \tau} \quad [\text{W-Int}] \qquad\qquad \frac{\Psi(\ell) \preccurlyeq \tau}{\Psi \vdash \ell : \tau} \quad [\text{W-Label}]$$

Les règles de typage des valeurs allouées sont elles aussi similaires à celles du langage v. Cependant, les valeurs contenues dans un bloc ou dans les environnements capturés par une fermeture étant syntaxiquement restreintes aux entiers et aux adresses-mémoire, la dérivation de typage d'une valeur allouée sera toujours de profondeur deux.

$$\frac{[\tau_1\,;\,\tau_2]\ \nabla\ \tau \qquad \Psi \vdash w_1 : \tau_1 \qquad \Psi \vdash w_2 : \tau_2}{\Psi \vdash \texttt{Blk}(w_1,\, w_2) : \tau} \quad \text{[H-Blk]}$$

$$\frac{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2 \qquad \Psi \vdash \rho : \Gamma}{\Psi \vdash \langle \lambda x.t, \rho \rangle : \tau_1 \to \tau_2} \quad \text{[H-Clos]}$$

$$\frac{\Gamma \vdash \texttt{fix}\ f = \lambda\, x.t : \tau_1 \to \tau_2 \qquad \Psi \vdash \rho : \Gamma}{\Psi \vdash \langle\!\langle f = \lambda x.t, \rho \rangle\!\rangle : \tau_1 \to \tau_2} \quad \text{[H-RecClos]}$$

La relation ∇ doit être étendue pour prendre en compte l'encapsulation des références dans un bloc.

$$[\tau\,;\,\texttt{Unit}] \ \nabla\ \texttt{Ref}(\tau) \ (\nabla\text{-Ref})$$

Dans ce système la compatibilité d'un tas μ avec un environnement de typage Ψ doit être étendue avec une condition annexe pour les références. Ainsi, on notera $\mu : \Psi$ ssi : (1) les adresses-mémoire du domaine de μ et celles apparaissant dans son image (notée $fl(img(\mu))$) appartiennent au domaine de Ψ ; (2) toutes les valeurs de μ sont compatibles avec les types de Ψ, c'est-à-dire : $\forall \ell \in dom(\mu)$, $\Psi \vdash \mu(\ell) : \Psi(\ell)$; et (3) toutes les références de Ψ sont monomorphes, c'est-à-dire : $\forall \ell \in dom(\Psi)$ si $\Psi(\ell) = \texttt{Ref}(\tau)$ alors $fv(\tau) = \emptyset$.

La condition (a) sur les domaines de μ et de Ψ permet d'utiliser cette notation aussi bien pour typer un *tas clos*, c'est-à-dire tel que $fl(img(\mu)) \subseteq dom(\mu)$, que pour typer un *fragment de tas*. Dans le second cas, l'environnement de typage Ψ doit aussi apporter des hypothèses de typage pour les adresses-mémoire référencées mais non définies par μ. En présence d'un fragment de tas, on notera souvent $\mu : \Psi \oplus \Psi'$ avec $dom(\Psi') = dom(\mu)$. La notation $\Psi \oplus \Psi'$ y représente l'extension de l'environnement Ψ par Ψ', un environnement de domaine $dom(\Psi) \cup dom(\Psi')$ et tel que :

$$(\Psi \oplus \Psi')(\ell) = \begin{cases} \Psi'(\ell) & si\ \ell \in dom(\Psi') \\ \Psi(\ell) & sinon \end{cases}$$

Exemple 3.1.2. *Le tas μ est compatible avec les environnements de typage Ψ et Ψ' :*

$$\begin{array}{lcl} \mu & \equiv & \{\ell_0 \mapsto \texttt{Blk}(\ell_1, \ell_1) \ ;\ \ell_1 \mapsto \texttt{Blk}(0,0)\} \\ \Psi & \equiv & \{\ell_0 : (\texttt{List}(\texttt{Int}) \times \texttt{List}(\texttt{Int})) \ ;\ \ell_1 : \texttt{List}(\texttt{Int})\} \\ \Psi' & \equiv & \{\ell_0 : (\texttt{Ref}(\texttt{Int}) \times \texttt{Ref}(\texttt{Int})) \ ;\ \ell_1 : \texttt{Ref}(\texttt{Int})\} \end{array}$$

En considérant la valeur ℓ_1 comme polymorphe en possédant les types $\texttt{List}(\texttt{List}(\texttt{Int}))$ $\texttt{List}(\texttt{List}(\texttt{Bool}))$*, le tas μ est aussi compatible avec l'environnement de typage Ψ'' :*

$$\begin{array}{lcl} \Psi'' & \equiv & \{\ \ell_0 : (\texttt{List}(\texttt{List}(\texttt{Int})) \times \texttt{List}(\texttt{List}(\texttt{Bool}))) \ ; \\ & & \ \ell_1 : \texttt{List}(\texttt{List}(\alpha))\ \} \end{array}$$

En effet, d'un côté le type $\texttt{List}(\texttt{List}(\alpha))$ *est plus général que les types* $\texttt{List}(\texttt{List}(\texttt{Int}))$ *et* $\texttt{List}(\texttt{List}(\texttt{Bool}))$ *; d'un autre côté, l'entier* 0 *représentant la liste vide est compatible avec le type* $\texttt{List}(\alpha)$ *et le bloc* $\texttt{Blk}(0, 0)$ *est compatible avec le type* $\texttt{List}(\texttt{List}(\alpha))$.

Propriétés Comme dans le système de types du langage de valeurs v, ce système de types admet un lemme de substitution pour les variables du type τ. Contrairement aux environnements de typage Γ utilisés pour typer les programmes, les environnement de typage du tas Ψ sont implicitement clos et ce lemme de substitution ne nécessite pas d'instancier conjointement le type et l'environnement.

Lemme 3.1.1. *Soit un type* τ*, une valeur* w*, une valeur allouée* h*, un environnement de typage du tas* Ψ *et une substitution* θ*. Si* $\Psi \vdash w : \tau$*, alors* $\Psi \vdash w : \theta(\tau)$*. Si* $\Psi \vdash h : \tau$*, alors* $\Psi \vdash h : \theta(\tau)$.

Preuve Par cas sur la règle de dérivation. Dans le cas [W-Int], la relation Δ est stable par substitution. Dans le cas [W-Label], [H-Clos] et [H-RecClos], la relation d'instanciation est stable par substitution. Dans le cas [H-Blk], la relation ∇ est stable par substitution.

Une démonstration complète de la correction de ce système de types demanderait de redéfinir l'ensemble de la sémantique opérationnelle dans un formalisme un peu plus verbeux et ne sera pas effectuée ici. Comme garantie de correction du système de types du langage w nous nous contenterons du résultat suivant : en définissant une fonction d'expansion des valeurs w vers les valeurs v sans partage, nous montrerons que cette expansion conserve le typage (lemme 3.1.2). Cette fonction d'expansion est telle que si en pratique un programme OCaml a pour valeur un couple w/μ, alors dans la sémantique opérationnelle basée sur le langage de valeurs v, ce programme s'évaluera vers le couple v/μ' qui est l'expansion de w/μ.

Valeurs cycliques Le langage de valeurs v permet la description de valeurs cycliques lorsqu'elles sont construites à l'aide de références. Par exemple, en étendant le langage de types avec le type récursif $\texttt{LI} = \texttt{Option}(\texttt{Int} \times \texttt{Ref}(\texttt{LI}))$, on pourrait typer et évaluer le programme suivant :

$$\begin{array}{l}\texttt{let } \mathit{tail} = \texttt{ref}(\texttt{None}) \texttt{ in}\\ \texttt{let } \mathit{cons} = \texttt{Some}(1, \mathit{tail}) \texttt{ in}\\ \mathit{tail} := \mathit{cons};\\ \mathit{tail}\end{array}$$

Le résultat de l'évaluation de ce programme dans le langage de valeurs v est un couple ℓ/μ où $\mu \equiv \{\ell \mapsto \texttt{Blk}(\texttt{Blk}(1, \ell), 0)\}$. Ce tas est correctement typé à l'aide d'un environnement $\Psi \equiv \{\ell : \texttt{LI}\}$. Le langage OCaml permet aussi la construction de cycles sans faire usage de références à l'aide d'une définition récursive de valeurs comme dans le programme :

$$\texttt{let rec } x = 1 :: x \texttt{ in } x$$

Le langage de valeurs v ne permet pas de décrire la liste infinie de 1 résultant de l'évaluation d'un tel programme. Dans le langage de valeurs w, les blocs étant explicitement alloués, il est possible de décrire cette liste infinie par un couple ℓ/μ' où $\mu'(\ell) = \texttt{Blk}(1,\, \ell)$; et ce couple est compatible avec le type $\texttt{List}(\texttt{Int})$ en utilisant l'environnement de typage trivial $\Psi' \equiv \{\ell : \texttt{List}(\texttt{Int})\}$.

De manière plus générale, le système de types des valeurs w permet de typer les valeurs cycliques issues de programmes utilisant le principe de récursion polymorphe [Mycroft, 1984]. Par exemple, en étendant le système de types avec un type $\texttt{NR}(\alpha) = \texttt{Option}(\texttt{NR}(\alpha \times \alpha))$, le programme suivant :

$$\texttt{let rec}\ x : \forall\alpha.\, \texttt{NR}(\alpha) = \texttt{Some}(x)\ \texttt{in}\ x$$

pourrait s'évaluer vers le tas $\mu \equiv \{\ell_0 \mapsto \texttt{Blk}(\ell_0, 0)\}$. Ce tas est typable dans un environnement $\Psi \equiv \{\ell_0 : \texttt{NR}(\alpha)\}$.

Expansions et conservation de types En présence d'une valeur w/μ compatible avec un type τ, si les seuls cycles dans le tas μ ont été construits par des références, il est possible de reconstruire une valeur v/μ' correspondante dans le langage de valeurs v, elle même compatible avec τ. Cette transformation est notée $[\![\mu : \Psi \vdash h : \tau]\!]$ et $[\![\mu : \Psi \vdash w : \tau]\!]$. Elle est définie par cas sur la structure de la valeur et s'étend aux environnements ρ :

$$\begin{array}{rcl}
[\![\mu : \Psi \vdash i : \tau]\!] & = & i \\
[\![\mu : \Psi \vdash \ell : \tau]\!] & = & \left\{ \begin{array}{ll} \ell & si\ \tau \equiv \texttt{Ref}(\tau') \\ [\![\mu : \Psi \vdash \mu(\ell) : \tau]\!] & sinon \end{array} \right. \\
[\![\mu : \Psi \vdash \texttt{Blk}(w_1,\, w_2) : \tau]\!] & = & \texttt{Blk}([\![\mu : \Psi \vdash w_1 : \tau_1]\!],\, [\![\mu : \Psi \vdash w_2 : \tau_2]\!]) \qquad si\ [\tau_1; \tau_2] \bigtriangledown \tau \\
[\![\mu : \Psi \vdash \langle \lambda x.t, \rho \rangle : \tau]\!] & = & \langle \lambda x.t, [\![\mu : \Psi \vdash \rho : \Gamma]\!] \rangle \qquad si\ \exists\Gamma.\ \Gamma \vdash \lambda x.t : \tau\ et\ \Psi \vdash \rho : \Gamma \\
[\![\mu : \Psi \vdash \langle\!\langle f = \lambda x.t, \rho \rangle\!\rangle : \tau]\!] & = & \langle\!\langle f = \lambda x.t, [\![\mu : \Psi \vdash \rho : \Gamma]\!] \rangle\!\rangle \\
 & & si\ \exists\Gamma.\ \Gamma \vdash \texttt{fix}\ f = \lambda\, x.t : \tau\ et\ \Psi \vdash \rho : \Gamma
\end{array}$$

Pour énoncer le lemme de préservation du typage par expansion, on note $\Psi_{|\texttt{Ref}}$ l'environnement de typage dont le domaine est l'ensemble des adresses-mémoire ℓ du domaine de Ψ qui vérifient $\Psi(\ell) = \texttt{Ref}(\tau)$ et tel que $\Psi_{|\texttt{Ref}}(\ell) = \tau$. De la même manière, on note $\mu_{|\Psi,\texttt{Ref}}$ le tas dont le domaine est l'ensemble des adresses-mémoire ℓ du domaine de Ψ qui vérifient $\Psi(\ell) = \texttt{Ref}(\tau)$ et telles que $\mu_{|\Psi,\texttt{Ref}}(\ell) = v$ avec v tel que $[\![\mu : \Psi \vdash \mu(\ell) : \Psi(\ell)]\!] = \texttt{Blk}(v, 0)$.

Lemme 3.1.2. *Soit un tas μ et un environnement de typage du tas Ψ tels que $\mu : \Psi$. Soit une valeur w et un type τ tels que $\Psi \vdash w : \tau$. On vérifie $\mu_{|\Psi,\texttt{Ref}} : \Psi_{|\texttt{Ref}}$ et $\Psi_{|\texttt{Ref}} \vdash [\![\mu : \Psi \vdash w : \Psi(\ell)]\!] : \tau$.*

Preuve Par induction sur l'expansion $[\![\mu : \Psi \vdash w : \Psi(\ell)]\!]$.

Ce lemme nous apporte donc une certaine garantie de la correction du système de types des valeurs w, en l'absence de valeurs cycliques construites sans référence. Nous conjecturons que les cycles introduits par les définitions récursives n'introduisent pas d'élément pouvant mettre à mal la correction.

Exemple 3.1.3. *En reprenant le tas μ et les environnements compatibles Ψ et Ψ' de l'exemple 3.1.2 :*

$$\begin{array}{rcl} \mu & \equiv & \{\ \ell_0 \mapsto \texttt{Blk}(\ell_1, \ell_1) \ ;\ \ell_1 \mapsto \texttt{Blk}(1, 0)\ \} \\ \Psi & \equiv & \{\ \ell_0 : (\texttt{List}(\texttt{Int}) \times \texttt{List}(\texttt{Int})) \ ;\ \ell_1 : \texttt{List}(\texttt{Int})\ \} \\ \Psi' & \equiv & \{\ \ell_0 : (\texttt{Ref}(\texttt{Int}) \times \texttt{Ref}(\texttt{Int})) \ ;\ \ell_1 : \texttt{Ref}(\texttt{Int})\ \} \end{array}$$

La valeur v reconstruite à partir de ℓ_0 lorsque μ est typé avec Ψ duplique la valeur allouée en ℓ_1 : on obtient alors la valeur $\texttt{Blk}(\texttt{Blk}(1, 0), \texttt{Blk}(1, 0))$ *et un tas vide $\mu_{|\Psi,\texttt{Ref}} = \emptyset$. La valeur reconstruite avec Ψ' est simplement* $\texttt{Blk}(\ell_1, \ell_1)$*. Dans ce cas, le domaine du tas est réduit à ℓ_1 et la référence est désencapsulée $\mu_{|\Psi',\texttt{Ref}} = \{\ell_1 \mapsto 1\}$.*

3.1.2 Généralisation du système de types

Dans le système de types pour les valeurs w que nous venons de définir, il existe des couples w/μ incompatibles avec un type τ mais dont l'expansion dans le langage de valeur v est compatible avec τ. Par exemple, pour le tas :

$$\mu \equiv \{\ell_0 \mapsto \texttt{Blk}(\ell_1, \ell_1)\ ;\ \ell_1 \mapsto \texttt{Blk}(1, 0)\}$$

l'expansion de la valeur ℓ_0 est $\texttt{Blk}(\texttt{Blk}(1, 0), \texttt{Blk}(1, 0))$ et cette valeur est compatible avec le type $(\texttt{Option}(\texttt{Int}) \times \texttt{List}(\texttt{Int}))$, puisque la sous-valeur $\mu(\ell_1) = \texttt{Blk}(1, 0)$ est à la fois compatible avec $\texttt{Option}(\texttt{Int})$ et $\texttt{List}(\texttt{Int})$. Mais, comme il n'existe pas de type plus général que ces deux types qui soit compatible avec $\mu(\ell_1)$, il n'existe pas de Ψ compatible avec μ dans lequel $\Psi(\ell_0) \preccurlyeq (\texttt{Option}(\texttt{Int}) \times \texttt{List}(\texttt{Int}))$.

Pour définir un système de types sur les valeurs w où typer une valeur w ou son expansé v sont équivalents, il est possible d'autoriser les environnements de typage à associer plusieurs types à une même adresse-mémoire. On vérifie alors que la valeur de cette adresse-mémoire est compatible avec chacun des types associés. Cette généralisation du système de types est correcte[3] tant que l'ensemble des types associé à une valeur modifiable reste un singleton monomorphe.

Les environnements de typage de ce système de types généralisé seront appelés environnements de typage généralisés et seront notés Φ. Ils consistent en une suite d'associations de la forme $\ell : \{\tau_1, \ldots, \tau_n\}$. Dans ce système de types, une adresse-mémoire ℓ sera compatible avec un type τ s'il existe dans $\Phi(\ell)$ un type plus général que τ. Ainsi, la règle de typage suivante :

$$\frac{\tau' \in \Phi(\ell) \qquad \tau' \preccurlyeq \tau}{\Phi \vdash \ell : \tau} \quad [\text{W-Label'}]$$

3. Il est à noter que le tas μ utilisé en exemple n'est pas constructible par un programme ML de type $(\texttt{Option}(\texttt{Int}) \times \texttt{List}(\texttt{Int}))$ — il faudrait de la même façon qu'il existe une valeur ayant un type plus général que $\texttt{Option}(\texttt{Int})$ et $\texttt{List}(\texttt{Int})$ — mais un tel tas semble tout de même acceptable par une fonction de désérialisation sûre : la valeur $\mu(\ell_1)$ n'étant pas modifiable, elle peut être parcourue sans danger tantôt avec le type $\texttt{Option}(\texttt{Int})$ et tantôt avec le type $\texttt{List}(\texttt{Int})$.

remplace la règle [W-Label]. Les autres règles sont inchangées. On notera $\mu : \Phi$ ssi : (1) les adresses-mémoire du domaine de μ et celles apparaissant dans son image (notée $fl(img(\mu))$) appartiennent au domaine de Φ ; (2) toutes les valeurs de μ sont compatibles avec les types de Φ, c'est-à-dire : pour tout $\ell \in dom(\mu)$ et pour tout $\tau \in \Phi(\ell)$ alors $\Phi \vdash \mu(\ell) : \Phi(\ell)$; et (3) toutes les références de Φ sont monomorphes, c'est-à-dire : pour tout $\ell \in dom(\Phi)$ si $\texttt{Ref}(\tau) \in \Phi(\ell)$ alors $fv(\tau) = \emptyset$ et $|\Phi(\ell)| = 1$.

Le lemme 3.1.2 de préservation du typage par expansion reste valable dans le système de types généralisé. Par exemple, le tas μ utilisé en exemple au début de la section est compatible avec l'environnement de typage généralisé ci-dessous :

$$\Phi \equiv \{\ell_0 : \{(\texttt{Option(Int)} \times \texttt{List(Int)})\} ; \ell_1 : \{\texttt{Option(Int)} ; \texttt{List(Int)}\}\}$$

La notation $\Phi \cup \Phi'$ représentera un environnement de typage généralisé de domaine $dom(\Phi) \cup dom(\Phi')$ et tel que $(\Phi \cup \Phi')(\ell) = \Phi(\ell) \cup \Phi'(\ell)$, en posant $\Phi(\ell) = \emptyset$ si $\ell \notin dom(\Phi)$.

Par opposition, le système de types de la section 3.1.1 sera parfois appelé système de types original et les environnements de typage Ψ seront parfois appelé environnements de typage simple.

3.1.3 Système(s) de réécriture et algorithme de vérification

Le début de ce chapitre a permis de définir deux notions de compatibilité d'un graphe mémoire avec un type donné. Avec la première notion, les valeurs w/μ compatibles avec un type τ sont celles pour lesquelles il existe un environnement de typage Ψ compatible avec μ et dans lequel w soit compatible avec τ. La seconde notion étend la première en remplaçant les environnement de typage Ψ par des environnements de typage dits généralisés Φ. La suite de ce chapitre propose un algorithme vérifiant la compatibilité d'une valeur avec un type. Cet algorithme est paramétré par le choix d'une de ces deux notions.

L'algorithme de vérification détaillé ici va tenter de reconstruire un environnement de typage Ψ ou Φ compatible avec un tas μ en collectant récursivement à partir d'une contrainte de type initiale $w : \tau$ l'ensemble des contraintes de type que doit satisfaire cet environnement de typage. Si la collecte des contraintes aboutit à un point fixe et s'il existe un environnement de typage satisfaisant toutes ces contraintes, la valeur est acceptée. Si par contre au cours de la collecte une des contraintes est trivialement non satisfaisable ou si deux d'entre elles sont contradictoires, la valeur est refusée. Ce mécanisme de propagation de contraintes va être décrit à l'aide d'un système de réécriture : chaque règle de réécriture correspondra à une décomposition possible d'une contrainte de type ou à un cas d'erreur.

Cette technique de vérification est correcte si l'on sait montrer pour chaque règle de réécriture que l'existence d'un environnement de typage satisfaisant les contraintes de types du membre droit implique l'existence d'un tel environnement pour les contraintes de types du membre gauche. Ainsi, si une contrainte en forme normale[4] est trivialement

4. Dans notre système les formes normales seront des contraintes de la forme `True` et `False`.

satisfaite, la contrainte initiale sera satisfaisable. La propriété de préservation de la satisfaisabilité dans le sens direct sera appelée semi-complétude. Ainsi, si une contrainte en forme normale est trivialement non satisfaisable, la contrainte initiale sera elle aussi non satisfaisable. Avec cette notion, si les formes normales du système de réécriture sont trivialement satisfaites ou non satisfaisables, un algorithme implémentant une stratégie de réécriture terminante sera complet au sens usuel du terme, c'est-à-dire : si une contrainte est satisfaisable, elle se réécrit en un nombre fini d'étapes en une contrainte trivialement satisfaite.

Les difficultés de conception d'un tel système de réécriture tiennent à la présence de valeurs contenant du partage avec ou sans cycle, des fermetures et des valeurs modifiables. La suite de ce chapitre décrit plus précisément chacune des difficultés rencontrées et étudie diverses possibilités pour y répondre. Le chapitre se termine par la présentation de l'algorithme effectivement mis en œuvre.

La section 3.2 présente un système de réécriture basé sur le système de types généralisé et qui ne considère que le cas simple des tas sans cycle, ni fermeture, ni valeur modifiable, mais contenant des blocs partagés. On montre comment dans ce cas on obtient les propriétés de correction et de complétude. La section 3.3 présente ensuite les extensions nécessaires au système de réécriture de la section 3.2 pour prendre en compte les fermetures et les cycles en l'absence toujours de valeurs modifiables. On montre que ce système de réécriture conserve les propriétés de correction et de semi-complétude et comment la présence de certains cycles fait perdre la propriété de terminaison.

La section 3.4 définit un nouveau système de réécriture qui se restreint au tas typable dans le système de types original. On montre alors que cette restriction permet, en l'absence de certaines fermetures polymorphes, de retrouver la terminaison pour tous les tas cycliques. On montre ensuite que ce système de réécriture conserve les propriétés de correction et de semi-complétude et comment il perd la propriété de terminaison en présence conjointe de certaines fermetures et de cycles.

La syntaxe des termes manipulés par le système de réécriture de la section 3.3 étant différente de celle du système de réécriture de la section 3.4, la section 3.5 redéfinit le premier système en utilisant la syntaxe du second. On obtient alors un système de réécriture paramétré par le système de type sous-jacent qui conserve les propriétés de correction et de semi-complétude.

La section 3.6 indique comment ajouter des valeurs modifiables tout en conservant les propriétés déjà obtenues pour le système de réécriture paramétré de la section 3.5. D'un point de vue extensionnel, le système de réécriture présenté dans cette section est le plus abouti. L'algorithme correspondant permet ainsi de vérifier la compatibilité de toutes les valeurs productibles par un programme OCaml ne comprenant pas le type de fermeture décrite à la section 3.4, ce qui est dans ce cadre l'algorithme le plus avancé que nous connaissons à ce jour.

La dernière section de ce chapitre ouvre une transition vers le prototype réalisé durant cette thèse qui est décrit au chapitre 4. On introduit dans le système de réécriture de la section 3.6 des considérations pratiques : comment obtenir la terminaison dans tous les cas et accroître l'efficacité. Ces deux buts sont satisfaits à l'aide d'une stratégie de

parcours du graphe mémoire reposant sur un tri topologique. Cependant, l'impératif de terminaison est obtenu en renonçant à la semi-complétude du système de réécriture. La section se conclut sur une question ouverte : comment caractériser plus précisément l'ensemble des valeurs acceptables que notre prototype refuse ?

3.2 Cas simple : bloc non modifiable et partage sans cycle

Dans le cas simple d'un tas ne contenant ni bloc modifiable, ni fermeture, ni cycle, les données manipulées par l'algorithme sont constituées d'un tas μ et d'un ensemble de contraintes de type dont il reste à vérifier la satisfaisabilité. Ces contraintes portent sur des valeurs w et sont notées $w : \tau$. À chaque étape de vérification, l'algorithme choisit une contrainte et tente de la décomposer. Si cette contrainte porte sur un entier $i : \tau$, la vérification est immédiate : les entiers acceptables pour un type τ ont été caractérisés à la section 2.3.2 par la relation $i \vartriangle \tau$. Si la contrainte à résoudre porte sur une étiquette $\ell : \tau$ correspondant à un bloc $\mu(\ell) = \texttt{Blk}(w_1, w_2)$ et si le type τ peut être décomposé en deux types τ_1 et τ_2 vérifiant la relation $[\tau_1; \tau_2] \bigtriangledown \tau$ décrite à la section 2.3.2, alors l'algorithme engendre deux nouvelles contraintes à vérifier $w_1 : \tau_1$ et $w_2 : \tau_2$. Dans les deux cas, $i : \tau$ et $\ell : \tau$, et conformément à la propriété 2.3.5, si le type τ est une variable alors la vérification échoue systématiquement.

Syntaxe des contraintes Pour formaliser ces principes, l'algorithme est décrit à l'aide d'un système de réécriture. Les termes manipulés par ce système sont appelés contraintes (de types), ils représentent l'état courant de l'ensemble des données manipulées par l'algorithme. Dans cette première version, la syntaxe des contraintes est définie par la grammaire ci-dessous :

$$
\begin{array}{rcll}
C & ::= & \texttt{True} \mid \texttt{False} \mid C \wedge C & \\
& \mid & \mu.\, C & \textit{fragment du tas} \\
& \mid & w : \tau & \textit{contrainte de type}
\end{array}
$$

La notation $\mu.\, C$ est inspirée du langage Reference ML utilisée par Wright et Felleisen [1994] : elle permet de manipuler explicitement le tas dans la syntaxe des contraintes. Cette construction lie l'ensemble des étiquettes de $dom(\mu)$ à l'intérieur de la contrainte C. La notation $fl(C)$ représente les adresses-mémoire libres de la contrainte C, elle peut être définie inductivement sur la syntaxe des contraintes :

$$
\begin{array}{rcrcl}
fl(\texttt{True}) & = & \emptyset & \qquad fl(\ell : \tau) & = & \{\ell\} \\
fl(\texttt{False}) & = & \emptyset & fl(C_1 \wedge C_2) & = & fl(C_1) \cup fl(C_2) \\
fl(i : \tau) & = & \emptyset & fl(\mu.\, C) & = & (fl(C) \cup fl(img(\mu))) \setminus dom(\mu)
\end{array}
$$

Dans ce système de réécriture, le problème initial de compatibilité d'un couple w/μ avec un type τ sera représenté par la contrainte $\mu.(w : \tau)$. Si la valeur w est compatible avec le type τ, cette contrainte se réécrira vers la contrainte `True`, sinon elle se réécrira vers la contrainte `False`. Les états intermédiaires du système de réécriture seront des contraintes de la forme $\mu'.\, (w_1 : \tau_1 \wedge \ldots \wedge w_n : \tau_n)$ où μ' est le fragment du tas initial sur lequel portent les contraintes restant à vérifier $w_1 : \tau_1 \wedge \ldots \wedge w_n : \tau_n$.

Règles de réécriture Les contraintes portant sur les entiers sont résolues à l'aide des deux règles de réécriture suivantes. Elles s'appliquent dans tous les contextes possibles.

$$i : \tau \gg \texttt{True} \qquad si\ i \vartriangle \tau \qquad \text{[R-Int-True]}$$
$$i : \tau \gg \texttt{False} \qquad sinon \qquad \text{[R-Int-False]}$$

Les contraintes portant sur les blocs sont résolues à l'aide de deux autres règles.

$$\mu.(\ell : \tau \wedge C) \gg \mu.(w' : \tau' \wedge w'' : \tau'' \wedge C) \qquad \text{[R-Blk-True]}$$
$$si\ \mu(\ell) = \texttt{Blk}(w', w'')\ et\ \exists \tau'\tau''.\ [\tau'; \tau''] \bigtriangledown \tau$$
$$\mu.(\ell : \tau \wedge C) \gg \texttt{False} \quad si\ \mu(\ell) = \texttt{Blk}(w', w'')\ et\ \nexists \tau'\tau''.\ [\tau'; \tau''] \bigtriangledown \tau \qquad \text{[R-Blk-False]}$$

Pour compléter le système de réécriture, on ajoute les règles administratives suivantes.

$$C \wedge \texttt{False} \gg \texttt{False} \qquad \text{[R-False]}$$
$$C \wedge \texttt{True} \gg C \qquad \text{[R-True]}$$

Lorsque toutes les contraintes portant les adresses-mémoire d'un tas auront été résolues, on peut faire disparaître ce tas de la contrainte. C'est ce qu'exprime la règle suivante.

$$\mu.\, C \gg C \qquad si\ fl(C) \# dom(\mu) \qquad \text{[R-Heap]}$$

En pratique, cette règle s'applique lorsque C est `True` ou `False`.

Exemple 3.2.1. *Étant donné le tas* $\mu \equiv \{\ell_0 \mapsto \texttt{Blk}(1, \ell_1)\,;\, \ell_1 \mapsto \texttt{Blk}(2, 0)\}$*, si le type attendu est* `List(Int)`*, une suite possible de réécritures est :*

$$\begin{array}{rcl} & & \mu.\,(\ell_0 : \texttt{List(Int)}) \\ \text{[R-Blk-True]} & \gg & \mu.\,(1 : \texttt{Int} \wedge \ell_1 : \texttt{List(Int)}) \\ \text{[R-Int-True]} & \gg & \mu.\,(\texttt{True} \wedge \ell_1 : \texttt{List(Int)}) \\ \text{[R-True]} & \gg & \mu.\,(\ell_1 : \texttt{List(Int)}) \\ \text{[R-Blk-True]} & \gg & \mu.\,(2 : \texttt{Int} \wedge 0 : \texttt{List(Int)}) \\ \text{[R-Int-True]} & \gg & \mu.\,(\texttt{True} \wedge 0 : \texttt{List(Int)}) \\ \text{[R-True]} & \gg & \mu.\,(0 : \texttt{List(Int)}) \\ \text{[R-Int-True]} & \gg & \mu.\,\texttt{True} \\ \text{[R-Heap]} & \gg & \texttt{True} \end{array}$$

Si pour le même tas le type attendu est `Option(Int)`*, une suite possible de réécritures est :*

$$\begin{array}{rcl} & & \mu.\,\ell_0 : \texttt{Option(Int)} \\ \text{[R-Blk-True]} & \gg & \mu.\,1 : \texttt{Int} \wedge \ell_1 : \texttt{Unit} \\ \text{[R-Int-False]} & \gg & \mu.\,1 : \texttt{Int} \wedge \texttt{False} \\ \text{[R-False]} & \gg & \mu.\,\texttt{False} \\ \text{[R-Heap]} & \gg & \texttt{False} \end{array}$$

En présence de partage dans le tas, ce système de réécriture va vérifier indépendamment chacune des contraintes de type portant sur une même adresse-mémoire. Étant donné le tas $\mu' \equiv \{\ell_0 \mapsto \texttt{Blk}(\ell_1, \ell_1)\,;\, \ell_1 \mapsto \texttt{Blk}(1, 0)\}$*, si le type attendu est* $(\texttt{Option(Int)} \times$

`List(Bool)`), *une suite possible de réécritures est :*

$$\begin{array}{rcl} & & \mu'.\,(\ell_0 : (\texttt{Option(Int)} \times \texttt{List(Bool)})) \\ [\text{R-Blk-True}] & \gg & \mu'.\,(\ell_1 : \texttt{Option(Int)} \wedge \ell_1 : \texttt{List(Bool)}) \\ [\text{R-Blk-True}] & \gg & \mu'.\,(\ell_1 : \texttt{Option(Int)} \wedge 1 : \texttt{Bool} \wedge 0 : \texttt{List(Bool)}) \\ [\text{R-Blk-True}] & \gg & \mu'.\,(1 : \texttt{Int} \wedge 0 : \texttt{Unit} \wedge 1 : \texttt{Bool} \wedge 0 : \texttt{List(Bool)}) \\ & \gg^* & \texttt{True} \end{array}$$

Il est à noter cependant qu'aucun programme OCaml de type (`Option(Int)`×`List(Bool)`) *ne peut produire le tas* μ''.

En présence d'un tas contenant un cycle, cette première version du système de réécriture ne termine pas. Étant donné le tas $\mu'' \equiv \{\ell_0 \mapsto \texttt{Blk}(1, \ell_0)\}$, *si le type attendu est* `List(Int)`, *une suite infinie de réécritures serait :*

$$\begin{array}{rcl} & & \mu''.\,(\ell_0 : \texttt{List(Int)}) \\ [\text{R-Blk-True}] & \gg & \mu''.\,(1 : \texttt{Int} \wedge \ell_0 : \texttt{List(Int)}) \\ [\text{R-Int-True}] & \gg & \mu''.\,(\texttt{True} \wedge \ell_0 : \texttt{List(Int)}) \\ [\text{R-True}] & \gg & \mu''.\,(\ell_0 : \texttt{List(Int)}) \\ & \gg & \ldots \end{array}$$

Terminaison et formes normales En l'absence de cycle dans le tas, ce système de réécriture termine toujours. Plus formellement, il est possible de vérifier cette propriété en définissant, dans un premier temps, la taille d'une contrainte comme la taille des arbres représentés par chacune des contraintes restant à vérifier. Autrement dit, la taille d'une contrainte close C est définie par la fonction $size(C)$ ci-dessous.

$$\begin{array}{rcll} size(C) & = & size_C(\emptyset, C) & \\ size_C(\mu, \texttt{True}) & = & 1 & \\ size_C(\mu, \texttt{False}) & = & 1 & \\ size_C(\mu, C_1 \wedge C_2) & = & size_C(\mu, C_1) + size_C(\mu, C_2) & \\ size_C(\mu_1, (\mu_2.\,C)) & = & |dom(\mu_2)| + size_C(\mu_1 \oplus \mu_2, C) & \\ size_C(\mu, w : \tau) & = & size_w(\mu, w) & \\ size_w(\mu, i) & = & 1 & \\ size_w(\mu, \ell) & = & 1 + size_w(\mu, w_1) + size_w(\mu, w_2) & si\ \mu(\ell) = \texttt{Blk}(w_1, w_2) \\ size_w(\mu, \ell) & = & 1 + \sum_{x \in dom(\rho)} size_w(\mu, \rho(x)) & si\ \mu(\ell) = \langle \lambda x.\,t, \rho \rangle \\ & & & ou\ \mu(\ell) = \langle\!\langle f = \lambda x.t, \rho \rangle\!\rangle \end{array}$$

Dans un second temps, il est possible de vérifier que chaque application d'une des règles de réécriture définies précédemment fait décroître la taille d'une contrainte.

Par ailleurs, les formes normales de ce système de réécriture sont `True` et `False`. Il suffit de remarquer que toute contrainte de type sur un entier $i : \tau$ peut être réécrite par une des règles de réécriture [R-Int-...] ; toute contrainte de type sur une adresse-mémoire peut être réécrite par une des règles [R-Blk-...] ; toutes les conjonctions inutiles peuvent être réécrites par les règles [R-False] et [R-True] ; et, le tas peut finalement disparaître à l'aide de la règle [R-Heap].

Tri topologique Ce système de réécriture peut être réalisé par un simple parcours du graphe mémoire ignorant le partage. Néanmoins, cette mise en œuvre directe est inefficace : dans certains cas un bloc sera vérifié plusieurs fois avec le même type. Par exemple, dans le cas d'un arbre binaire complet dont tous les nœuds sont partagés $\mu \equiv \{\ell_0 \mapsto \texttt{Blk}(\ell_1, \ell_1)\,;\, \ell_1 \mapsto \texttt{Blk}(\ell_2, \ell_2)\,;\, \ldots\}$ le bloc $\mu(\ell_n)$ sera vérifié 2^n fois. L'algorithme peut être amélioré : avant de parcourir un bloc partagé, on attend d'avoir parcouru l'ensemble des blocs qui le référencent et on simplifie l'ensemble des contraintes collectées. Ce parcours détermine un ordre topologique des valeurs allouées.

La notation $\mu.C$ permet d'exprimer le tri topologique dans le système de réécriture, par la règle ci-dessous, où la notation $\uplus$ représente l'union disjointe. En l'absence de cycle et de fermeture, cette règle permet de décomposer un tas en fragments de la forme $\{\ell \mapsto \texttt{Blk}(w_1, w_2)\}$.

$$\mu.\,C \;\gg \mu_1.\,\mu_2.\,C \qquad si\ \mu_1 \uplus \mu_2 = \mu\ et\ fl(img(\mu_1))\ \#\ dom(\mu_2) \qquad \text{[R-Sort]}$$

La simplification de contraintes peut être décrite par la règle ci-dessous, dont la validité sera vérifiée en utilisant le lemme de substitution (lemme 3.1.1).

$$\ell : \tau \wedge \ell : \tau' \gg \ell : \tau' \qquad si\ \tau' \preccurlyeq \tau \qquad \text{[R-Merge]}$$

La première étape d'une stratégie de réécriture efficace est alors d'appliquer la règle [R-Sort] autant que possible au début de l'algorithme. Une fois le tas décomposé, la règle [R-Blk-True] est applicable uniquement sur le premier fragment de la contrainte. Lorsque toutes les contraintes portant sur un fragment ont été vérifiées, la règle [R-Heap] permet de faire disparaître ce fragment de la contrainte et autorise alors la vérification du fragment suivant. Avant de commencer cette vérification, l'algorithme appliquera autant que possible la règle [R-Merge] pour fusionner les contraintes de type redondantes.

Exemple 3.2.2. *Étant donné le tas :*

$$\mu \equiv \{\ell_0 \mapsto \texttt{Blk}(\ell_1, \ell_1)\,;\, \ell_1 \mapsto \texttt{Blk}(\ell_2, 0); \ell_2 \mapsto \texttt{Blk}(1, 0)\}$$

Une contrainte $\mu.\,C$ peut être décomposée en $\mu_2.\mu_1.\mu_0.\,C$ où $\mu_i \equiv \{\ell_i \mapsto \mu(\ell_i)\}$. De la même façon, elle peut être décomposé en $\mu_2.\mu_0.\mu_1.\,C$. Si le type attendu est $(\tau_1 \times \tau_2)$ où $\tau_1 = \texttt{Option}(\texttt{List}(\texttt{Int}))$ et $\tau_2 = \texttt{List}(\texttt{List}(\texttt{Int}))$, une suite possible de réécritures est :

$$\begin{array}{rcl}
 & & \mu.\,(\ell_0 : (\tau_1 \times \tau_2)) \\
\text{[R-Sort]} & \gg & \mu_2.\mu_1.\mu_0.\,(\ell_0 : (\tau_1 \times \tau_2)) \\
\text{[R-Blk-True]} & \gg & \mu_2.\mu_1.\mu_0.\,(\ell_1 : \tau_1 \wedge \ell_1 : \tau_2) \\
\text{[R-Heap]} & \gg & \mu_2.\mu_1.\,(\ell_1 : \tau_1 \wedge \ell_1 : \tau_2) \\
 & \gg^* & \mu_2.\mu_1.\,(\ell_2 : \texttt{List}(\texttt{Int}) \wedge \ell_1 : \tau_2) \\
 & \gg^* & \mu_2.\mu_1.\,(\ell_2 : \texttt{List}(\texttt{Int}) \wedge \ell_2 : \texttt{List}(\texttt{Int})) \\
\text{[R-Merge]} & \gg & \mu_2.\mu_1.\,(\ell_2 : \texttt{List}(\texttt{Int})) \\
\text{[R-Heap]} & \gg & \mu_2.\,(\ell_2 : \texttt{List}(\texttt{Int})) \\
 & \gg^* & \texttt{True}
\end{array}$$

Correction et complétude du système de réécriture Le début de cette section a proposé un système de réécriture et une stratégie de réécriture décrivant un algorithme de vérification de types pour un tas restreint à des blocs non modifiables et à du partage sans cycle. Il reste maintenant à vérifier que cet algorithme est correct et complet vis-à-vis du système de types généralisé défini à la section 3.1 : la correction de l'algorithme signifiera que si une contrainte $\mu.(w : \tau)$ se réécrit vers `True`, alors il existe un environnement de typage Φ tel que $\mu : \Phi$ et $\Phi \vdash w : \tau$; réciproquement, la complétude de l'algorithme signifiera que s'il existe un environnement Φ tel que $\mu : \Phi$ et $\Phi \vdash w : \tau$ alors la contrainte $\mu.(w : \tau)$ se réécrit vers `True` en un nombre fini d'étapes.

Pour vérifier ces deux propriétés, les contraintes sont munies d'une notion de satisfaisabilité, définie comme une relation entre un environnement de typage Φ et une contrainte C, que l'on note $\Phi \models C$. Elle est définie inductivement par l'ensemble de règles suivant :

$$\text{[C-True]}\quad \Phi \models \texttt{True} \qquad \text{[C-And]}\quad \frac{\Phi \models C_1 \qquad \Phi \models C_2}{\Phi \models C_1 \wedge C_2} \qquad \text{[C-Question]}\quad \frac{\Phi \vdash w : \tau}{\Phi \models w : \tau}$$

$$\text{[C-Heap]}\quad \frac{dom(\Phi') = dom(\mu) \qquad \mu : \Phi \oplus \Phi' \qquad \Phi \oplus \Phi' \models C}{\Phi \models \mu.C}$$

Il n'y a pas d'autre cas possible, en particulier `False` n'est jamais satisfaisable. Lorsque tout environnements Φ satisfait une contrainte C, on dira que la contrainte C est valide, ou encore que c'est une tautologie.

Avec ces notions, le lemme 3.2.1 montre que la validité d'une contrainte initiale $\mu.(w : \tau)$ implique la compatibilité de la valeur w/μ avec le type τ. Les lemmes 3.2.2 et 3.2.3 montrent deux propriétés de la relation $\Phi \models C$. Le lemme 3.2.4 et le théorème 3.2.5 montrent la correction du système de réécriture et le lemme 3.2.6 et le théorème 3.2.7 montrent sa semi-complétude. Le théorème 3.2.8 en déduit finalement la complétude de l'algorithme.

Lemme 3.2.1. *Soit un tas μ, une valeur w et un type τ. Alors, $\emptyset \models \mu.(w : \tau)$ ssi il existe un environnement de typage Φ tel que $\mu : \Phi$ et $\Phi \vdash w : \tau$.*

Preuve Trivial avec les règles [C-Heap] et [C-Question]. □

Lemme 3.2.2. *Soit une contrainte C et deux environnements de typage généralisés Φ et Φ' tels que $\Phi \subseteq \Phi'$. Si $\Phi \models C$ alors $\Phi' \models C$.*

Preuve Par induction sur la contrainte C. □

Lemme 3.2.3. *Soit une contrainte C et un environnement de typage généralisé Φ. Alors, $\Phi \models C$ ssi $\Phi_{|fl(C)} \models C$.*

Preuve Par induction sur la contrainte C. □

Lemme 3.2.4. *Pour chacune des règles de réécriture $C_1 \gg C_2$ définies ci-dessus, et pour tout environnement Φ, si $\Phi \models C_2$ alors $\Phi \models C_1$.*

Preuve Les preuves de correction sont détaillées dans l'ordre de définition des règles de réécriture.

– Entiers

Cas [R-Int-True]. Si $i \triangle \tau$, alors par la règle de typage [W-Int] pour tout Φ on a $\Phi \vdash i : \tau$; et par [C-Question] on en conclut $\Phi \models i : \tau$.

Cas [R-Int-False]. La contrainte `False` étant invalide, la correction de cette règle est triviale.

– Étiquettes

Cas [R-Blk-True]. Vérifions, lorsque $\mu(\ell) = \texttt{Blk}(w', w'')$ et $[\tau' ; \tau''] \triangledown \tau$, que pour tout Φ si $\Phi \models \mu.(w' : \tau' \wedge w'' : \tau'' \wedge C)$ alors $\Phi \models \mu.(\ell : \tau \wedge C)$.
Par définition de $\Phi \models \mu.(w' : \tau' \wedge w'' : \tau'' \wedge C)$, à l'aide de [C-Heap] et [C-Question], il existe un environnement de typage Φ' tel que :

(a) $dom(\Phi') = dom(\mu)$

(b) $\mu : \Phi \oplus \Phi'$

(c) $\Phi \oplus \Phi' \vdash w' : \tau'$

(d) $\Phi \oplus \Phi' \vdash w'' : \tau''$

(e) $\Phi \oplus \Phi' \models C$

La condition d'application $[\tau' ; \tau''] \triangledown \tau$ et les hypothèses (c) et (d) permettent à l'aide de la règle de typage [H-Blk] de vérifier $\Phi \oplus \Phi' \vdash \texttt{Blk}(w', w'') : \tau$. Cette dérivation de typage permet d'étendre l'hypothèse (b) pour vérifier $\mu : \Phi \oplus (\Phi' \cup \{\ell : \tau\})$ et en déduire $\Phi \models \mu.(\ell : \tau)$ avec [C-Heap]. L'hypothèse (e) généralisée à l'aide du lemme 3.2.2 en $\Phi \oplus (\Phi' \cup \{\ell : \tau\}) \models C$ permet alors de conclure $\Phi \models \mu.(\ell : \tau \wedge C)$.

Cas [R-Blk-False]. La contrainte `False` étant invalide, la correction de cette règle est triviale.

– Règles administratives

Cas [R-False]. La contrainte `False` étant invalide, la correction de cette règle est triviale.

Cas [R-True] La contrainte `True` est une tautologie.

Cas [R-Heap]. Vérifions, lorsque $fl(C) \# dom(\mu)$, que pour tout Φ si $\Phi \models C$ alors $\Phi \models \mu.\, C$. Il suffit pour cela d'utiliser [C-Heap] avec un environnement Φ' tel que $\Phi'(\ell) = \emptyset$ ou $\Phi'(\ell) = \{\star\}$. □

Théorème 3.2.5 (Correction). *Soit un tas μ, clos et ne contenant ni cycle, ni valeur modifiable, ni fermeture. Soit une adresse-mémoire $\ell \in dom(\mu)$ et un type τ. Si la contrainte $\mu.(\ell : \tau)$ se réécrit vers* `True` *alors il existe un environnement de typage généralisé Φ' tel que $\mu : \Phi'$ et $\Phi' \vdash \ell : \tau$.*

Preuve Le lemme 3.2.4 permet de vérifier que si une contrainte initiale $\mu.(\ell : \tau)$ se réécrit vers `True` alors cette contrainte est valide. La validité de cette contrainte lorsque le tas μ est clos implique l'existence d'un environnement de typage généralisé Φ' tel que $\mu : \Phi'$ et $\Phi' \vdash \ell : \tau$ (lemme 3.2.1). □

Lemme 3.2.6. *Pour chacune des règles de réécriture $C_1 \gg C_2$ définies ci-dessus, et pour tout environnement Φ, si $\Phi \models C_1$ alors $\Phi \models C_2$.*

Preuve Les preuves de complétude sont détaillées dans l'ordre de définition des règles de réécriture.

– Entiers

Cas [R-Int-True]. La contrainte `True` est une tautologie.

Cas [R-Int-False]. Si la relation $i \triangle \tau$ n'est pas vérifiée, alors la contrainte $i : \tau$ est invalide.

– Étiquettes

Cas [R-Blk-True]. Vérifions, lorsque $\mu(\ell) = \texttt{Blk}(w', w'')$ et $[\tau' ; \tau'] \bigtriangledown \tau$, que pour tout Φ, si $\Phi \models \mu.(\ell : \tau \wedge C)$ alors $\Phi \models \mu.(w' : \tau' \wedge w'' : \tau'' \wedge C)$.
Par définition de $\Phi \models \mu.(\ell : \tau \wedge C)$, à l'aide de [C-Heap], [C-And] et [C-Question], il existe un environnement de typage Φ' tel que :

(a) $dom(\Phi') = dom(\mu)$

(b) $\mu : \Phi \oplus \Phi'$

(c) $\Phi \oplus \Phi' \vdash \ell : \tau$

(d) $\Phi \oplus \Phi' \models C$

L'hypothèse (c) et la règle [W-Label] permettent d'obtenir un type $\tau' \in \Phi'(\ell)$ tel que $\tau' \preccurlyeq \tau$. L'hypothèse (b) appliquée au cas particulier de ℓ et τ' permet de vérifier $\Phi \oplus \Phi' \vdash \texttt{Blk}(w', w'') : \tau'$. Le lemme de substitution (lemme 3.1.1) permet d'en déduire $\Phi \oplus \Phi' \vdash \texttt{Blk}(w', w'') : \tau$. À partir de ce jugement de typage, de le règle [H-Blk], on obtient deux types τ_1 et τ_2 tels que $[\tau_1 ; \tau_2] \bigtriangledown \tau$ et $\Phi \oplus \Phi' \vdash w' : \tau_1$ et $\Phi \oplus \Phi' \vdash w'' : \tau_2$. La condition d'application $[\tau' ; \tau'] \bigtriangledown \tau$ et l'unicité de la décomposition par $\bigtriangledown$ (propriété 2.3.4) nous donnent $\tau' = \tau_1$ et $\tau'' = \tau_2$. On vérifie alors $\Phi \oplus \Phi' \models w' : \tau' \wedge w'' : \tau''$. Les hypothèses (b) et (d) permettent alors de conclure $\Phi \models \mu.(w' : \tau' \wedge w'' : \tau'' \wedge C)$.

Cas [R-Blk-False]. S'il n'existe pas de types τ_1 et τ_2 tels que $[\tau_1 ; \tau_2] \bigtriangledown \tau$, alors aucun environnement de typage Φ'' ne permettra de vérifier $\Phi'' \vdash \texttt{Blk}(w_1, w_2) : \tau$. Ainsi, une contrainte de la forme $(\mu, \Phi').(\ell : \tau)$ où $\mu(\ell) = \texttt{Blk}(w_1, w_2)$ ne sera jamais satisfaisable.

– Règles administratives

Cas [R-TRUE] **et** [R-FALSE]**.** Les deux implications sont triviales.

Cas [R-HEAP]**.** L'implication découle directement du lemme 3.2.3. □

Théorème 3.2.7 (Semi-complétude)**.** *Soit un tas μ, clos et ne contenant ni cycle, ni valeur modifiable, ni fermeture. Soit une adresse-mémoire $\ell \in dom(\mu)$ et un type τ. Si la contrainte $\mu.(\ell : \tau)$ se réécrit vers* `False` *alors il n'existe aucun environnement de typage généralisé Φ tel que $\mu : \Phi$ et $\Phi \models \ell : \tau$.*

Preuve Les implications du lemme 3.2.6 permettent de vérifier que si une contrainte initiale $\mu.(\ell : \tau)$ se réécrit vers `False`, alors cette contrainte est invalide. La non validité de cette contrainte lorsque que la tas μ est clos implique l'inexistence d'un environnement de typage généralisé Φ tel que $\mu : \Phi$ et $\Phi \models \ell : \tau$ (lemme 3.2.1). □

Théorème 3.2.8 (Complétude)**.** *Soit un tas μ, clos et ne contenant ni cycle, ni valeur modifiable, ni fermeture. Soit une adresse-mémoire $\ell \in dom(\mu)$ et un type τ. S'il existe un environnement de typage Φ tel que $\mu : \Phi$ et $\Phi \models \ell : \tau$, alors la contrainte $\mu.(\ell : \tau)$ se réécrit vers* `True` *en un nombre fini d'étapes.*

Preuve On a montré précédemment qu'en l'absence de cycles, de blocs modifiables et de fermetures, ce système de réécriture termine et que ses formes normales sont `True` ou `False`. La contraposition du théorème 3.2.7 permet de vérifier que s'il existe un environnement de typage Φ tel que $\mu : \Phi$ et $\Phi \models \ell : \tau$, alors la contrainte $\mu.(\ell : \tau)$ ne se réécrit pas vers `False`. On en déduit que $\mu.(\ell : \tau)$ se réécrit vers `True`. □

3.3 Bloc non modifiable, fermeture et partage avec cycle

La section précédente a défini un système de réécriture permettant de vérifier la compatibilité d'un tas sans fermeture ni cycle ni valeur modifiable. Cette section va étendre ce système de réécriture pour prendre en compte les fermetures et le partage en présence de cycles ; le système de types utilisé pour le tas reste le système de types généralisé (section 3.1.2). La première partie de cette section étudie les extensions nécessaires à la gestion des fermetures et la deuxième partie étudie celles nécessaires pour retrouver la terminaison en présence de certaines valeurs cycliques. La troisième partie de cette section détaillera la preuve de correction du système de réécriture obtenu et la quatrième partie celle de sa preuve de semi-complétude. La formalisation d'une stratégie de réécriture efficace est reportée à la section 3.7.

3.3.1 Fermeture

Pour décomposer une contrainte de type $\ell : \tau$, lorsque l'adresse-mémoire ℓ désigne une fermeture $\mu(\ell) = \langle \lambda x.t, \rho \rangle$, il va être nécessaire de retrouver un environnement de typage

Γ permettant de vérifier $\Gamma \vdash \lambda x.t : \tau$ et d'engendrer autant de nouvelles contraintes de type à vérifier qu'il y a d'éléments dans les environnements ρ et Γ. La principale difficulté est de retrouver cet environnement Γ.

Lors de l'exécution du programme le terme $\lambda x.t$ ne sera plus directement disponible sous la forme de son code source, mais sous la forme d'un pointeur vers une suite d'instruction en code octet ou en code binaire exécutable. Dans un système tel que le *Typed Assembly Language* de Morrisett *et al.* [1999], il est imaginable de pouvoir retyper le code exécutable pour vérifier la compatibilité du code assembleur avec le type attendu et de retrouver les types qu'il attend pour les valeurs capturées ρ. Néanmoins, dans le cadre du compilateur OCaml et de code octet ou assembleur non typé, cette approche demande de trop grandes modifications à l'environnement d'exécution. Il a plutôt été choisi, dans le prototype développé durant cette thèse, de conserver pour chaque pointeur de code de l'environnement de typage qui avait été utilisé par le compilateur pour typer la fonction originale. Ainsi, lorsque dans la dérivation de typage du programme source, une fonction $\lambda x.t$ aura été typée dans un environnement Γ avec le type $\tau_1 \to \tau_2$, le type associé au code binaire engendré par le compilateur sera noté $\forall \bar{\alpha}.\,(\Gamma \Rightarrow \tau_1 \to \tau_2)$, où $\bar{\alpha} = fv(\Gamma) \cup fv(\tau_1 \to \tau_2)$. Dans la suite de ce document, ce type est plus important pour l'algorithme que le code source lui-même ; une fermeture sera parfois notée $\langle \forall \bar{\alpha}.\,(\Gamma \Rightarrow \tau_1 \to \tau_2), \rho \rangle$.

Avec cette information de typage supplémentaire, il reste nécessaire de vérifier à quelle instance de ce type correspond la fermeture. Ainsi, pour vérifier la compatibilité d'une fermeture $\langle \forall \bar{\alpha}.\,(\Gamma \Rightarrow \tau_1 \to \tau_2), \rho \rangle$ avec un type τ on commence par unifier τ et $\tau_1 \to \tau_2$. Si l'on obtient un unificateur θ, il faut ensuite vérifier que les domaines de Γ et de ρ sont égaux et que pour chaque variable x de ces domaines, la valeur $\rho(x)$ est compatible avec $\theta(\Gamma(x))$. Pour procéder à cette vérification, il faut tenir compte du fait que l'unificateur θ n'a pas systématiquement instancié toutes les variables libres de Γ. L'exemple ci-dessous illustre une telle situation.

Exemple 3.3.1. *Le programme suivant a pour type* `Unit` $\to$ `Int`.

$$\begin{array}{l}\texttt{let}\ \mathit{delay} = \lambda f.\lambda x.\lambda y.\ f\ x\ \texttt{in}\\ \texttt{let}\ \mathit{succ} = \lambda z.\ z + 1\ \texttt{in}\\ \mathit{delay}\ \mathit{succ}\ 1\end{array}$$

Il va s'évaluer dans le langage v vers la fermeture :

$$\begin{array}{rl} & \langle \lambda y.\ f\ x,\ \rho \rangle \\ \mathit{où} & \rho \equiv \{x \mapsto 1; f \mapsto \langle \lambda z.z + 1, \emptyset \rangle\}\end{array}$$

Suivant le principe du lemme de préservation du typage par évaluation (lemme 2.5.2), cette fermeture peut être typée avec le même type que le programme source. Pour cela, il faut dans la dérivation de typage de cette fermeture utiliser pour typer ρ un environnement de typage $\{x : \texttt{Int}\,;\, f : \texttt{Int} \to \texttt{Int}\}$. D'un autre côte, la fonction $\lambda y.\ f\ x$ étant polymorphe le type conservé par le compilateur sera :

$$\forall \alpha\beta\gamma.\,(\{x : \alpha\,;\, f : \alpha \to \beta\} \Rightarrow \gamma \to \beta)$$

et suivant le principe de vérification décrit ci-dessus, si le type attendu est $\texttt{Unit} \to \texttt{Int}$ *alors l'unification avec* $\gamma \to \beta$ *produire une substitution* $\theta = \{\beta \mapsto \texttt{Int}\,;\, \gamma \mapsto \texttt{Unit}\}$*, et le variable* α *ne sera pas instanciée. Cette variable pourra être instanciée comme attendu en* `Int` *lors de la vérification de compatibilité de la fermeture* $\langle \lambda z.z+1, \emptyset \rangle$ *avec le type* $\theta(\alpha \to \beta)$*.*

Variables existentielles De telles variables de type non instanciées au moment de l'unification du type de la fermeture avec le type attendu, sont du point de vue de l'algorithme des inconnues de types dont l'instance exacte n'est pas encore connue. Pour cela, elles seront explicitement quantifiées existentiellement dans la syntaxe des contraintes. Par opposition, les variables de type laissées libres dans la syntaxe des contraintes et représentant notamment le type vide seront appelées variables universelles. Pour distinguer syntaxiquement les deux types de variables, les variables existentielles seront systématiquement pointées $\dot{\alpha}$. Avec cette notation, une contrainte $w : \alpha$ sera trivialement fausse, tandis que la vérification d'une contrainte $w : \dot{\alpha}$ devra être retardée jusqu'à l'instanciation de la variable $\dot{\alpha}$.

En plus de représenter le type vide, en présence de fermeture les variables universelles serviront aussi à représenter le polymorphisme : par exemple, pour vérifier qu'une fermeture est compatible avec le schéma de type $\forall\alpha.\,\alpha \to \alpha$ la contrainte générée sera $\ell : \alpha \to \alpha$ où la variable α sera « fraîche ».

On notera $fev(C)$ l'ensemble des variables existentielles libres de la contrainte C et $fuv(C)$ celui de ses variables universelles libres.

Syntaxe des contraintes La syntaxe des contraintes est étendue avec une quantification existentielle des variables de types et avec un prédicat d'égalité entre types. Ces égalités seront décidées via un mécanisme d'unification, portant uniquement sur les variables de type existentielles.

$$
\begin{array}{rcll}
C & ::= & \ldots & \\
 & | & \exists\dot{\alpha}.\,C & \textit{quantification existentielle} \\
 & | & \tau = \tau & \textit{unification}
\end{array}
$$

Il sera toujours possible de résoudre les contraintes d'unification et de faire remonter les quantifications existentielles en tête de la contrainte. Ainsi, les états intermédiaires du système de réécriture sont représentés par des contraintes de la forme :

$$\exists\bar{\dot{\alpha}}.\,\mu.\,(w_1 : \tau_1 \wedge \ldots \wedge w_m : \tau_m)$$

Avant résolution des contraintes d'unification, les contraintes auront la forme ci-dessous, où l'accolade représente une conjonction de contraintes :

$$\exists\bar{\dot{\alpha}}.\begin{cases} \tau_1 = \tau_1' \wedge \ldots \wedge \tau_n = \tau_n' \\ \mu.\,(w_1 : \tau_1 \wedge \ldots \wedge w_m : \tau_m) \end{cases}$$

Règles de réécriture L'ensemble des règles de réécriture doit être étendu avec une règle pour les fermetures et un ensemble de règles pour gérer les contraintes d'unification. Dans la règle ci-dessous, la notation $\dot{\tau}$ désigne une instance du type τ dans lequel les variables existentielles $\dot{\alpha}$ remplacent les variables universelles α. De même, la notation $\dot{\Gamma}$ représente une instanciation de l'environnement Γ où des variables existentielles remplacent les variables universelles libres et où les variables quantifiées universellement à l'intérieur de Γ sont remplacées dans $\dot{\Gamma}$ par des variables universelles « fraîches » dans la contrainte.

$$\mu.\,(\ell : \tau \wedge C) \gg \exists\bar{\dot{\alpha}}.\begin{cases} \tau = \dot{\tau}' \to \dot{\tau}'' \\ \mu.\,(C \wedge \bigwedge_{\ell \in dom(\rho)} \rho(\ell) : \dot{\Gamma}(\ell)) \end{cases} \qquad \text{[R-Clos]}$$
$$si\ \mu(\ell) = \langle \forall\bar{\alpha}.\,(\Gamma \Rightarrow \tau' \to \tau''), \rho\rangle\ et\ \bar{\dot{\alpha}}\ \#\ ftv(\tau) \cup ftv(C)$$

Les contraintes d'égalité seront résolues en calculant l'unificateur le plus général des deux types (*m.g.u.*) [Robinson, 1965]. Lors du calcul de cet unificateur, seules les variables existentielles pourront être instanciées. Les variables universelles seront considérées comme des constructeurs de types distincts.

$$\exists\bar{\dot{\alpha}}.\,\tau_1 = \tau_2 \wedge C \gg \dot{\theta}(C) \quad si\ \dot{\theta}\ est\ le\ m.g.u\ de\ \tau_1\ et\ \tau_2\ et\ si\ dom(\dot{\theta}) = \bar{\dot{\alpha}} \qquad \text{[R-Unif]}$$
$$\exists\bar{\dot{\alpha}}.\,\tau_1 = \tau_2 \wedge C \gg \texttt{False} \quad si\ \tau_1\ et\ \tau_2\ n'admettent\ pas\ d'unificateur \qquad \text{[R-NUnif]}$$

En pratique les contraintes d'unification seront résolues immédiatement après leur introduction. Ainsi, les seules variables existentielles apparaissant dans la syntaxe des contraintes seront celles ne pouvant pas immédiatement être instanciées par la contrainte d'unification introduite par la règle [R-Clos]— c'est-à-dire lorsque le type attendu est clos, celles apparaissant uniquement dans l'environnement d'une fermeture.

Exemple 3.3.2. *Le programme suivant a pour type* $\texttt{Unit} \to \texttt{List}(\alpha)$.

$$\begin{array}{l} \texttt{let}\ delay = \lambda f.\lambda x.\lambda y.\ f\ x\ \texttt{in} \\ \texttt{let}\ id = \lambda z.\ z\ \texttt{in} \\ delay\ id\ [] \end{array}$$

Dans le langage w, il peut s'évaluer vers une valeur ℓ_0/μ *où :*

$$\mu \equiv \{\ \ell_0 \mapsto \langle \lambda y.\ f\ x, \{x \mapsto 0\,;\ f \mapsto \ell_1\}\rangle\ ;\ \ell_1 \mapsto \langle \lambda z.\ z, \emptyset\rangle\ \}$$

et le type associé à $\lambda y.\ f\ x$ *est* $\forall\beta\gamma\delta.\,(\{f : \beta \to \gamma\,;\ x : \beta\} \Rightarrow \delta \to \gamma)$ *et celui associé à* $\lambda z.\ z$ *est* $\forall\varepsilon.\,(\emptyset \Rightarrow \varepsilon \to \varepsilon)$. *Une suite possible de réécritures est :*

$$\begin{array}{rcl} & & \mu.\,(\ell_0 : \texttt{Unit} \to \texttt{List}(\alpha)) \\ \text{[R-Clos]} & \gg & \exists\dot{\beta}\dot{\gamma}\dot{\delta}.\begin{cases} \texttt{Unit} \to \texttt{List}(\alpha) = \dot{\delta} \to \dot{\gamma} \\ \mu.\,(\ell_1 : \dot{\beta} \to \dot{\gamma} \wedge 0 : \dot{\beta}) \end{cases} \\ \text{[R-Unif]} & \gg & \exists\dot{\beta}.\,\mu.\,(\ell_1 : \dot{\beta} \to \texttt{List}(\alpha) \wedge 0 : \dot{\beta}) \\ \text{[R-Clos]} & \gg & \exists\dot{\beta}\dot{\varepsilon}.\begin{cases} \dot{\beta} \to \texttt{List}(\alpha) = \dot{\varepsilon} \to \dot{\varepsilon} \\ \mu.\,(0 : \dot{\beta}) \end{cases} \\ \text{[R-Unif]} & \gg & \mu.\,(0 : \texttt{List}(\alpha)) \\ \text{[R-Int-True]} & \gg & \mu.\,\texttt{True} \\ \text{[R-Heap]} & \gg & \texttt{True} \end{array}$$

Exemple 3.3.3. *Le programme suivant a lui aussi pour type* $\texttt{Unit} \to \texttt{List}(\alpha)$.

$$\begin{array}{l}\texttt{let}\ id = \lambda z.\, z\ \texttt{in}\\ \texttt{let}\ id\_delay = \lambda x.\lambda y.\, id\ x\ \texttt{in}\\ id\_delay\ []\end{array}$$

Il s'évalue vers une valeur ℓ_0/μ *où* μ *est isomorphe au tas de l'exemple 3.3.2 :*

$$\mu \equiv \{\ \begin{array}{lcl}\ell_0 & \mapsto & \langle \lambda y.\, id\ x, \{x \mapsto 0\,;\, id \mapsto \ell_1\}\rangle \\ \ell_1 & \mapsto & \langle \lambda z.\, z, \emptyset\rangle\end{array}\ \begin{array}{l};\\ \}\end{array}$$

mais le type associé à $\lambda y.\, id\ x$ *est* $\forall\gamma\delta.\,(\{id : \forall\beta.\,\beta \to \beta\,;\, x : \gamma\} \Rightarrow \delta \to \gamma)$*, il comprend un schéma de type dans son environnement de typage. Le type associé à* $\lambda z.\, z$ *est toujours* $\forall\varepsilon.\,(\emptyset \Rightarrow \varepsilon \to \varepsilon)$*. Une suite possible de réécritures est alors :*

$$\begin{array}{rcl}
 & & \mu.\,(\ell_0 : \texttt{Unit} \to \texttt{List}(\alpha)) \\
[\text{R-Clos}] & \gg & \exists\dot\gamma\dot\delta.\ \left\{\begin{array}{l}\texttt{Unit} \to \texttt{List}(\alpha) = \dot\delta \to \dot\gamma \\ \mu.\,(\ell_1 : \beta \to \beta \wedge 0 : \dot\gamma)\end{array}\right. \\
[\text{R-Unif}] & \gg & \mu.\,(\ell_1 : \beta \to \beta \wedge 0 : \texttt{List}(\alpha)) \\
[\text{R-Clos}] & \gg & \exists\dot\varepsilon.\ \left\{\begin{array}{l}\beta \to \beta = \dot\varepsilon \to \dot\varepsilon \\ \mu.\,(0 : \texttt{List}(\alpha))\end{array}\right. \\
[\text{R-Unif}] & \gg & \mu.\,(0 : \texttt{List}(\alpha)) \\
[\text{R-Int-True}] & \gg & \mu.\,\texttt{True} \\
[\text{R-Heap}] & \gg & \texttt{True}
\end{array}$$

Valeurs inutiles L'ajout d'inconnues de type à la syntaxe des contraintes va bloquer la résolution de certaines contraintes. Par exemple, une contrainte de la forme $\exists\dot\alpha\dot\beta.\,\mu.\,(\ell : \dot\alpha \wedge i : \dot\beta)$ où $\mu(\ell) = \texttt{Blk}(w_1, w_2)$ ne peut plus se réécrire. Néanmoins une contrainte de cette forme peut-être trivialement résolue en instanciant $\dot\alpha$ et $\dot\beta$ avec le type universel $\star$. D'une manière générale, une contrainte composée uniquement de conjonctions de contraintes de type portant sur des variables existentielles sera trivialement satisfaisable à l'aide du type universel $\star$. En ce sens, les valeurs typées à l'aide du type universel seront les valeurs sur lesquelles ne portent aucune contrainte de type.

$$\exists\bar{\dot\alpha}.\,\mu.\,\overline{(w : \dot\alpha)} \gg \texttt{True} \qquad [\text{R-Univ}]$$

À l'aide de cette règle et en l'absence de décomposition du tas par un tri topologique, les formes normales sont à nouveaux `True` et `False`. En présence d'une stratégie de réécriture basée sur un tri topologique et de variables de type existentielles, certaines contraintes de type non entièrement résolues peuvent ne plus se réécrire, comme par exemple :

$$\exists\dot\alpha.\,\mu_2.\,\mu_1.\,(\ell_2 : \dot\alpha \to \texttt{Int} \wedge \ell_1 : \dot\alpha)$$

Les modifications nécessaires pour débloquer ces contraintes seront étudiées dans la section 3.7. Il est à noter néanmoins que l'ajout des valeurs fonctionnelles ne modifie par l'argument de terminaison en l'absence de cycle qui a été décrit à la section 3.2.

3.3.2 Partage avec cycle

Le mécanisme de gestion du partage par vérification multiple d'une valeur partagée, qui a été décrit à la section 3.2, ne termine pas en présence de cycles. Une possibilité pour surmonter ce problème est de mémoriser comme des hypothèses les types déjà vérifiés ou partiellement vérifiés pour chaque adresse-mémoire et lors de la vérification d'un type provenant de l'intérieur d'un cycle de le comparer aux types déjà mémorisées comme hypothèses. Pour représenter ce comportement, la construction syntaxique $\mu.\, C$ va être remplacée par une construction $(\mu, \Phi').\, C$ où Φ' est un environnement de typage généralisé servant à mémoriser les hypothèses de type[5]. Ainsi, la contrainte initiale sera $(\mu, \emptyset).\, (\ell : \tau)$ et les états intermédiaires seront de la forme :

$$(\mu, \Phi').\, (w_1 : \tau_1 \wedge \ldots \wedge w_m : \tau_m)$$

Règles de réécriture Les règles de réécriture d'une contrainte de type portant sur un bloc (règles [R-Blk-True] et [R-Blk-False]) ou sur une fermeture (règle [R-Clos]) doivent être adaptées pour mémoriser une nouvelle hypothèse de type.

$$(\mu, \Phi').\, (\ell : \tau \wedge C) \gg (\mu, \Phi' \cup \{\ell : \tau\}).\, (w' : \tau' \wedge w'' : \tau'' \wedge C) \quad [\text{R-Blk-True'}]$$
$$si\ \mu(\ell) = \texttt{Blk}(w', w'')\ et\ \exists \tau'\tau''\ tels\ que\ [\tau'; \tau''] \bigtriangledown \tau$$

$$(\mu, \Phi').\, (\ell : \tau \wedge C) \gg \texttt{False} \quad [\text{R-Blk-False'}]$$
$$si\ \mu(\ell) = \texttt{Blk}(w', w'')\ et\ \tau \not\equiv \dot{\alpha}\ et\ \nexists \tau'\tau''\ tels\ que\ [\tau'; \tau''] \bigtriangledown \tau$$

$$(\mu, \Phi').\, (\ell : \tau \wedge C) \gg \exists \bar{\dot{\alpha}}.\, \begin{cases} \tau = \dot{\tau}' \to \dot{\tau}'' \\ (\mu, \Phi' \cup \{\ell : \tau\}).\, (C \wedge \bigwedge_{\ell \in dom(\rho)} \rho(\ell) : \dot{\Gamma}(\ell)) \end{cases} \quad [\text{R-Clos'}]$$
$$si\ \mu(\ell) = \langle \forall \bar{\alpha}.\, (\Gamma \Rightarrow \tau' \to \tau''), \rho \rangle\ et\ \bar{\dot{\alpha}}\ \#\ ftv(\tau) \cup ftv(\Phi') \cup ftv(C)$$

Lorsqu'il sera possible de décider qu'une contrainte de type est instance d'une hypothèse quelles que soient les substitutions à venir pour les variables de type existentielles, la résolution sera immédiate.

$$(\mu, \Phi').\, (\ell : \tau \wedge C) \gg (\mu, \Phi').\, C \qquad si\ \forall \dot{\theta}.\, \exists \tau' \in \Phi'(\ell).\, \dot{\theta}(\tau') \preccurlyeq \dot{\theta}(\tau) \quad [\text{R-Merge'}]$$

La condition d'application de la règle [R-Merge], n'étant pas toujours décidable, la section 3.7.2 détaillera la condition plus restrictive utilisée en pratique.

La règle [R-Heap], décrite à la section 3.2, permet de terminer une suite de réécritures en faisant disparaître de la syntaxe d'une contrainte les fragments du tas sur lesquels ne portent plus de contraintes de type. Elle se transpose directement dans la nouvelle syntaxe.

$$(\mu, \Phi').\, C \gg C \qquad si\ fl(C)\ \#\ dom(\mu) \quad [\text{R-Heap'}]$$

Exemple 3.3.4. *Étant donné le tas* $\mu \equiv \{\ell_0 \mapsto \texttt{Blk}(1, \ell_0)\}$, *si le type attendu est*

5. Cet environnement représentant des hypothèses de typage sera souvent noté Φ' pour le distinguer de l'environnement Φ servant implicitement de contexte de satisfaisabilité.

`List(Int)`, *une suite possible de réécritures est :*

$$
\begin{array}{rcl}
 & & (\mu, \emptyset).\,(\ell_0 : \texttt{List(Int)}) \\
\text{[R-Blk-True']} & \gg & (\mu, \{\ell_0 : \{\texttt{List(Int)}\}\}).\,(1 : \texttt{Int} \wedge \ell_0 : \texttt{List(Int)}) \\
\text{[R-Merge']} & \gg & (\mu, \{\ell_0 : \{\texttt{List(Int)}\}\}).\,(1 : \texttt{Int}) \\
\text{[R-Int-True]} & \gg & (\mu, \{\ell_0 : \{\texttt{List(Int)}\}\}).\,\texttt{True} \\
\text{[R-Heap']} & \gg & \texttt{True}
\end{array}
$$

Terminaison Si la mémorisation d'hypothèses de type permet la vérification de certaines valeurs cycliques, le système de réécriture ne termine pas systématiquement, notamment, comme le montre l'exemple ci-dessous en présence de valeurs cycliques ayant un type polymorphe.

Exemple 3.3.5. *Par exemple, en définissant le type* $\texttt{NR}(\alpha) = \texttt{Option}(\texttt{NR}(\alpha \times \alpha))$*, le tas* $\mu \equiv \{\ell_0 \mapsto \texttt{Blk}(\ell_0, 0)\}$ *est typable dans un environnement* $\Phi \equiv \{\ell_0 : \{\texttt{NR}(\alpha)\}\}$*. On vérifie en particulier* $\Phi \vdash \ell_0 : \texttt{NR(Int)}$*. Néanmoins, la contrainte* $\mu.\,(\ell_0 : \texttt{NR(Int)})$ *se réécrit à l'infini :*

$$
\begin{array}{rl}
 & (\mu, \emptyset).\,(\ell_0 : \texttt{NR(Int)}) \\
\gg^* & (\mu, \{\ell_0 : \{\texttt{NR(Int)}\}\}).\,(\ell_0 : \texttt{NR(Int} \times \texttt{Int)}) \\
\gg^* & (\mu, \{\ell_0 : \{\texttt{NR(Int)} ; \texttt{NR(Int} \times \texttt{Int)}\}\}).\,(\ell_0 : \texttt{NR((Int} \times \texttt{Int)} \times \texttt{(Int} \times \texttt{Int)))} \\
\gg^* & \ldots
\end{array}
$$

Cette possibilité de boucle ne peut pas être évitée dans le système de réécriture décrit ici. La restriction aux valeurs typables dans le système de types original à la section 3.4 permettra de retrouver la terminaison sur cet exemple particulier.

3.3.3 Correction

Les parties précédentes ont présentées les règles de réécriture nécessaires pour vérifier la compatibilité d'un tas contenant des blocs non modifiables, des fermetures et des cycles, lorsque le système de types sous-jacent est le système généralisé. La syntaxe des contraintes manipulées est maintenant :

$$
\begin{array}{rcll}
C & ::= & \texttt{True} \mid \texttt{False} \mid C \wedge C & \\
 & \mid & (\mu, \Phi').\,C & \textit{fragment du tas} \\
 & \mid & w : \tau & \textit{contrainte de type} \\
 & \mid & \exists \dot{\alpha}.\,C & \textit{quantification existentielle} \\
 & \mid & \tau = \tau & \textit{unification}
\end{array}
$$

L'ensemble des règles de réécriture est regroupé dans la figure 3.1. Dans ce système, le contexte de satisfaisabilité d'une contrainte contient aussi une substitution $\dot{\theta}$ dont le domaine ne contient que des variables de type existentielles mais dont l'image n'en contient pas. Ainsi, la notion de satisfaisabilité d'une contrainte est définie comme une relation entre un environnement de typage Φ, une substitution $\dot{\theta}$ et une contrainte C, que l'on note $\dot{\theta}, \Phi \models C$. Elles est définie inductivement par l'ensemble de règles suivant.

$$\text{[C-True]}\quad \dot\theta, \Phi \models \texttt{True} \qquad \text{[C-And]}\quad \frac{\dot\theta, \Phi \models C_1 \qquad \dot\theta, \Phi \models C_2}{\dot\theta, \Phi \models C_1 \wedge C_2} \qquad \text{[C-Question]}\quad \frac{\Phi \vdash w : \dot\theta(\tau)}{\dot\theta, \Phi \models w : \tau}$$

$$\text{[C-Equal]}\quad \frac{\dot\theta(\tau_1) \equiv \dot\theta(\tau_2)}{\dot\theta, \Phi \models \tau_1 = \tau_2} \qquad \text{[C-Exists]}\quad \frac{fev(\tau) = \emptyset \qquad \dot\theta \oplus \{\dot\alpha \mapsto \tau\}, \Phi \models C}{\dot\theta, \Phi \models \exists \dot\alpha.\, C}$$

$$\text{[C-Heap]}\quad \frac{dom(\dot\theta(\Phi') \cup \Phi'') = dom(\mu) \qquad \Phi \oplus (\dot\theta(\Phi') \cup \Phi'') \vdash \mu : \Phi'' \qquad \dot\theta, \Phi \oplus (\dot\theta(\Phi') \cup \Phi'') \models C}{\dot\theta, \Phi \models (\mu, \Phi').\, C}$$

Autrement dit :

- une contrainte $\exists \dot\alpha.\, C$ est satisfaite par une substitution $\dot\theta$, s'il existe un type τ sans variable existentielle tel que la contrainte C soit validée par la substitution $\dot\theta$ étendue par l'association $\{\dot\alpha \mapsto \tau\}$ — le lemme 3.3.4 montre qu'il est équivalent de substituer directement la variable $\dot\alpha$ dans C ;
- une contrainte d'égalité $\tau_1 = \tau_2$ est satisfaite par une substitution $\dot\theta$, si après application de la substitution les types $\dot\theta(\tau_1)$ et $\dot\theta(\tau_2)$ sont syntaxiquement égaux ;
- et une contrainte $(\mu, \Phi').\, C$ est satisfaite par un environnement Φ et une substitution $\dot\theta$ s'il existe un environnement Φ'' compatible avec μ sous les hypothèses $\Phi \oplus (\dot\theta(\Phi') \cup \Phi'')$ et tel que ces mêmes hypothèses permettent de satisfaire la contrainte C. Avec cette notion de satisfaisabilité, une contrainte $(\mu, \emptyset).\, C$ a la même signification que la construction $\mu.\, C$. Cependant, la validité d'une contrainte $(\mu, \Phi').\, C$ n'impose pas la compatibilité de μ avec Φ'. Cette "non compatibilité" est utile pour simplifier la preuve de correction du système de réécriture et en particulier celle de la règle [R-Heap']. La section 3.3.4 montrera qu'en pratique, si la contrainte initiale est valide, les hypothèses de types mémorisées dans Φ' au cours de la réécriture sont effectivement compatibles avec μ.

Lemme 3.3.1. *Soit un tas μ, une valeur w et un type τ. On a $\emptyset, \emptyset \models (\mu, \emptyset).\, (w : \tau)$ ssi il existe un environnement de typage Φ'' tel que $\mu : \Phi''$ et $\Phi'' \vdash w : \tau$.*

Preuve Trivial avec les règles [C-Heap] et [C-Question]. □

Lemme 3.3.2. *Soit une contrainte C et deux environnement de typage Φ et Φ' tels que $\Phi \subseteq \Phi'$. Si $\dot\theta, \Phi \models C$ alors $\dot\theta, \Phi' \models C$.*

Preuve Par induction sur la dérivation de $\dot\theta, \Phi \models C$. □

Lemme 3.3.3. *Étant donnés une contrainte C et un environnement de typage Φ, on vérifie $\dot\theta, \Phi \models C$ ssi $\dot\theta, \Phi_{|fl(C)} \models C$.*

Preuve Par induction sur la dérivation de $\dot{\theta}, \Phi \models C$. Le seul cas non immédiat est le cas [C-HEAP]. Dans ce cas, la condition $dom(\dot{\theta}(\Phi') \cup \Phi'') = dom(\mu)$ permet d'appliquer l'hypothèse d'induction. □

Lemme 3.3.4. *Pour toute contrainte C, on vérifie $\dot{\theta}, \Phi \models C$ ssi $\emptyset, \Phi \models \dot{\theta}(C)$.*

Preuve Par induction sur la taille de la contrainte. Le seul cas non immédiat est le cas $\exists \dot{\alpha}.\, C$. Dans ce cas, on peut supposer $\dot{\alpha} \notin dom(\dot{\theta})$. Si $\dot{\theta}, \Phi \models \exists \dot{\alpha}.\, C$, alors par la règle [C-EXISTS], il existe un type τ sans variables existentielles tel qu'en posant $\dot{\theta}' \equiv \{\dot{\alpha} \mapsto \tau\}$ on ait $\dot{\theta} \oplus \dot{\theta}', \Phi \models C$. Par double application de l'hypothèse d'induction, on a alors $\emptyset, \Phi \models \dot{\theta}'(\dot{\theta}(C))$ et $\dot{\theta}', \Phi \models \dot{\theta}(C)$. On en déduit $\emptyset, \Phi \models \exists \dot{\alpha}.\, \dot{\theta}(C)$ et $\emptyset, \Phi \models \dot{\theta}(\exists \dot{\alpha}.\, C)$. La preuve de la réciproque est similaire.

□

Corollaire 3.3.5. *Soit une contrainte C, une substitution $\dot{\theta}$ et un environnement de typage généralisé Φ. Si $\emptyset, \Phi \models \dot{\theta}(C)$ alors $\emptyset, \Phi \models \exists \bar{\dot{\alpha}}.\, C$ où $\bar{\dot{\alpha}} = dom(\dot{\theta})$.*

Lemme 3.3.6. *Pour chacune des règles de réécriture $C_1 \gg C_2$ de la figure 3.1, on vérifie pour tout environnement Φ et toutes substitutions $\dot{\theta}$, si $\dot{\theta}, \Phi \models C_2$ alors $\dot{\theta}, \Phi \models C_1$.*

FIGURE 3.1 Règles de réécriture pour le système de types généralisé

Entiers

$i : \tau \gg \texttt{True}$ *si* $i \vartriangle \tau$ [R-INT-TRUE]

$i : \tau \gg \texttt{False}$ *sinon* [R-INT-FALSE]

Étiquettes

$(\mu, \Phi').\, (\ell : \tau \wedge C) \gg (\mu, \Phi').\, C$ *si* $\forall \dot{\theta}.\, \exists \tau' \in \Phi'(\ell).\, \dot{\theta}(\tau') \preccurlyeq \dot{\theta}(\tau)$ [R-MERGE']

$(\mu, \Phi').\, (\ell : \tau \wedge C) \gg (\mu, \Phi' \cup \{\ell : \tau\}).\, (w' : \tau' \wedge w'' : \tau'' \wedge C)$ [R-BLK-TRUE']
si $\mu(\ell) = \texttt{Blk}(w', w'')$ *et* $\exists \tau' \tau''$ *tels que* $[\tau'; \tau''] \triangledown \tau$

$(\mu, \Phi').\, (\ell : \tau \wedge C) \gg \texttt{False}$ [R-BLK-FALSE']
si $\mu(\ell) = \texttt{Blk}(w', w'')$ *et* $\tau \not\equiv \dot{\alpha}$ *et* $\nexists \tau' \tau''$ *tels que* $[\tau'; \tau''] \triangledown \tau$

$$(\mu, \Phi').\, (\ell : \tau \wedge C) \gg \exists \bar{\dot{\alpha}}. \begin{cases} \tau = \dot{\tau}' \to \dot{\tau}'' \\ (\mu, \Phi' \cup \{\ell : \tau\}).\, (C \wedge \bigwedge_{\ell \in dom(\rho)} \rho(\ell) : \dot{\Gamma}(\ell)) \end{cases} \quad \text{[R-CLOS']}$$
si $\mu(\ell) = \langle \forall \bar{\alpha}.\, (\Gamma \Rightarrow \tau' \to \tau''), \rho \rangle$ *et* $\bar{\dot{\alpha}} \# ftv(\tau) \cup ftv(\Phi') \cup ftv(C)$

Unification

$\exists \bar{\dot{\alpha}}.\, \tau_1 = \tau_2 \wedge C \gg \dot{\theta}(C)$ *si* $\dot{\theta}$ *est le m.g.u de* τ_1 *et* τ_2 *et si* $dom(\dot{\theta}) = \bar{\dot{\alpha}}$ [R-UNIF]

$\exists \bar{\dot{\alpha}}.\, \tau_1 = \tau_2 \wedge C \gg \texttt{False}$ *si* τ_1 *et* τ_2 *n'admettent pas d'unificateur* [R-NUNIF]

Paramétricité

$\exists \bar{\dot{\alpha}}.\, (\mu, \Phi').\, \overline{(w : \dot{\alpha})} \gg \texttt{True}$ [R-UNIV']

Autres

$C \wedge \texttt{False} \gg \texttt{False}$ [R-FALSE]

$C \wedge \texttt{True} \gg C$ [R-TRUE]

$(\mu, \Phi').\, C \gg C$ *si* $fl(C) \# dom(\mu)$ [R-HEAP']

Preuve Les preuves de correction des règles de réécriture sont détaillées dans l'ordre de la figure 3.1.

– Entiers

Cas [R-Int-True]**.** La relation $\vartriangle$ est stable par substitution (propriété 2.3.2). Ainsi, si $i \vartriangle \tau$ alors pour toute substitution $\dot{\theta}$ et tout environnement de typage Φ, on vérifie $\dot{\theta}, \Phi \models i \vartriangle \tau$.

Cas [R-Int-False]**.** La contrainte `False` étant non satisfaisable, la correction de cette règle est triviale.

– Étiquettes

Cas [R-Merge']**.** Vérifions lorsque $\forall \dot{\theta}.\, \exists \tau' \in \Phi'(\ell).\, \dot{\theta}(\tau') \preccurlyeq \dot{\theta}(\tau)$, que pour tout $\dot{\theta}$ et Φ, si $\dot{\theta}, \Phi \models (\mu, \Phi').\, C$ alors $\dot{\theta}, \Phi \models (\mu, \Phi').\, (\ell : \tau \wedge C)$. Avec cette condition d'application, la contrainte $(\mu, \Phi').\, (\ell : \tau)$ est une tautologie (règles [C-Heap], [C-Question] et [W-Label]). L'implication est donc immédiate.

Cas [R-Blk-True']**.** Vérifions, lorsque $\mu(\ell) = \texttt{Blk}(w', w'')$ et qu'il existe deux types τ' et τ'' tels que $[\tau' ; \tau''] \bigtriangledown \tau$, que pour tout $\dot{\theta}$ et Φ si $\dot{\theta}, \Phi \models (\mu, \Phi').\, (w' : \tau' \wedge w'' : \tau'' \wedge C)$ alors $\dot{\theta}, \Phi \models (\mu, \Phi').(\ell : \tau \wedge C)$.

Par définition à l'aide des règles [C-Heap] et [C-Question] du jugement :

$$\dot{\theta}, \Phi \models (\mu, \Phi').\, (w' : \tau' \wedge w'' : \tau'' \wedge C)$$

il existe un environnement de typage Φ'' tel que :

(a) $\Phi''' = \dot{\theta}(\Phi') \cup \{\ell : \texttt{T}(\dot{\theta}(\tau), \ldots)\} \cup \Phi''$ et $dom(\Phi''') = dom(\mu)$;

(b) $\forall \ell \in dom(\mu).\, \forall \tau'' \in \Phi''(\ell).\, \Phi \oplus \Phi''' \vdash \mu(\ell) : \tau''$;

(c) $\dot{\theta}, \Phi \oplus \Phi''' \models C$

(d) $\Phi \oplus \Phi''' \vdash w' : \dot{\theta}(\tau')$

(e) $\Phi \oplus \Phi''' \vdash w'' : \dot{\theta}(\tau'')$

Par ailleurs, la relation $\bigtriangledown$ étant stable par substitution (propriété 2.3.3), on vérifie à partir de la condition $[\tau' ; \tau''] \bigtriangledown \tau$ que $[\dot{\theta}(\tau') ; \dot{\theta}(\tau'')] \bigtriangledown \dot{\theta}(\tau)$. Les hypothèses (d) et (e) et la règle de typage [H-Blk] permettent d'en déduire :

$$\Phi \oplus \Phi''' \vdash \texttt{Blk}(w', w'') : \dot{\theta}(\tau)$$

Comme par hypothèse $\mu(\ell) = \texttt{Blk}(w', w'')$, ce jugement de typage permet d'étendre Φ'' avec $\{\ell : \dot{\theta}(\tau)\}$ pour compléter les hypothèses (b) et (a) afin de vérifier $\dot{\theta}, \Phi \models (\mu, \Phi').(\ell : \tau)$. L'hypothèse (c) permet alors de conclure :

$$\dot{\theta}, \Phi \models (\mu, \Phi').(\ell : \tau \wedge C)$$

Cas [R-Blk-False']**.** La contrainte `False` étant non satisfaisable, la correction de cette règle est triviale.

Cas [R-Clos']**.** Vérifions lorsque $\mu(\ell) = \langle \forall \alpha_{1..n}.\,(\Gamma \Rightarrow \tau' \to \tau''), \rho\rangle$ et $ftv(\tau) \cup ftv(\Phi') \cup ftv(C) \# \dot{\alpha}_{1..n}$ que pour tout $\dot{\theta}$ et Φ si $\dot{\theta}, \Phi \models \exists \dot{\alpha}_{1..n}.(\tau = \dot{\tau}' \to \dot{\tau}'' \wedge (\mu, \Phi' \cup \{\ell : \tau\}).\,(C \wedge \bigwedge_{\ell \in dom(\rho)} \rho(\ell) : \Gamma(\ell)))$ alors $\dot{\theta}, \Phi \models (\mu, \Phi').(\ell : \tau \wedge C)$. Par définition à l'aide des règles [C-Exists], [C-Equal], [C-Heap] et [C-Question] de :

$$\dot{\theta}, \Phi \models \exists \dot{\alpha}_{1..n}. \begin{cases} \tau = \dot{\tau}' \to \dot{\tau}'' \\ (\mu, \Phi' \cup \{\ell : \tau\}).\,(C \wedge \bigwedge_{\ell \in dom(\rho)} \rho(\ell) : \Gamma(\ell)) \end{cases}$$

on obtient des types $\tau_{1..n}$ et un environnement de typage Φ'' tel que :

(a) $\dot{\theta}' = \dot{\theta} \oplus \{\dot{\alpha}_i \mapsto \tau_i\}_{i=1..n}$

(b) $\dot{\theta}(\tau) = \dot{\theta}'(\dot{\tau}') \to \dot{\theta}'(\dot{\tau}'')$

(c) $\Phi''' = \dot{\theta}'(\Phi') \cup \{\ell : \dot{\theta}'(\tau)\} \cup \Phi''$ et $dom(\Phi''') = dom(\mu)$

(d) $\forall \ell \in dom(\mu).\,\forall \tau'' \in \Phi''(\ell).\,\Phi \oplus \Phi''' \vdash \mu(\ell) : \tau''$

(e) $\dot{\theta}', \Phi \oplus \Phi''' \models C$

(f) $\forall x \in dom(\rho).\,\Phi \oplus \Phi''' \vdash \rho(x) : \dot{\theta}'(\dot{\Gamma}(x))$

L'hypothèse (b) permet de vérifier $\Gamma \Rightarrow \tau' \to \tau'' \preccurlyeq \dot{\theta}'(\dot{\Gamma}) \Rightarrow \dot{\theta}'(\dot{\tau}') \to \dot{\theta}'(\dot{\tau}'')$. On en déduit à l'aide de l'hypothèse (f) et de la règle de typage [H-Clos] :

$$\Phi \oplus \Phi''' \vdash \langle \forall \alpha_{1..n}.\,(\Gamma \Rightarrow \tau' \to \tau''), \rho\rangle : \dot{\theta}'(\tau)$$

Comme par hypothèse d'application $\mu(\ell) = \langle \forall \bar{\alpha}.\,(\Gamma \Rightarrow \tau' \to \tau''), \rho\rangle$, ce jugement permet d'étendre Φ'' avec $\{\ell : \dot{\theta}'(\tau)\}$ pour compléter les hypothèses (d) et (c) afin de vérifier $\dot{\theta}', \Phi \models (\mu, \Phi').\,(\ell : \tau)$. Comme par hypothèse d'application $\dot{\alpha}_{1..n} \# ftv(\tau) \cup ftv(\Phi') \cup ftv(C)$, on peut restreindre le domaine de $\dot{\theta}'$ et, avec l'aide de l'hypothèse (e), en conclure :

$$\dot{\theta}, \Phi \models (\mu, \Phi').(\ell : \tau \wedge C)$$

– Unification

Cas [R-Unif]**.** La correction de cette règle découle du corollaire 3.3.5.

Cas [R-NUnif]**.** La contrainte `False` étant invalide, la correction de cette règle est triviale.

– Paramétricité

Cas [R-Univ]**.** Il suffit d'utiliser le type universel $\star$ pour instancier $\bar{\alpha}$.

– Autres

Cas [R-False]**.** La contrainte `False` étant invalide, la correction de cette règle est triviale.

Cas [R-True]**.** La contrainte `True` est une tautologie.

Cas [R-Heap']. Vérifions, lorsque $fl(C) \# dom(\mu)$, que pour tout $\dot{\theta}$ et Φ si $\dot{\theta}, \Phi \models C$ alors $\dot{\theta}, \Phi \models (\mu, \Phi').\, C$. Il suffit pour cela d'utiliser [C-Heap] avec un environnement Φ'' tel que $\Phi''(\ell) = \emptyset$ ou $\Phi''(\ell) = \{\star\}$. □

Théorème 3.3.7 (Correction). *Soit un tas μ, clos et ne contenant pas de valeur modifiable. Soit une adresse-mémoire $\ell \in dom(\mu)$ et un type τ. Si en utilisant les règles de la figure 3.1, la contrainte $(\mu, \emptyset).(\ell : \tau)$ se réécrit vers* `True` *alors il existe un environnement de typage généralisé Φ tel que $\mu : \Phi$ et $\Phi \vdash \ell : \tau$.*

Preuve Le lemme 3.3.6 permet de vérifier que si une contrainte initiale $(\mu, \emptyset).\, (\ell : \tau)$ se réécrit vers `True` alors cette contrainte est valide. La validité de cette contrainte lorsque le tas μ est clos implique l'existence d'un environnement de typage généralisé Φ tel que $\mu : \Phi$ et $\Phi \vdash \ell : \tau$ (lemme 3.3.1). □

3.3.4 Semi-complétude

Le contre-exemple 3.3.5 a montré qu'en présence de certains cycles le système de réécriture de la figure 3.1 pouvait ne pas terminer, y compris en présence d'un tas correctement typé. L'algorithme correspondant n'est donc pas complet. Cette section va néanmoins vérifier la semi-complétude de ce système de réécriture, c'est-à-dire si une contrainte C se réécrit vers `False` alors C est non satisfaisable. Pour cette preuve, la notion de satisfaisabilité utilisée jusqu'à maintenant n'est pas suffisante et il est nécessaire [6] de caractériser le sous-ensemble des contraintes satisfaisables pour lesquelles la satisfaisabilité de la construction $(\mu, \Phi').\, C$ implique aussi la compatibilité de μ avec Φ'. Cette restriction sera appelé semi-satisfaisabilité et sera notée $\dot{\theta}, \Phi \models^{s} C$; elle est utile uniquement pour les étapes intermédiaires de la preuve : en particulier, elle ne modifie pas ni la notion de validité d'une contrainte initiale $(\mu, \emptyset).\, (w : \tau)$, ni celle des formes normales `True` et `False`. Dans le système inductif, les contraintes semi-satisfaisables peuvent être caractérisées en remplaçant la règle [C-Heap] par la règle ci-dessous.

[CS-Heap]
$$\frac{dom(\dot{\theta}(\Phi') \cup \Phi'') = dom(\mu) \qquad \mu : \Phi \oplus (\dot{\theta}(\Phi') \cup \Phi'') \qquad \dot{\theta}, \Phi \oplus (\dot{\theta}(\Phi') \cup \Phi'') \models^{s} C}{\dot{\theta}, \Phi \models^{s} (\mu, \Phi').\, C}$$

Lemme 3.3.8. *Pour chacune des règles de réécriture $C_1 \gg C_2$ de la figure 3.1, on vérifie pour toute substitution $\dot{\theta}$ et tout environnement Φ, si $\dot{\theta}, \Phi \models C_1$ alors $\dot{\theta}, \Phi \models C_2$.*

Preuve Les preuves de semi-complétude des règles de réécriture sont détaillées dans l'ordre de la figure 3.1.

– Entiers

Cas [R-Int-True]. La contrainte `True` est une tautologie.

6. La nécessité d'introduire cette restriction de la notion de satisfaisabilité pour effectuer la preuve de semi-complétude est discuté à la fin de la section.

Cas [R-Int-False]**.** Lorsque $\tau \equiv \mathtt{T}(\tau_1\,,\,\ldots\,,\tau_n)$, la relation $\vartriangle$ étant définie par cas sur le constructeur de types, si $i \vartriangle \mathtt{T}(\tau_1\,,\,\ldots\,,\tau_n)$ n'est pas vérifié, alors la contrainte $i : \mathtt{T}(\tau_1\,,\,\ldots\,,\tau_n)$ est invalide indépendamment de la substitution $\dot{\theta}$. Lorsque $\tau \equiv \alpha$, la relation $i \vartriangle \alpha$ n'étant jamais vérifiée, la contrainte $i : \alpha$ est invalide.

– Étiquettes

Cas [R-Merge']**.** L'implication est immédiate.

Cas [R-Blk-True']**.** Vérifions, lorsque $\mu(\ell) = \mathtt{Blk}(w', w'')$ et qu'il existe deux types τ' et τ'' tels que $[\tau'\,;\,\tau''] \triangledown \tau$, que pour tout $\dot{\theta}$ et Φ si $\dot{\theta}, \Phi \models^{s} (\mu, \Phi').(\ell : \tau \wedge C)$ alors $\dot{\theta}, \Phi \models^{s} (\mu, \Phi').\,(w' : \tau' \wedge w'' : \tau'' \wedge C)$.

Par définition à l'aide des règles [CS-Heap], [C-Question] de :

$$\dot{\theta}, \Phi \models^{s} (\mu, \Phi').(\ell : \tau \wedge C)$$

il existe un environnement de typage Φ'' tel que :

(a) $\Phi''' = \dot{\theta}(\Phi') \cup \Phi''$ et $dom(\Phi''') = dom(\mu)$

(b) $\mu : \Phi \oplus \Phi'''$

(c) $\Phi \oplus \Phi''' \vdash \ell : \dot{\theta}(\tau)$

(d) $\dot{\theta}, \Phi \oplus \Phi''' \models^{s} C$

Par définition à l'aide de la règle [W-Label] de l'hypothèse (c) et par l'hypothèse (a), on obtient un type $\tau''' \in \Phi'''(\ell)$ tel que $\tau''' \preccurlyeq \dot{\theta}(\tau)$. L'hypothèse (b) et la condition d'application $\mu(\ell) = \mathtt{Blk}(w',\, w'')$ permettent alors de vérifier $\Phi \oplus \Phi''' \vdash \mathtt{Blk}(w',\, w'') : \tau'''$ et le lemme de substitution (lemme 3.1.1) permet d'en déduire :

$$\Phi \oplus \Phi''' \vdash \mathtt{Blk}(w',\, w'') : \dot{\theta}(\tau)$$

La décomposition d'un type par la relation $\triangledown$ étant unique (propriété 2.3.4) et stable par substitution (propriété 2.3.3), par définition à l'aide de la règle [H-Blk] de ce dernier jugement de typage, on a $\Phi \oplus \Phi''' \vdash w' : \dot{\theta}(\tau')$ et $\Phi \oplus \Phi''' \vdash w'' : \dot{\theta}(\tau'')$ où τ' et τ'' sont les types de la condition d'application $[\tau'\,;\,\tau''] \triangledown \tau$. Ces deux jugements et les hypothèses (a), (b) et (d) permettent alors de vérifier $\dot{\theta}, \Phi \models^{s} (\mu, \Phi').\,(w' : \tau' \wedge w'' : \tau'' \wedge C)$. Le jugement de typage $\Phi \oplus \Phi''' \vdash \mu(\ell) : \dot{\theta}(\tau)$ permet d'étendre Φ' pour conclure :

$$\dot{\theta}, \Phi \models^{s} (\mu, \Phi' \cup \{\ell : \tau\}).\,(w' : \tau' \wedge w'' : \tau'' \wedge C)$$

Cas [R-Blk-False']**.** S'il n'existe pas de types τ_1 et τ_2 tels que $[\tau_1\,;\,\tau_2] \triangledown \tau$, alors aucun environnement de typage Φ'' ne permettra de vérifier $\Phi'' \vdash \mathtt{Blk}(w_1,\, w_2) : \tau$. Ainsi, une contrainte de la forme $(\mu, \Phi').\,(\ell : \tau)$ où $\mu(\ell) = \mathtt{Blk}(w_1,\, w_2)$ est non satisfaisable.

Cas [R-Clos']. Vérifions, lorsque $\mu(\ell) = \langle \forall \alpha_{1..n}.\, (\Gamma \Rightarrow \tau' \to \tau''), \rho \rangle$ et $\bar{\dot{\alpha}} \# ftv(\tau) \cup ftv(\Phi') \cup ftv(C)$, que pour tout $\dot{\theta}$ et Φ si $\dot{\theta}, \Phi \models^s (\mu, \Phi').(\ell : \tau \wedge C)$ alors $\dot{\theta}, \Phi \models^s \exists \dot{\alpha}_{1..n}.(\tau = \dot{\tau}' \to \dot{\tau}'' \wedge (\mu, \Phi' \cup \{\ell : \tau\}).\, (C \wedge \bigwedge_{\ell \in dom(\rho)} \rho(\ell) : \Gamma(\ell)))$.
Par définition à l'aide des règles [CS-Heap], [C-Question] de :

$$\dot{\theta}, \Phi \models^s (\mu, \Phi').(\ell : \tau \wedge C)$$

il existe un environnement de typage Φ'' tel que :
(a) $\Phi''' = \dot{\theta}(\Phi') \cup \Phi''$ et $dom(\Phi''') = dom(\mu)$
(b) $\mu : \Phi \oplus \Phi'''$
(c) $\Phi \oplus \Phi''' \vdash \ell : \dot{\theta}(\tau)$
(d) $\dot{\theta}, \Phi \oplus \Phi''' \models^s C$

Par définition à l'aide de la règle [W-Label] de l'hypothèse (c) on obtient un type $\tau''' \in \Phi'''(\ell)$ tel que $\tau''' \preccurlyeq \dot{\theta}(\tau)$. L'hypothèse (b) et la condition d'application $\mu(\ell) = \langle \forall \alpha_{1..n}.\, (\Gamma \Rightarrow \tau' \to \tau''), \rho \rangle$ permettent alors de vérifier $\Phi \oplus \Phi''' \vdash \langle \forall \alpha_{1..n}.\, (\Gamma \Rightarrow \tau' \to \tau''), \rho \rangle : \tau'''$ et le lemme de substitution (lemme 3.1.1) permet d'en déduire :

$$\Phi \oplus \Phi''' \vdash \langle \forall \alpha_{1..n}.\, (\Gamma \Rightarrow \tau' \to \tau''), \rho \rangle : \dot{\theta}(\tau)$$

Par définition à l'aide de la règle [H-Clos] de ce jugement de typage, on a une substitution θ' telle que $\tau' \to \tau'' \preccurlyeq_{\theta'} \dot{\theta}(\tau)$ et telle que $\Phi \oplus \Phi''' \vdash \rho : \theta'(\Gamma)$. En pointant les variables $\dot{\alpha}_{1..n}$ et en remarquant que l'on peut choisir des $\bar{\dot{\alpha}}$ telles que $\bar{\dot{\alpha}} \# dom(\dot{\theta}) \cup fev(img(\dot{\theta}))$, on a alors $\dot{\theta}(\dot{\theta}'(\dot{\tau}' \to \dot{\tau}'')) = \dot{\theta}(\dot{\theta}'(\tau))$ et $\Phi \oplus \Phi''' \vdash \rho : \dot{\theta}(\dot{\theta}'(\dot{\Gamma}))$. Cela permet de compléter les hypothèses (a), (b) et (d) pour vérifier $\dot{\theta}, \Phi \models^s \dot{\theta}'(\dot{\tau}') \to \dot{\theta}'(\dot{\tau}'') = \dot{\theta}'(\tau)$ et $\dot{\theta}, \Phi \models^s (\mu : \Phi').\, (C \wedge \bigwedge_{\ell \in dom(\rho)} \rho(\ell) : \dot{\theta}'(\dot{\Gamma}(\ell)))$. Comme par hypothèse d'application $\dot{\alpha}_{1..n} \# ftv(\tau) \cup ftv(\Phi') \cup ftv(C)$, le corollaire 3.3.5 permet d'en déduire :

$$\dot{\theta}, \Phi \models^s \exists \dot{\alpha}_{1..n}. \begin{cases} \tau = \dot{\tau}' \to \dot{\tau}'' \\ (\mu, \Phi').\, (C \bigwedge_{\ell \in dom(\rho)} \rho(\ell) : \dot{\Gamma}(\ell)) \end{cases}$$

Par ailleurs, la dérivation de typage $\Phi \oplus \Phi''' \vdash \langle \forall \alpha_{1..n}.\, (\Gamma \Rightarrow \tau' \to \tau''), \rho \rangle : \dot{\theta}(\tau)$ permet de conclure :

$$\dot{\theta}, \Phi \models^s \exists \dot{\alpha}_{1..n}. \begin{cases} \tau = \dot{\tau}' \to \dot{\tau}'' \\ (\mu, \Phi' \cup \{\ell : \tau\}).\, (C \wedge \bigwedge_{\ell \in dom(\rho)} \rho(\ell) : \dot{\Gamma}(\ell)) \end{cases}$$

– Unification

Cas [R-Unif]. La complétude de cette règle découle du corollaire 3.3.5 et des propriétés de l'unificateur le plus général [Robinson, 1965].

Cas [R-NUnif]. Si les types τ_1 et τ_2 n'admettent pas d'unificateur, il n'existe aucune substitution $\dot{\theta}$ tel que $\dot{\theta}(\tau_1) = \dot{\theta}(\tau_2)$. La contrainte $\tau_1 = \tau_2$ est donc invalide.

– Paramétricité

La contrainte `True` étant une tautologie, l'implication est toujours vérifiée.

– Autres

Cas [R-False]. Une contrainte contenant `False` est toujours invalide.

Cas [R-True]. La contrainte `True` est une tautologie.

Cas [R-Heap]. Par application directe du lemme 3.3.3. □

Théorème 3.3.9 (Semi-complétude). *Soit un tas μ, clos et ne contenant pas de valeur modifiable. Soit une adresse-mémoire $\ell \in dom(\mu)$ et un type τ. Si en utilisant les règles de la figure 3.1, la contrainte $(\mu, \emptyset).(\ell : \tau)$ se réécrit vers* `False` *alors il n'existe aucun environnement de typage généralisé Φ tel que $\mu : \Phi$ et $\Phi \vdash \ell : \tau$.*

Preuve Le lemme 3.3.8 permet de vérifier que si une contrainte initiale $(\mu, \emptyset).(\ell : \tau)$ se réécrit vers `False`, alors cette contrainte est invalide. L'invalidité de cette contrainte lorsque que le tas μ est clos implique par contraposition du lemme 3.3.1 l'inexistence d'un environnement de typage généralisé Φ tel que $\mu : \Phi$ et $\Phi \vdash \ell : \tau$. □

Satisfaisabilité et semi-satisfaisabilité Avec la notion de satisfaisabilité utilisée pour montrer la correction du système de réécriture, une contrainte $(\mu, \Phi').(\ell : \texttt{List(Bool)})$ où $\texttt{List}(\alpha) \in \Phi'(\ell)$ est une tautologie, indépendamment de la valeur $\mu(\ell)$. Cela peut être vérifié par la suite de réécriture :

$$\begin{array}{rcl} & & (\mu, \Phi').(\ell : \texttt{List(Bool)}) \\ \text{[R-Merge']} & \gg & (\mu, \Phi').\texttt{True} \\ \text{[R-Heap']} & \gg & \texttt{True} \end{array}$$

Néanmoins, si $\mu(\ell) = \texttt{Blk}(23, 0)$ alors en appliquant d'abord la règle de réécriture [R-Blk-True'], cette contrainte peut aussi se réécrire vers `False`.

$$\begin{array}{rcl} & & (\mu, \Phi').(\ell : \texttt{List(Bool)}) \\ \text{[R-Blk-True']}\text{[R-Int-False]} & \gg & (\mu, \Phi' \cup \{\ell : \texttt{List(Bool)}\}).(\texttt{False} \wedge 0 : \texttt{List(Bool)}) \\ \text{[R-False]} & \gg & \texttt{False} \end{array}$$

Le système pouvant réécrire une contrainte valide vers `False`, cet exemple suggère que le système de réécriture n'est pas semi-complet. Mais cet exemple est artificiel du point de vue de l'algorithme de vérification : il contient une hypothèse de type incompatible avec le tas, ce qui est impossible dans notre système de réécriture si la contrainte initiale est valide.

Avec la notion de satisfaisabilité utilisée pour montrer la correction, il est de fait impossible de vérifier la semi-complétude des règles [R-Blk-True'] et [R-Clos']. Néanmoins, il devient possible de le faire en leur ajoutant la condition d'application suivante « le type attendu n'est instance d'aucune des hypothèses mémorisées. » Dans l'exemple ci-dessus, le type attendu `List(Bool)` étant une instance de l'hypothèse $\texttt{List}(\alpha)$, cette

condition interdit l'application de la règle [R-Blk-True']. Le principe de la preuve de semi-complétude dans le cas de la règle [R-Blk-True'] est alors le suivant : lorsque $\mu(\ell) = \texttt{Blk}(w_1, w_2)$ et $[\tau_1 ; \tau_2] \triangledown \tau$, pour pouvoir vérifier que la satisfaisabilité de la contrainte $(\mu, \Phi').(\ell : \tau \wedge C)$ implique celle de $(\mu, \Phi').(w_1 : \tau_1 \wedge w_2 : \tau_2 \wedge C)$, il faut que la satisfaisabilité de la première contrainte implique l'existence d'un environnement de typage Φ''' tel que $\Phi''' \vdash \texttt{Blk}(w_1, w_2) : \tau$ — cette dérivation de typage pouvant ensuite être décomposée en $\Phi''' \vdash w_1 : \tau_1$ et $\Phi''' \vdash w_2 : \tau_2$; la notion de satisfaisabilité choisie pour la construction $(\mu, \Phi').C$ à la section 3.3.3 implique l'existence d'un tel environnement uniquement lorsque τ n'est instance d'aucune des hypothèses de $\Phi'(\ell)$.

Dans le cadre d'un tas sans fermeture, et donc sans variables existentielles, cette condition de « non instanciation » est la négation de la condition d'application de la règle [R-Merge'], elle peut donc être ajoutée comme condition d'application à la [R-Blk-True'] sans perdre de généralité. En présence de variables existentielles, ces conditions d'instanciation et de non instanciation doivent être valables pour toutes les substitutions possibles des variables existentielles. Elles ne sont alors plus la négation l'une de l'autre et la résolution de certaines contraintes sera bloquée sur une forme non triviale. Par exemple, une contrainte $\exists\dot{\alpha}.(\mu, \{\ell : \{\texttt{Int} \to \texttt{Int}\}\}).(\ell : \dot{\alpha} \to \dot{\alpha})$ ne peut plus se réécrire : dans une substitution $\{\dot{\alpha} \mapsto \texttt{Int}\}$, il faudrait appliquer la règle [R-Merge'], dans toutes autres substitutions il faudrait appliquer la règle [R-Blk-True']. Ajouter une condition d'application aux règles [R-Blk-True'] et [R-Clos'] nous a alors semblé trop restrictif et nous avons préféré caractériser le sous ensemble des contraintes satisfaisables pour lesquelles les hypothèses de types sont compatibles avec le tas.

3.4 Restriction au système de types original

Les sections 3.2 et 3.3 ont permis la définition progressive d'un système de réécriture permettant dans le cadre du système de types généralisé — c'est-à-dire basé sur des environnements de typage Φ associant un ensemble de schémas de types à chaque adresse-mémoire — de vérifier la compatibilité avec un type d'un tas sans bloc modifiables. Ce système est résumé dans la figure 3.1. Nous avons montré que ce système de réécriture vérifie des propriétés de correction et de semi-complétude et nous avons montré comment certains cycles peuvent malheureusement produire des suites infinies de réécritures. L'objet de cette section est d'étudier les restrictions à apporter à ce système de réécriture pour qu'il accepte uniquement les valeurs typables dans le système de types original — c'est-à-dire basé sur des environnement de typage simple Ψ associant un seul schéma de type à chaque adresse-mémoire.

La section 3.4.1 va étudier dans un premier temps les principes nécessaires à cette restriction en l'absence de fermetures, nous montrons que cette restriction permet de retrouver la propriété de terminaison. La section 3.4.2 étendra ces principes aux tas contenant des fermetures. Nous montrerons que la présence de certaines fermetures fait à nouveau perdre la propriété de terminaison. Néanmoins, les sections 3.4.3 et 3.4.4 montrerons que ce système de réécriture vérifie respectivement les propriétés de correction et de semi-complétude vis-à-vis du système de types original.

3.4.1 Anti-unification

Dans le système de réécriture basé sur le système de types généralisé (figure 3.1), il est possible de vérifier indépendamment la compatibilité d'une valeur partagée avec chacun des types attendus. Pour construire un système de réécriture basé sur le système de types original, ces vérifications indépendantes ne suffisent pas, il est aussi nécessaire qu'il existe un type plus général que tous les types attendus qui soit compatible avec la valeur allouée. Une possibilité est de vérifier la compatibilité d'une valeur partagée une seule fois avec l'*anti-unificateur principal* de l'ensemble des types attendus [Plotkin, 1970; Huet, 1976].

Anti-unificateur principal On appelle *anti-unificateur* d'un ensemble de types, un type plus général que tous les types de l'ensemble ; par exemple α est un anti-unificateur trivial pour tous les ensembles de types. Parmi les anti-unificateurs d'un ensemble de types, on en distingue un particulier, instance de tous les autres anti-unificateurs, il est appelé anti-unificateur principal. Ainsi, suivant le principe du lemme de substitution (lemme 3.1.1), une valeur allouée h sera compatible avec tous les types d'un ensemble de types ssi elle est compatible avec l'anti-unificateur principal de cet ensemble. Pour ce chapitre, nous reprenons la définition de [Huet, 1976] basée sur une bijection $au(\tau_1, \tau_2)$ de l'ensemble des paires de types distincts vers un sous-ensemble des variables universelles. On notera $\tau_1 \curlywedge \tau_2$, l'anti-unificateur de τ_1 et τ_2 calculé selon les deux règles ci-dessous :

$$\begin{array}{rcll} \mathtt{T}(\tau_1, \ldots, \tau_n) \curlywedge \mathtt{T}'(\tau'_1, \ldots, \tau'_n) & = & \mathtt{T}(\tau_1 \curlywedge \tau'_1, \ldots, \tau_n \curlywedge \tau'_n) & \textit{si } \mathtt{T} \equiv \mathtt{T}' \\ \tau_1 \curlywedge \tau_2 & = & au(\tau_1, \tau_2) & \textit{dans tous les autres cas} \end{array}$$

Cette définition de l'anti-unification est commutative et associative après renommage des variables de type produites. Ainsi, elle s'étend naturellement à un ensemble de types et on notera $\curlywedge\{\tau_{1..n}\}$ ou $\tau_1 \curlywedge \ldots \curlywedge \tau_n$ l'anti-unificateur principal des types τ_1 à τ_n. Par exemple $\curlywedge\{\mathtt{List(Int)}; \mathtt{List(Bool)}; \mathtt{List(Option(Bool))}\} = \mathtt{List}(\alpha)$.

Lemme 3.4.1. *Étant donnés trois types τ, τ_1 et τ_2, on vérifie :*

Anti-unification $(\tau_1 \curlywedge \tau_2) \preccurlyeq \tau_1$ *et* $(\tau_1 \curlywedge \tau_2) \preccurlyeq \tau_2$ *;*

Principalité *si* $\tau \preccurlyeq \tau_1$ *et* $\tau \preccurlyeq \tau_2$ *alors* $\tau \preccurlyeq (\tau_1 \curlywedge \tau_2)$*.*

Anti-unification et fermetures En présence de fermetures dans un tas, il n'est pas toujours possible de calculer l'anti-unificateur des types attendus pour une valeur partagée. En effet, la section 3.3.1 a montré que la vérification d'une fermeture peut nécessiter l'introduction de variables existentielles dans la syntaxe des contraintes et l'anti-unificateur principal de types contenant de telles variables dépend de leur instanciation future. Par exemple, lorsque les types attendus pour une fermeture seront $\mathtt{Int} \to \dot{\alpha}$ et $\dot{\beta} \to \mathtt{Bool}$, où $\dot{\alpha}$ et $\dot{\beta}$ sont des variables existentielles, l'anti-unificateur sera différent selon leur instanciation : si elles sont respectivement instanciées avec `Int` et `Bool`, l'anti-unificateur sera $\gamma \to \gamma$; si elles sont respectivement instanciées avec `Bool` et `Int`, ce sera $\mathtt{Int} \to \mathtt{Bool}$.

Pour cela, on suppose pour cette section que le tas ne contient pas de fermeture et que les types ne contiennent pas de variables existentielles. Dans un second temps, la section 3.4.2 réintroduira les fermetures.

Règle de réécriture En l'absence de fermetures et de variables existentielles, la fonction de calcul de l'anti-unificateur principal permet d'étendre le système de réécriture, en remplaçant un ensemble de contraintes de type par leur anti-unificateur principal :

$$\ell : \tau_1 \wedge \ldots \wedge \ell : \tau_n \gg \ell : \curlywedge\{\tau_{1..n}\} \qquad \text{[R-AntiUnif]}$$

En l'absence de cycle, il suffit alors d'interdire l'application de la règle [R-Blk-True] lorsque tous les types attendus pour le bloc n'ont pas été collectés et anti-unifiés, pour selon toute vraisemblance obtenir un système correct et complet. Plus précisément cette restriction est obtenu en ajoutant une condition d'application supplémentaire $\ell \notin fl(C)$ à la règle [R-Blk-True].

$$\mu.(\ell : \tau \wedge C) \gg \mu.(w' : \tau' \wedge w'' : \tau'' \wedge C) \qquad \text{[R-Blk-True]}$$
$$si\ \mu(\ell) = \texttt{Blk}(w', w'')\ et\ [\tau'; \tau''] \bigtriangledown \tau\ et\ \ell \notin fl(C)$$

Exemple 3.4.1. *Étant donné le tas* $\mu \equiv \{\ell_0 \mapsto \texttt{Blk}(\ell_1, \ell_1)\,;\, \ell_1 \mapsto \texttt{Blk}(0, 0)\}$ *et si le type attendu est* $(\texttt{List}(\texttt{Option}(\texttt{Int})) \times \texttt{List}(\texttt{Option}(\texttt{Bool})))$*, une suite possible de réécritures est :*

$$\begin{array}{rcl}
 & & \mu.\,(\ell_0 : (\texttt{List}(\texttt{Option}(\texttt{Int})) \times \texttt{List}(\texttt{Option}(\texttt{Bool}))) \\
\text{[R-Blk-True]} & \gg & \mu.\,(\ell_1 : \texttt{List}(\texttt{Option}(\texttt{Int})) \wedge \ell_1 : \texttt{List}(\texttt{Option}(\texttt{Bool}))) \\
\text{[R-AntiUnif]} & \gg & \mu.\,(\ell_1 : \texttt{List}(\texttt{Option}(\alpha))) \\
\text{[R-Blk-True]} & \gg & \mu.\,(0 : \texttt{Option}(\alpha) \wedge 0 : \texttt{List}(\texttt{Option}(\alpha))) \\
\text{[R-Int-True]} & \gg & \mu.\,(\texttt{True} \wedge 0 : \texttt{List}(\texttt{Option}(\alpha))) \\
\text{[R-Int-True]} & \gg & \mu.\,(\texttt{True} \wedge \texttt{True}) \\
 & \gg^* & \texttt{True}
\end{array}$$

Si le type attendu est $(\texttt{List}(\texttt{Int}) \times \texttt{List}(\texttt{Bool}))$*, une suite possible de réécritures est :*

$$\begin{array}{rcl}
 & & \mu.\,(\ell_0 : (\texttt{List}(\texttt{Int}) \times \texttt{List}(\texttt{Bool}))) \\
\text{[R-Blk-True]} & \gg & \mu.\,(\ell_1 : \texttt{List}(\texttt{Int}) \wedge \ell_1 : \texttt{List}(\texttt{Bool})) \\
\text{[R-AntiUnif]} & \gg & \mu.\,(\ell_1 : \texttt{List}(\alpha)) \\
\text{[R-Blk-True]} & \gg & \mu.\,(0 : \alpha \wedge 0 : \texttt{List}(\alpha)) \\
\text{[R-Int-False]} & \gg & \mu.\,(\texttt{False} \wedge 0 : \texttt{List}(\alpha)) \\
 & \gg^* & \texttt{False}
\end{array}$$

Hypothèse(s) de types Dans le cadre du système de types original, il suffit en présence de cycles dans le tas de ne mémoriser qu'une seule hypothèse de type pour chaque adresse. Pour cela, les environnements de typage généralisés utilisés dans la construction $(\mu, \Phi').\,C$ peuvent être remplacés par des environnements de typage simples $(\mu, \Psi').\,C$; et la règle [R-Blk-True'] définie à la section 3.3.2 dans le cadre du système de types généralisé doit être remplacée par les deux règles ci-dessous. La première s'applique

lorsqu'aucune hypothèse de type n'a déjà été mémorisée pour un bloc partagé, c'est-à-dire lors de la première vérification de ce bloc ; la seconde s'applique lorsqu'une hypothèse de type a déjà été mémorisée, par exemple lorsqu'une contrainte de type provenant de l'intérieur d'un cycle n'est pas une instance de l'hypothèse initiale. Dans le second cas, il est alors nécessaire de calculer l'anti-unificateur principal de l'hypothèse mémorisée et de la nouvelle contrainte avant de vérifier à nouveau la valeur.

$$(\mu, \Psi').(\ell : \tau \wedge C) \gg (\mu, \Psi' \oplus \{\ell : \tau\}).(w' : \tau' \wedge w'' : \tau'' \wedge C) \quad \text{[R-Blk-True"]}$$
$$\text{si } \mu(\ell) = \texttt{Blk}(w', w') \text{ et } \ell \notin dom(\Psi')$$
$$\text{et } [\tau' ; \tau''] \bigtriangledown \tau$$

$$(\mu, \Psi').(\ell : \tau \wedge C) \gg (\mu, \Psi' \oplus \{\ell : \tau'''\}).(w' : \tau' \wedge w'' : \tau'' \wedge C) \quad \text{[R-Blk-True"']}$$
$$\text{si } \Psi'(\ell) \not\preccurlyeq \tau \text{ et } \mu(\ell) = \texttt{Blk}(w', w')$$
$$\text{et } \tau''' = \Psi'(\ell) \curlywedge \tau \text{ et } [\tau' ; \tau''] \bigtriangledown \tau'''$$

Exemple 3.4.2. *Reprenons le contre-exemple 3.3.5 qui avait permis de montrer la non terminaison des versions précédentes de l'algorithme : étant donné le tas* $\mu \equiv \{\ell_0 \mapsto \texttt{Blk}(\ell_0, 0)\}$*, si le type attendu est* $\texttt{NR}(\texttt{Int})$ *où* `NR` *est défini comme* $\texttt{NR}(\alpha) = \texttt{Option}(\texttt{NR}(\alpha \times \alpha))$*, une suite possible de réécritures est :*

$$\begin{array}{rcl}
 & & (\mu, \emptyset).(\ell_0 : \texttt{NR}(\texttt{Int})) \\
\text{[R-Blk-True"]} & \gg & (\mu, \{\ell_0 : \texttt{NR}(\texttt{Int})\}).(\ell_0 : \texttt{NR}(\texttt{Int} \times \texttt{Int}) \wedge 0 : \texttt{Unit}) \\
 & \gg^* & (\mu, \{\ell_0 : \texttt{NR}(\texttt{Int})\}).(\ell_0 : \texttt{NR}(\texttt{Int} \times \texttt{Int})) \\
\text{[R-Blk-True"']} & \gg & (\mu, \{\ell_0 : \texttt{NR}(\alpha)\}).(\ell_0 : \texttt{NR}(\alpha \times \alpha) \wedge 0 : \texttt{Unit}) \\
\text{[R-Merge']} & \gg & (\mu, \{\ell_0 : \texttt{NR}(\alpha)\}).(0 : \texttt{Unit}) \\
 & \gg^* & \texttt{True}
\end{array}$$

Terminaison De manière plus générale que l'exemple ci-dessus, il est possible de montrer la terminaison de ce système de réécriture en remarquant que lors de l'application de la règle [R-Blk-True"'], la condition d'application $\Psi(\ell) \not\preccurlyeq \tau$ implique la diminution du nombre de constructeurs [7] de type dans les hypothèses mémorisées. Le nombre d'applications de la règle [R-Blk-True"'] est alors borné par le nombre de constructeurs de types dans le type mémorisé initialement à l'aide de la règle [R-Blk-True"].

Lemme 3.4.2. *Soit deux types* τ *et* τ'*. Si* $\tau \not\preccurlyeq \tau'$*, alors leur anti-unificateur principal contient strictement moins de constructeurs de types que* τ*.*

Preuve Par induction sur la taille des types. Si $\tau \equiv \texttt{T}(\tau_1, \ldots, \tau_n)$ et $\tau' \equiv \texttt{T}'(\ldots)$ ou $\tau' \equiv \alpha$ lorsque $\texttt{T} \neq \texttt{T}'$ alors le résultat est immédiat, car $\tau \curlywedge \tau'$ est une variable de type. Sinon, $\tau' \equiv \texttt{T}(\tau'_1, \ldots, \tau'_n)$ et, puisque $\tau \not\preccurlyeq \tau'$, il existe au moins un des sous termes pour lequel $\tau_i \not\preccurlyeq \tau'_i$. L'hypothèse d'induction permet alors de conclure. □

Ainsi, en définissant la taille d'une contrainte comme un triplet constitué :

- du nombre d'adresses-mémoire pour lesquelles aucune hypothèse de type n'a encore été mémorisée ;

7. On ne comptent donc pas les variables de types.

- du nombre de constructeurs de types dans toutes les hypothèses ;
- du nombre de contraintes de type restant à vérifier.

alors, chacune des règles de réécriture fait décroître, selon l'ordre lexicographique, la taille de la contrainte à résoudre.

En présence d'expressions cycliques de types ce résultat de terminaison ne semble plus valable, car l'anti-unification ne fait plus nécessairement décroître la taille d'un type. Par exemple, pour $\tau \equiv \mathtt{T}(\mathtt{Int} \times \tau)$ et $\tau' \equiv \mathtt{T}(\mathtt{Bool} \times \mathtt{T}(\mathtt{Int} \times \tau'))$, alors $\tau \curlywedge \tau' \equiv \mathtt{T}(\alpha \times \mathtt{T}(\mathtt{Int} \times \tau \curlywedge \tau'))$ contient plus de constructeurs de types que τ.

3.4.2 Ensemble de types homogène

La section précédente a proposé en l'absence de fermeture un mécanisme à base d'anti-unification restreignant l'ensemble des valeurs acceptées par le système de réécriture à celles typables dans le système de types original (section 3.1.1). Cette restriction permet de retrouver un argument de terminaison en présence de cycle. Avant d'établir que le mécanisme proposé permet d'obtenir un système de réécriture correct et complet vis-à-vis du système de types original, cette section étudie les modifications nécessaires à la prise en compte des fermetures.

Si l'on cherche à appliquer directement le même principe en présence de fermetures, et donc de variables existentielles, certaines contraintes vont se retrouver bloquées sur des formes non résolues. L'exemple ci-dessous illustre une telle situation.

Exemple 3.4.3. *Soit le tas :*

$$\begin{array}{lll} \mu \equiv \{ & \ell_0 \mapsto \mathtt{Blk}(\ell_1, \ell_2); & \\ & \ell_1 \mapsto \langle\ \lambda().\, f\ x\ ;\ \{f \mapsto \ell_3\ ;\ x \mapsto 3\}\ \rangle; & \\ & \ell_2 \mapsto \langle\ \lambda().\, f\ x\ ;\ \{f \mapsto \ell_3\ ;\ x \mapsto 0\}\ \rangle; & \\ & \ell_3 \mapsto \langle\ \lambda y.\, (y :: z)\ ;\ \{z \mapsto 0\}\ \rangle & \} \end{array}$$

La valeur 0 *associée à* z *dans l'environnement de la fermeture* $\mu(\ell_3)$ *représente une liste vide ; elle peut être typée* $\mathtt{List}(\alpha)$*. En supposant que le type conservé par le compilateur pour le pointeur de code associé à* $\lambda y.\, (y :: z)$ *est* $\forall\alpha.\, (\{z : \mathtt{List}(\alpha)\} \Rightarrow \alpha \to \mathtt{List}(\alpha))$*, alors ce tas est compatible avec l'environnement de typage simple* Ψ_3 *:*

$$\begin{array}{lcl} \Psi_3 & \equiv & \Psi_2 \oplus \{\ell_3 : \alpha \to \mathtt{List}(\alpha)\} \\ \Psi_2 & \equiv & \Psi_1 \oplus \{\ell_2 : \mathtt{Unit} \to \mathtt{List}(\mathtt{Bool})\} \\ \Psi_1 & \equiv & \Psi_0 \oplus \{\ell_1 : \mathtt{Unit} \to \mathtt{List}(\mathtt{Int})\} \\ \Psi_0 & \equiv & \{\ell_0 : (\mathtt{Unit} \to \mathtt{List}(\mathtt{Int})) \times (\mathtt{Unit} \to \mathtt{List}(\mathtt{Bool}))\} \end{array}$$

Ainsi, si le type attendu pour ℓ_0 *est* $\tau_{\mathtt{Int}} \times \tau_{\mathtt{Bool}}$ *où* $\tau_{\mathtt{Int}} \equiv \mathtt{Unit} \to \mathtt{List}(\mathtt{Int})$ *et* $\tau_{\mathtt{Bool}} \equiv$

$\texttt{Unit} \to \texttt{List}(\texttt{Bool})$), *une suite possible de réécritures est :*

$$
\begin{array}{rcl}
 & & (\mu, \emptyset).\, (\ell_0 : (\tau_{\texttt{Int}} \times \tau_{\texttt{Bool}})) \\
\text{[R-Blk-True"]} & \gg & (\mu, \Psi_0).\, (\ell_1 : \tau_{\texttt{Int}} \wedge \ell_2 : \tau_{\texttt{Bool}}) \\
\text{[R-Clos'][R-Unif]} & \gg & \exists \dot{\alpha}.\, (\mu, \Psi_1).\, \left\{ \begin{array}{l} 3 : \dot{\alpha} \wedge \ell_3 : \dot{\alpha} \to \texttt{List}(\texttt{Int}) \\ \ell_2 : \tau_{\texttt{Bool}} \end{array} \right. \\
\text{[R-Clos'][R-Unif]} & \gg & \exists \dot{\alpha}\dot{\beta}.\, (\mu, \Psi_2).\, \left\{ \begin{array}{l} 3 : \dot{\alpha} \wedge \ell_3 : \dot{\alpha} \to \texttt{List}(\texttt{Int}) \\ 1 : \dot{\beta} \wedge \ell_3 : \dot{\beta} \to \texttt{List}(\texttt{Bool}) \end{array} \right.
\end{array}
$$

À cette étape, la réécriture sera bloquée, car il est impossible d'anti-unifier les types $\dot{\alpha} \to \texttt{List}(\texttt{Int})$ *et* $\dot{\beta} \to \texttt{List}(\texttt{Bool})$ *: le résultat dépend de l'instanciation de* $\dot{\alpha}$ *et* $\dot{\beta}$*.*

Décomposition multiple et anti-unification Une possibilité simple pour résoudre cette contrainte bloquée par une anti-unification est de vérifier, dans un premier temps, séparément la compatibilité de la fermeture $\mu(\ell_3)$ avec les types $\dot{\alpha} \to \texttt{List}(\texttt{Int})$ et $\dot{\beta} \to \texttt{List}(\texttt{Bool})$ — cette vérification permet d'instancier les variables existentielles $\dot{\alpha}$ et $\dot{\beta}$ respectivement avec `Int` et `Bool` —, puis dans un second temps, il est alors possible d'anti-unifier les types $\texttt{Int} \to \texttt{List}(\texttt{Int})$ et $\texttt{Bool} \to \texttt{List}(\texttt{Bool})$ et de vérifier la compatibilité de la fermeture $\mu(\ell_3)$ avec le type obtenu $\gamma \to \texttt{List}(\gamma)$. Cette dernière étape permet de vérifier la compatibilité de la valeur capturée pour z avec $\texttt{List}(\gamma)$.

Cette vérification supplémentaire est nécessaire comme le montre l'exemple suivant. Dans un tas μ' similaire à μ mais où la valeur capturée pour z est le bloc $\texttt{Blk}(1, 0)$, le type $(\tau_{\texttt{Int}} \times \tau_{\texttt{Bool}})$ n'est plus acceptable : dans un premier temps, les vérifications indépendantes de l'environnement capturé dans $\mu(\ell_3)$ avec $\texttt{List}(\texttt{Bool})$ et $\texttt{List}(\texttt{Int})$ réussiront, dans un second temps, la vérification de l'environnement avec leur anti-unificateur $\texttt{List}(\gamma)$ échouera à juste titre.

Décomposition parallèle et ensemble de type Le principe de résolution en présence de variables existentielles décrit ci-dessus a deux désavantages : d'une part il impose une vérification supplémentaire et d'autre part en présence de paramétricité, c'est-à-dire de variables existentielles sur lesquelles ne porte aucune contrainte de type, il ne semble pas garantir l'absence de contraintes bloquées sur une forme non résolue. Une autre possibilité va être de représenter un anti-unificateur non calculable par l'ensemble des types qu'il est censé généraliser, puis de vérifier en parallèle les types de cet ensemble en s'assurant à chaque étape de réécriture que le résultat sera valable pour leur anti-unificateur, indépendamment de l'instanciation future des variables existentielles. Dans le cas des fermetures, le lemme 3.4.7 montrera qu'il suffit d'un côté de vérifier que chacun des types attendus est effectivement instance du type associé au pointeur de code de la fermeture et d'un autre côté de vérifier que l'environnement d'évaluation capturé est compatible avec l'anti-unificateur de l'ensemble des environnements de typage attendus. Si cet anti-unificateur lui non plus n'est pas calculable immédiatement, il sera de la même manière représenté sous la forme d'un ensemble de types. Dans le cas des blocs, le lemme 3.4.6 montrera qu'il suffit d'un côté de vérifier que chaque élément du bloc

est compatible avec l'anti-unificateur des types obtenus par décomposition à l'aide de la relation Δ des types attendus et d'un autre côté de vérifier que tous les types attendus possèdent le même constructeur de types de tête.

Pour représenter ce mécanisme de décomposition parallèle, nous allons reformuler l'ensemble des règles de réécriture, en remplaçant les contrainte de type $w : \tau$ par des contraintes multiples $w : \{\tau_1 , \ldots , \tau_n\}$ regroupant un ensemble de contraintes de type pour la valeur w. Les nouvelles règles seront nommées [R'-...].

Implicitement, cette contrainte de types multiples sera valide ssi w est compatible avec l'anti-unificateur des types $\{\tau_1 , \ldots , \tau_n\}$. Ainsi, de la même manière que la règle [R-AntiUnif] permettait d'anti-unifier deux contraintes de type, il est possible de fusionner deux ensembles de contraintes de type et, dans certains cas, d'anti-unifier deux types d'un ensemble de contraintes de type.

$$\ell : \{\tau_{1..n}\} \wedge \ell : \{\tau_{n+1..m}\} \gg \ell : \{\tau_{1..m}\} \qquad [\text{R'-Join}]$$

$$\ell : \{\tau_1 \,;\, \tau_2 \,;\, \tau_{3..n}\} \gg \ell : \{\tau \,;\, \tau_{3..n}\} \quad si\ \forall \dot\theta.\, \dot\theta(\tau) = \dot\theta(\tau_1) \curlywedge \dot\theta(\tau_2) \qquad [\text{R'-Merge}]$$

Hypothèses de types Avec cette notion de contraintes de types multiples, il est nécessaire de pouvoir mémoriser plusieurs hypothèses de types dont il est pour l'instant impossible de calculer l'anti-unificateur principal. Ainsi, la construction $(\mu, \Phi').\, C$ contiendra toujours un environnement de typage généralisé. Lorsque les variables de type existentielles bloquant l'anti-unification de deux types auront été instanciées, il sera possible de simplifier un ensemble d'hypothèses de types en calculant son anti-unificateur principal. Il ne sera alors pas nécessaire de vérifier à nouveau la compatibilité de la valeur partagée avec le résultat de l'anti-unification.

$$(\mu, \Phi' \cup \{\ell : \{\tau_1 \,;\, \tau_2\}\}).\, C \gg (\mu, \Phi' \cup \{\ell : \{\tau\}\}).\, C \quad si\ \forall \dot\theta.\, \dot\theta(\tau) = \dot\theta(\tau_1) \curlywedge \dot\theta(\tau_2) \qquad [\text{R'-Simpl}]$$

Comme dans le système de réécriture sans contrainte de types multiples, il est possible de vérifier immédiatement la validité d'une contrainte de type qui est une instance d'une des hypothèses de types, quelle que soit l'instanciation future des variables existentielles.

$$(\mu, \Phi').\, (\ell : \{\tau_{1..n}\} \wedge C) \gg (\mu, \Phi').\, C \quad si\ \ell \in dom(\Phi')\ et\ \forall \dot\theta.\ \curlywedge\{\dot\theta(\Phi'(\ell))\} \preccurlyeq \curlywedge\{\dot\theta(\tau_{1..n})\} \qquad [\text{R'-Merge'}]$$

Fermetures Il reste maintenant à redéfinir les règles de réécriture permettant de vérifier la compatibilité d'un entier, d'un bloc ou d'une fermeture avec une contrainte de types multiples. Dans le cas des fermetures, on va d'abord vérifier que chacun des types attendus est une instance du type du code de la fermeture. Ces vérifications vont produire autant d'environnements de typage attendus pour l'environnement d'évaluation capturé. Le lemme 3.4.7 montrera qu'il suffit alors de vérifier que cet environnement d'évaluation est compatible avec l'anti-unificateur principal des environnements de typage obtenus pour avoir montré que la fermeture est compatible avec l'anti-unificateur principal des types attendus.

Ainsi, pour représenter ce comportement dans le système de réécriture, si le nombre de types attendus est m et si le type principal du code est $\forall\bar{\alpha}.\,(\Gamma \Rightarrow \tau' \rightarrow \tau'')$, où le cardinal de $\bar{\alpha}$ est p, on va introduire mp variables existentielles notées $\dot{\bar{\alpha}}_{1..m}$. Ces variables permettent dans un premier temps de générer m contraintes d'instanciation $\tau_i = \dot{\tau}'_i \rightarrow \dot{\tau}''_i$ où la notation $\dot{\tau}'_i$ représente alors le type τ' dans lequel les variables existentielles $\dot{\bar{\alpha}}_i$ ont été substituées aux variables universelles $\bar{\alpha}$. Dans un second temps, on va générer pour chaque valeur dans l'environnement d'évaluation de la fermeture une contrainte portant sur l'ensemble des m environnements de typage attendus $\rho(x) : \{\dot{\Gamma}_{1..m}(x)\}$, où let $\dot{\Gamma}_i$ représentent des environnements où les variables existentielles $\dot{\bar{\alpha}}_i$ ont été substituées aux variables universelles $\bar{\alpha}$ libres dans Γ. Si la vérification de ces contraintes réussit, on a alors vérifié que l'environnement d'évaluation de la fermeture est compatible avec l'anti-unificateur des environnements de typage attendus et que la fermeture est compatible avec l'anti-unificateur de tous les types attendus. Ce comportement est décrit par la règle de réécriture ci-dessous, où pour simplifier l'écriture on pose $\Phi'(\ell) = \emptyset$ lorsque $\ell \notin dom(\Phi')$.

$$(\mu, \Phi').\,(\ell : \{\tau_{1..n}\} \wedge C) \gg \qquad \text{[R'-CLOS]}$$
$$\exists \dot{\bar{\alpha}}_{1..m}.\begin{cases} \bigwedge_{i=1..m} \tau_i = \dot{\tau}'_i \rightarrow \dot{\tau}''_i \\ (\mu, \Phi' \cup \{\ell : \{\tau_{1..n}\}\}).\,(C \wedge \bigwedge_{x \in dom(\rho)} (\rho(x) : \{\dot{\Gamma}_{1..m}(x)\})) \end{cases}$$
$$\textit{si } \mu(\ell) = \langle \forall\bar{\alpha}.\,(\Gamma \Rightarrow \tau' \rightarrow \tau''), \rho\rangle \textit{ et } \Phi'(\ell) = \{\tau_{n+1..m}\}$$
$$\textit{et } \dot{\bar{\alpha}}_{1..m} \# \mathit{ftv}(\tau_{1..m}) \cup \mathit{ftv}(\Phi') \cup \mathit{ftv}(C)$$

Les règles de résolution de l'unification sont inchangées (cf. [R-UNIF] et [R-NUNIF]).

$$\exists\dot{\bar{\alpha}}.\,\tau_1 = \tau_2 \wedge C \gg \dot{\theta}(C) \quad \textit{si } \dot{\theta} \textit{ est le m.g.u de } \tau_1 \textit{ et } \tau_2 \textit{ et si } dom(\dot{\theta}) = \dot{\bar{\alpha}} \qquad \text{[R'-UNIF]}$$
$$\exists\dot{\bar{\alpha}}.\,\tau_1 = \tau_2 \wedge C \gg \texttt{False} \quad \textit{si } \tau_1 \textit{ et } \tau_2 \textit{ n'admettent pas d'unificateur} \qquad \text{[R'-NUNIF]}$$

Blocs et ensembles de types homogènes Dans le cas des blocs, la règle de réécriture va décomposer en parallèle l'ensemble des types attendus en des contraintes multiples sur les éléments du bloc : un ensemble de types $\{\tau_{1..n}\}$ sera décomposé en deux ensembles de types $\{\tau'_{1..n}\}$ et $\{\tau''_{1..n}\}$ tels que pour tout $i = 1..n$ on vérifie $[\tau'_i\,;\,\tau''_i] \nabla \tau_i$ et les contraintes $w' : \{\tau'_{1..n}\}$ et $w'' : \{\tau''_{1..n}\}$ seront à vérifier. Si leur vérification réussit, le système garantira que w' et w'' sont respectivement compatibles avec les anti-unificateurs des ensembles de types $\{\tau'_{1..n}\}$ et $\{\tau''_{1..n}\}$. Pour garantir la correction du système, il faut pouvoir en déduire que $\texttt{Blk}(w_1,\, w_2)$ est compatible avec l'anti-unificateur de $\{\tau_{1..n}\}$. Ce résultat n'est pas vrai dans la cas général, néanmoins, le lemme 3.4.6 montrera que cela est vrai lorsque l'ensemble des $\tau_{1..n}$ partagent le même constructeur de types. Lorsque les types d'un ensemble partageront le même constructeur de types de tête, on dira que cet ensemble est homogène et on notera $\Diamond\{\tau_{1..n}\}$. Ainsi, la décomposition en parallèle de contraintes de type portant sur un bloc peut être représentée par la règle de réécriture suivante, où comme précédemment pour simplifier l'écriture on pose $\Phi'(\ell) = \emptyset$ lorsque $\ell \notin dom(\Phi')$.

$$(\mu, \Phi').\,(\ell : \{\tau_{1..n}\} \wedge C) \gg \qquad \text{[R'-BLK-TRUE]}$$
$$(\mu, \Phi' \oplus \{\ell : \{\tau_{1..m}\}\}).\,(w_1 : \{\tau'_{1..m}\} \wedge w_2 : \{\tau''_{1..m}\} \wedge C)$$
$$\textit{si } \mu(\ell) = \texttt{Blk}(w_1,\, w_2) \textit{ et } \Phi'(\ell) = \{\tau_{n+1..m}\}$$
$$\textit{et } \Diamond\{\tau_{1..m}\} \textit{ et } \forall i = 1..m.\ \exists \tau'_i \tau''_i.\,[\tau'_i\,;\,\tau''_i] \nabla \tau_i$$

Exemple 3.4.4. *Reprenons l'exemple 3.4.3.*

$$\begin{array}{rl}\mu \equiv \{ & \ell_0 \mapsto \texttt{Blk}(\ell_1, \ell_2); \\ & \ell_1 \mapsto \langle\ \lambda().\, f\ x\ ;\ \{f \mapsto \ell_3\ ;\ x \mapsto 3\}\ \rangle; \\ & \ell_2 \mapsto \langle\ \lambda().\, f\ x\ ;\ \{f \mapsto \ell_3\ ;\ x \mapsto 0\}\ \rangle; \\ & \ell_3 \mapsto \langle\ \lambda y.\, (y :: z)\ ;\ \{z \mapsto 0\}\ \rangle \qquad\qquad \}\end{array}$$

Si le type attendu pour ℓ_0/μ est $\tau_{\texttt{Int}} \times \tau_{\texttt{Bool}}$ où $\tau_{\texttt{Int}} \equiv \texttt{Unit} \to \texttt{List(Int)}$ et $\tau_{\texttt{Bool}} \equiv \texttt{Unit} \to \texttt{List(Bool)}$), une suite possible de réécritures est :

$$\begin{array}{rcl}
 & & (\mu, \emptyset).\, (\ell_0 : \{(\tau_{\texttt{Int}} \times \tau_{\texttt{Bool}})\}) \\
\text{[R'-Blk-True]} & \gg & (\mu, \Phi_0).\, (\ell_1 : \{\tau_{\texttt{Int}}\} \wedge \ell_2 : \{\tau_{\texttt{Bool}}\}) \\
\text{[R'-Clos][R'-Unif]} & \gg & \exists \dot\alpha.\, (\mu, \Phi_1).\, \left\{ \begin{array}{l} 3 : \{\dot\alpha\} \wedge \ell_3 : \{\dot\alpha \to \texttt{List(Int)}\} \\ \ell_2 : \{\tau_{\texttt{Bool}}\} \end{array} \right. \\
\text{[R'-Clos][R'-Unif]} & \gg & \exists \dot\alpha \dot\beta.\, (\mu, \Phi_2).\, \left\{ \begin{array}{l} 3 : \{\dot\alpha\} \wedge \ell_3 : \{\dot\alpha \to \texttt{List(Int)}\} \\ 1 : \{\dot\beta\} \wedge \ell_3 : \{\dot\beta \to \texttt{List(Bool)}\} \end{array} \right. \\
\text{[R'-Join][R'-Clos]} & \gg & \exists \dot\alpha \dot\beta \dot\gamma_1 \dot\gamma_2.\, \left\{ \begin{array}{l} \dot\alpha \to \texttt{List(Int)} = \dot\gamma_1 \to \texttt{List}(\dot\gamma_1) \\ \dot\beta \to \texttt{List(Bool)} = \dot\gamma_2 \to \texttt{List}(\dot\gamma_2) \\ (\mu, \Phi_3').\, \left\{ \begin{array}{l} 3 : \{\dot\alpha\} \wedge 1 : \{\dot\beta\} \\ 0 : \{\texttt{List}(\dot\gamma_1); \texttt{List}(\dot\gamma_2)\} \end{array} \right. \end{array} \right. \\
\text{[R'-Unif]} & \gg & (\mu, \Phi_3).\, \left\{ \begin{array}{l} 3 : \{\texttt{Int}\} \wedge 1 : \{\texttt{Bool}\} \\ 0 : \{\texttt{List(Int)}; \texttt{List(Bool)}\} \end{array} \right. \\
\text{[R'-Merge]} & \gg & (\mu, \Phi_3).\, (3 : \{\texttt{Int}\} \wedge 1 : \{\texttt{Bool}\} \wedge 0 : \{\texttt{List}(\alpha)\}) \\
 & \gg^* & \texttt{True}
\end{array}$$

Décomposition explicite Si l'ensemble des types attendus pour un bloc n'est pas homogène, ou si le constructeur de types de tête commun à tous les types attendus n'est pas décomposable par la relation $\triangledown$, alors la vérification échoue. Si d'un autre côté, l'ensemble des types attendus contient des variables existentielles, le bloc ne peut pas être décomposé pour l'instant.

$$\begin{array}{rl}(\mu, \Phi').\, (\ell : \{\tau_{1..n}\} \wedge C) \gg \texttt{False} & \text{[R'-Blk-False]} \\ \quad si\ \mu(\ell) = \texttt{Blk}(w_1, w_2)\ et\ \Phi'(\ell) = \{\tau_{n+1..m}\} & \\ \quad et\ \neg(\Diamond\{\tau_{1..m}\})\ ou\ \nexists \tau_1' \tau_1''.\, [\tau_1'\ ;\ \tau_1''] \triangledown \tau_1 & \end{array}$$

Néanmoins, si l'ensemble des types attendus est constitué d'un ensemble de variables existentielles et d'un ensemble homogène de types, il sera possible d'instancier partiellement les variables existentielles avec le même constructeur de types que les autres types attendus, permettant ainsi de commencer la vérification du bloc. Pour simplifier l'expression des règles de réécriture décrivant le principe ci-dessus, la syntaxe des contraintes est étendue avec des constructions représentant explicitement les relations $\Diamond\{\tau_{1..n}\}$ et $[\tau'\ ;\ \tau''] \triangledown \tau$. Ces constructions supplémentaires seront aussi nécessaire à la section 3.7 pour exprimer une stratégie de réécriture basée sur un tri topologique du tas.

$$\begin{array}{rll} C & ::= & \ldots \\ & \mid [\tau\ ;\ \tau] \triangledown \tau & \textit{propagateur de types} \\ & \mid \Diamond\{\tau\ ;\ \ldots\} & \textit{prédicat d'homogénéité} \end{array}$$

Avec ces nouvelles constructions syntaxiques, les règles de décomposition d'un bloc [R'-Blk-True] et [R'-Blk-False] peuvent alors être remplacées par la règle générale ci-dessous, où encore une fois on pose $\Phi'(\ell) = \emptyset$ lorsque $\ell \notin dom(\Phi')$.

$$(\mu, \Phi').(\ell : \{\tau_{1..n}\} \wedge C) \gg \exists \dot\alpha_{1..m}.\, \dot\beta_{1..m}.\, \begin{cases} \Diamond\{\tau_{1..m}\} \\ \bigwedge_{i=1..m} [\dot\alpha_i\,;\,\dot\beta_i] \bigtriangledown \tau_i \\ (\mu, \Phi' \cup \{\ell : \{\tau_{1..n}\}\}).(w' : \{\dot\alpha_{1..m}\} \wedge w'' : \{\dot\beta_{1..m}\} \wedge C) \\ \quad si\ \mu(\ell) = \mathtt{Blk}(w', w'')\ et\ \Phi'(\ell) = \{\tau_{n+1..m}\} \\ \quad et\ \{\dot\alpha_{1..m}\,;\,\dot\beta_{1..m}\} \# fv(\{\tau_{1..m}\}) \cup fv(\Phi') \cup fv(C) \end{cases} \quad \text{[R'-Blk]}$$

La relation $\bigtriangledown$ étant définie par cas sur le constructeur de types, la résolution d'une contrainte $[\tau'\,;\,\tau''] \bigtriangledown \mathtt{T}(\tau, \dots)$ sera immédiate ; une contrainte $[\tau'\,;\,\tau''] \bigtriangledown \alpha$ sera trivialement fausse ; et la résolution d'une contrainte $[\tau'\,;\,\tau''] \bigtriangledown \dot\alpha$ sera impossible avant l'instanciation de la variable existentielle.

$$[\tau'\,;\,\tau''] \bigtriangledown \tau \gg \tau' = \tau_1 \wedge \tau'' = \tau_2 \quad si\ \exists \tau_1 \tau_2.\, [\tau_1\,;\,\tau_2] \bigtriangledown \tau \quad \text{[R'-}\bigtriangledown\text{-True]}$$

$$[\tau'\,;\,\tau''] \bigtriangledown \tau \gg \mathtt{False} \quad si\ \tau \not\equiv \dot\alpha\ et\ \nexists \tau_1 \tau_2.\, [\tau_1\,;\,\tau_2] \bigtriangledown \tau \quad \text{[R'-}\bigtriangledown\text{-False]}$$

De la même manière, la résolution d'une contrainte $\Diamond\{\tau_{1..n}\}$ où tous les types partagent le même constructeur sera immédiate ; une contrainte $\Diamond\{\tau_{1..n}\}$ où deux types ne partagent pas le même constructeur de types est trivialement fausse ; une contrainte $\Diamond\{\dot\alpha_{1..n}\}$ constituée entièrement de variables existentielles ne peut pas être résolue avant l'instanciation d'au moins une de ces variables existentielles ; il sera alors possible de propager le constructeur de types à toutes les autres variables existentielles. Tous ces comportements peuvent être décris par la règle de réécriture ci-dessous.

$$\Diamond\{\mathtt{T}(\tau_1, \dots, \tau_n)\,;\,\tau'_{1..m}\} \gg \exists \dot\alpha_{(1,1)..(n,m)}.\, \bigwedge_{i=1..m} (\tau'_i = \mathtt{T}(\dot\alpha_{(1,i)}, \dots, \dot\alpha_{(n,i)})) \quad si\ \dot\alpha_{(1,1)..(n,m)} \# fv(\{\tau'_{1..m}\}) \quad \text{[R'-}\Diamond\text{]}$$

En pratique, lors de l'application de la règle [R'-Blk] lorsque les types attendus ne sont pas uniquement des variables existentielles, la contraintes d'homogénéité est immédiatement résolue. Il en va de même pour les contraintes d'unification introduites ensuite par la règle [R'-$\Diamond$]. Si ces unifications réussissent, les contraintes de propagation de type introduites par la règle [R'-Blk] pourront toutes être résolues et les variables $\dot\alpha_{1..m}$ et $\dot\beta_{1..m}$ seront immédiatement instanciées. Ainsi, dans le prototype décrit au chapitre 4, les variables existentielles introduites par la règle [R'-Blk] seront explicitées uniquement lorsque l'ensemble des types attendus ne contiendra que des variables existentielles.

Entiers Pour compléter le système de réécriture il ne reste plus qu'à adapter les règles de réécriture s'appliquant sur les entiers à la construction $i : \{\tau_{1..n}\}$ selon le principe suivant : la relation $\bigtriangleup$ étant définie par cas sur le constructeur de types, un entier est compatible avec l'anti-unificateur d'un ensemble de types ssi il est compatible avec l'un des types et si l'ensemble de types est homogène.

$$i : \{\tau_{1..n}\} \gg \Diamond\{\tau_{1..n}\} \wedge i : \{\tau_1\} \quad si\ n \geq 2 \quad \text{[R'-Int]}$$

$$i : \{\tau\} \gg \mathtt{True} \quad si\ i \bigtriangleup \tau \quad \text{[R'-Int-True]}$$

$$i : \{\tau\} \gg \mathtt{False} \quad si\ \tau \not\equiv \dot\alpha\ et\ \neg(i \bigtriangleup \tau) \quad \text{[R'-Int-False]}$$

Formes normales Avec l'ensemble de règles de réécriture défini jusqu'à maintenant, toute contrainte de type sur un entier $i : \tau$ telle que $\tau \not\equiv \dot{\alpha}$ peut être réécrite par une des règles de réécriture [R'-Int-. . .] ; toute contrainte de type sur une adresse-mémoire peut être réécrite inconditionnellement par l'une des règles [R'-Blk] et [R'-Clos] ; toutes les variables existentielles introduites sont explicitement quantifiées existentiellement et les contraintes d'égalité peuvent être systématiquement réécritent par l'une des règles [R'-Unif] et [R'-NUnif] ; toute contrainte $[\tau_1 ; \tau_2] \triangledown \tau$ tel que $\tau \not\equiv \dot{\alpha}$ se réécrit par l'une des règles [R'-$\triangledown$-. . .] ; toute contrainte $\Diamond\{\tau_{1..n}\}$ ne portant pas uniquement sur des variables existentielles peut se réécrire par la règle [R'-$\Diamond$] ; toutes les conjonctions inutiles peuvent être réécrites par les règles [R'-False] et [R'-True], et le tas peut finalement disparaître à l'aide de la règle [R'-Heap]. Ainsi, si la contrainte initiale est close — du point de vue des adresses-mémoire et des variables existentielles — les formes normales sont alors `True`, `False` et des contraintes close de la forme :

$$\exists \bar{\dot{\alpha}}_i \bar{\dot{\beta}} \bar{\dot{\gamma}}.\ \overline{\Diamond\{\dot{\alpha}_{1..n}\}} \wedge \overline{[\tau' ; \tau''] \triangledown \dot{\beta}} \wedge \overline{i : \{\dot{\gamma}\}}$$

De la même manière que la règle [R-Univ] permettait d'ignorer les valeurs du tas sur lesquelles ne porte aucune contrainte de type, ces formes normales non triviales peuvent être résolues en instanciant les variables existentielles restantes à l'aide du type universel. Ainsi, en ajoutant la règle de réécriture ci-dessous, les formes normales sont uniquement `True` et `False`.

$$\exists \bar{\dot{\alpha}}_i \bar{\dot{\beta}} \bar{\dot{\gamma}}.\ \overline{\Diamond\{\dot{\alpha}_{1..n}\}} \wedge \overline{[\tau' ; \tau''] \triangledown \dot{\beta}} \wedge \overline{i : \{\dot{\gamma}\}} \gg \texttt{True} \qquad \text{[R'-Univ]}$$

Terminaison Si la restriction du système de réécriture au système de types original permet la vérification de certain tas issues de programme OCaml avec récursion polymorphe dont la vérification ne terminait pas dans le système de réécriture basé sur le système de type généralisé (voir l'exemple 3.4.2), l'exemple ci-dessous montre qu'il existe toujours des suites infinies de réécritures.

Exemple 3.4.5. *Le programme*[8] *suivant :*

$$\begin{array}{l}\texttt{let}\ delay = \lambda f.\lambda x.\lambda y.\ f\ x\ \texttt{in}\\ \texttt{let rec}\ f = delay\ f\ ()\ \texttt{in}\\ f\end{array}$$

peut s'évaluer, dans le langage de valeur w, vers un couple ℓ_0/μ où :

$$\mu \equiv \{\ell_0 \mapsto \langle \lambda().f\ x, \{f : \ell_0 ; x : 0\}\rangle\}$$

8. Tel quel ce programme n'est pas accepté par le compilateur OCaml. Néanmoins, il reste possible d'écrire un programme similaire à l'aide de paresse ou de module récursif. Par exemple :

$\texttt{let}\ ldelay = \lambda f.\lambda x.\lambda y.\ (Lazy.force\ f)\ x\ \texttt{in}$ $\texttt{let rec}\ f = \texttt{lazy}\ (ldelay\ f\ ())\ \texttt{in}$ $(Lazy.force\ f)$	$\texttt{let}\ delay = \lambda f.\lambda x.\lambda y.\ f\ x$ $\texttt{module rec}\ A\ :\ \texttt{sig val}\ f : \texttt{unit} \to \alpha\ \texttt{end} =$ $\texttt{struct let}\ f = delay\ A.f\ ()\ \texttt{end}$

Par ailleurs, si l'évaluation d'une application $f\ ()$ ne termine pas, il existe aussi des fermetures dont l'application termine mais dont la vérification ne termine pas.

et où le type associé à $\lambda y.\, f\ x$ *est* $\forall\alpha\beta\gamma.\,(\{f : \alpha \to \beta\,;\, x : \alpha\} \Rightarrow \gamma \to \beta)$*. Ce programme et la valeur* $\mu(\ell_0)$ *sont chacun compatibles avec le type* $\texttt{Unit} \to \texttt{Int}$*, mais la vérification de compatibilité du tas* μ *avec ce type produit une suite infinie de réécritures :*

$$
\begin{array}{lcl}
 & & (\mu, \emptyset).\,(\ell_0 : \{\texttt{Unit} \to \texttt{Int}\}) \\
\text{[R'-Clos][R'-Unif]} & \gg & \exists\dot\alpha.\,(\mu, \{\ell_0 : \{\texttt{Unit} \to \texttt{Int}\}\}).\,(\ell_0 : \{\dot\alpha \to \texttt{Int}\} \wedge 0 : \{\dot\alpha\}) \\
\text{[R'-Clos][R'-Unif]} & \gg & \exists\dot\alpha\dot\beta\dot\gamma.\,(\mu, \{\ell_0 : \{\texttt{Unit} \to \texttt{Int}\,;\, \dot\alpha \to \texttt{Int}\}). \\
 & & \qquad (\ell_0 : \{\dot\beta \to \texttt{Int}; \dot\gamma \to \texttt{Int}\} \wedge 0 : \{\dot\alpha\} \wedge 0 : \{\dot\beta\,;\, \dot\gamma\}) \\
 & \gg & \ldots
\end{array}
$$

La section 3.7 étudiera les compromis possibles pour obtenir un algorithme utilisable en pratique : c'est-à-dire un algorithme terminant et « raisonnablement » complet vis-à-vis du système de types. En attendant, on peut remarquer informellement que l'argument de terminaison utilisé en l'absence de fermeture peut s'étendre en présence de certaines fermetures : celles dont toutes les variables de types libres de l'environnement de typage associé au pointeur de code sont aussi libres dans le type associé du pointeur de code — par exemple une fermeture dont le type associé est $\forall\alpha.\,\{x : \alpha\} \Rightarrow \texttt{Unit}x \to \alpha$ mais pas celle dont le type associé est $\forall\alpha\beta.\,\{f : \beta \to \alpha\,;\, x : \beta\} \Rightarrow \texttt{Unit} \to \alpha$ car β n'apparaît pas dans $\texttt{Unit} \to \alpha$.

En effet, la vérification de telles fermetures ne nécessite pas l'utilisation d'inconnue de type. Dans le système de réécriture, toutes les variables de type introduites par la règle [R'-Clos] peuvent alors être immédiatement instanciées par la contrainte d'unification introduite conjointement. L'explicitation de variables existentielles est alors inutile en pratique : pour toutes les autres règles de réécriture, si la contrainte de gauche ne contient pas de variable de type existentielle, les variables introduites par la règle peuvent être immédiatement instanciées par les contrainte d'unification générées conjointement, ou, pour la règle [R'-Blk], par les contraintes de propagation de type.

En l'absence d'inconnue de type, il est systématiquement possible d'appliquer les règles [R'-Merge] et [R'-Simpl] pour réduire respectivement les contraintes de types multiples et les ensembles d'hypothèses à des singletons. Dans ce cas, la condition d'application de la règle [R'-Merge'] devient systématiquement décidable et, de la même manière que pour la règle [R-Blk-True"'] dans le cas sans fermeture, il est alors possible d'interdire l'application de la règle [R'-Blk], lorsque la règle [R'-Merge'] est applicable. Cette interdiction permet d'utiliser le même argument de décroissance du nombre de constructeurs de types que dans le cas sans fermeture à la section 3.4.1.

3.4.3 Correction

Les sections 3.4.1 et 3.4.2 ont présenté un système de réécriture reprenant les idées décrites à la section 3.3, mais permettant de restreindre les valeurs acceptées à celles qui sont typables dans le système de type original. Cette restriction a été réalisée en manipulant explicitement l'ensemble des types attendus par une valeur partagée dans la syntaxe des contraintes. Cette section va maintenant détailler la preuve de correction de ce système de réécriture, en commençant par préciser la notion de satisfaisabilité utilisée pour cela. La syntaxe des contraintes manipulées dans ce système de réécriture est :

$$
\begin{array}{rll}
C & ::= & \texttt{True} \mid \texttt{False} \mid C \wedge C \\
& \mid & (\mu, \Phi').\, C & \textit{fragment du tas} \\
& \mid & w : \{\tau\,;\ldots\} & \textit{contrainte de type} \\
& \mid & \exists\dot{\alpha}.\, C & \textit{quantification existentielle} \\
& \mid & \tau = \tau & \textit{unification} \\
& \mid & [\tau\,;\tau] \bigtriangledown \tau & \textit{propagateur de types} \\
& \mid & \Diamond\{\tau\,;\ldots\} & \textit{prédicat d'homogénéité}
\end{array}
$$

L'ensemble des règles de réécriture nécessaire pour résoudre ces contraintes est regroupé dans la figure 3.2. Dans ce système le contexte de satisfaisabilité d'une contrainte est un environnement de typage simple Ψ et une substitution $\dot{\theta}$ dont le domaine ne contient que des variables de type existentielles mais dont l'image n'en contient pas. Ainsi, la notion de satisfaisabilité d'une contrainte est définie comme une relation entre un environnement de typage Ψ, une substitution $\dot{\theta}$ et une contrainte C, que l'on note $\dot{\theta}, \Psi \models C$. En notant $\curlywedge\{\dot{\theta}(\Phi')\}$ un environnement de typage simple tel que :

$$(\curlywedge\{\dot{\theta}(\Phi')\})(\ell) = \curlywedge\{\dot{\theta}(\tau) \mid \tau \in \Phi'(\ell)\}$$

et $\curlywedge\{\dot{\theta}(\Phi')\} \curlywedge \Psi''$ est un environnement de typage simple tel que :

$$(\curlywedge\{\dot{\theta}(\Phi')\} \curlywedge \Psi'')(\ell) = \begin{cases} (\curlywedge\{\dot{\theta}(\Phi')\})(\ell) \curlywedge \Psi''(\ell) & si\ \ell \in dom(\Phi') \cap dom(\Psi'') \\ (\curlywedge\{\dot{\theta}(\Phi')\})(\ell) & si\ \ell \notin dom(\Psi'') \\ \Psi''(\ell) & si\ \ell \notin dom(\Phi') \end{cases}$$

on peut définir inductivement la relation $\dot{\theta}, \Psi \models C$ par l'ensemble de règles suivant :

$$\text{[C'-True]}\quad \dot{\theta}, \Psi \models \texttt{True}$$

$$\text{[C'-And]}\quad \frac{\dot{\theta}, \Psi \models C_1 \qquad \dot{\theta}, \Psi \models C_2}{\dot{\theta}, \Psi \models C_1 \wedge C_2}$$

$$\text{[C'-OrigQuestion]}\quad \frac{\Psi \vdash w : \curlywedge\{\dot{\theta}(\tau_{1..n})\}}{\dot{\theta}, \Psi \models w : \{\tau_{1..n}\}}$$

$$\text{[C'-Equal]}\quad \frac{\dot{\theta}(\tau_1) \equiv \dot{\theta}(\tau_2)}{\dot{\theta}, \Psi \models \tau_1 = \tau_2}$$

$$\text{[C'-Exists]}\quad \frac{\dot{\theta} \oplus \{\dot{\alpha} \mapsto \tau\}, \Psi \models C}{\dot{\theta}, \Psi \models \exists\dot{\alpha}.\, C}$$

$$\text{[C'-}\bigtriangledown\text{]}\quad \frac{[\dot{\theta}(\tau')\,;\,\dot{\theta}(\tau'')] \bigtriangledown \dot{\theta}(\tau)}{\dot{\theta}, \Psi \models [\tau'\,;\,\tau''] \bigtriangledown \tau}$$

$$\text{[C'-}\Diamond\text{]}\quad \frac{\dot{\theta}(\tau_1) \equiv \mathtt{T}(\tau_1^1, \ldots, \tau_1^m) \qquad \ldots \qquad \dot{\theta}(\tau_n) \equiv \mathtt{T}(\tau_n^1, \ldots, \tau_n^m)}{\dot{\theta}, \Psi \models \Diamond\{\tau_{1..n}\}}$$

$$\text{[C'-OrigHeap]}\quad \frac{dom(\curlywedge\{\dot{\theta}(\Phi')\} \curlywedge \Psi'') = dom(\mu) \qquad \Psi \oplus (\curlywedge\{\dot{\theta}(\Phi')\} \curlywedge \Psi'') \vdash \mu : \Psi'' \qquad \dot{\theta}, \Psi \oplus (\curlywedge\{\dot{\theta}(\Psi')\} \curlywedge \Psi'') \models C}{\dot{\theta}, \Psi \models (\mu, \Phi').\, C}$$

Autrement dit, par rapport à la notion de satisfaisabilité définie à la section 3.3.3 :

- une contrainte de propagation de types $[\tau_1;\tau_2] \bigtriangledown \tau$ est satisfaite par une substitution $\dot{\theta}$ si, après application de la substitution, on vérifie $[\dot{\theta}(\tau_1);\dot{\theta}(\tau_2)] \bigtriangledown \dot{\theta}(\tau)$;

FIGURE 3.2 Règles de réécriture pour le système de types original

Entiers

$$i : \{\tau_{1..n}\} \gg \Diamond\{\tau_{1..n}\} \wedge i : \{\tau_1\} \quad si\ n \geq 2 \qquad [\text{R'-INT}]$$

$$i : \{\tau\} \gg \texttt{True} \quad si\ i \vartriangle \tau \qquad [\text{R'-INT-TRUE}]$$

$$i : \{\tau\} \gg \texttt{False} \quad si\ \tau \not\equiv \dot{\alpha}\ et\ \neg(i \vartriangle \tau) \qquad [\text{R'-INT-FALSE}]$$

Étiquettes

$$(\mu, \Phi').\,(\ell : \{\tau_{1..n}\} \wedge C) \gg (\mu, \Phi').\,C \quad si\ \ell \in dom(\Phi')\ et\ \forall\dot{\theta}.\ \curlywedge\{\dot{\theta}(\Phi'(\ell))\} \preccurlyeq \curlywedge\{\dot{\theta}(\tau_{1..n})\} \qquad [\text{R'-MERGE'}]$$

$$(\mu, \Phi').\,(\ell : \{\tau_{1..n}\} \wedge C) \gg \exists\dot{\alpha}_{1..m}.\,\dot{\beta}_{1..m}.\ \begin{cases} \Diamond\{\tau_{1..m}\} \\ \bigwedge_{i=1..m}[\dot{\alpha}_i\,;\,\dot{\beta}_i] \bigtriangledown \tau_i \\ (\mu, \Phi' \cup \{\ell : \{\tau_{1..n}\}\}).\,(w' : \{\dot{\alpha}_{1..m}\} \wedge w'' : \{\dot{\beta}_{1..m}\} \wedge C) \end{cases}$$
$$si\ \mu(\ell) = \texttt{Blk}(w', w'')\ et\ \Phi'(\ell) = \{\tau_{n+1..m}\}$$
$$et\ \{\dot{\alpha}_{1..m}\,;\,\dot{\beta}_{1..m}\} \#\ fv(\{\tau_{1..m}\}) \cup fv(\Phi') \cup fv(C) \qquad [\text{R'-BLK}]$$

$$(\mu, \Phi').\,(\ell : \{\tau_{1..n}\} \wedge C) \gg \exists\bar{\dot{\alpha}}_{1..m}.\ \begin{cases} \bigwedge_{i=1..m} \tau_i = \dot{\tau}'_i \to \dot{\tau}''_i \\ (\mu, \Phi' \cup \{\ell : \{\tau_{1..n}\}\}).\,(C \wedge \bigwedge_{x \in dom(\rho)} (\rho(x) : \{\dot{\Gamma}_{1..m}(x)\})) \end{cases}$$
$$si\ \mu(\ell) = \langle \forall\bar{\alpha}.\,(\Gamma \Rightarrow \tau' \to \tau''), \rho\rangle\ et\ \Phi'(\ell) = \{\tau_{n+1..m}\}$$
$$et\ \bar{\dot{\alpha}}_{1..m} \#\ ftv(\tau_{1..m}) \cup ftv(\Phi') \cup ftv(C) \qquad [\text{R'-CLOS}]$$

Décomposition et homogénéité

$$[\tau'\,;\,\tau''] \bigtriangledown \tau \gg \tau' = \tau_1 \wedge \tau'' = \tau_2 \quad si\ \exists\tau_1\tau_2.\,[\tau_1\,;\,\tau_2] \bigtriangledown \tau \qquad [\text{R'-}\bigtriangledown\text{-TRUE}]$$

$$[\tau'\,;\,\tau''] \bigtriangledown \tau \gg \texttt{False} \quad si\ \tau \not\equiv \dot{\alpha}\ et\ \nexists\tau_1\tau_2.\,[\tau_1\,;\,\tau_2] \bigtriangledown \tau \qquad [\text{R'-}\bigtriangledown\text{-FALSE}]$$

$$\Diamond\{\mathtt{T}(\tau_1, \ldots, \tau_n)\,;\,\tau'_{1..m}\} \gg \exists\dot{\alpha}_{(1,1)..(n,m)}.\ \bigwedge_{i=1..m}(\tau'_i = \mathtt{T}(\dot{\alpha}_{(1,i)}, \ldots, \dot{\alpha}_{(n,i)})) \quad si\ \dot{\alpha}_{(1,1)..(n,m)} \#\ fv(\{\tau'_{1..m}\}) \qquad [\text{R'-}\Diamond]$$

Anti-unification

$$\ell : \{\tau_{1..n}\} \wedge \ell : \{\tau_{n+1..m}\} \gg \ell : \{\tau_{1..m}\} \qquad [\text{R'-JOIN}]$$

$$\ell : \{\tau_1\,;\,\tau_2\,;\,\tau_{3..n}\} \gg \ell : \{\tau\,;\,\tau_{3..n}\} \quad si\ \forall\dot{\theta}.\,\dot{\theta}(\tau) = \dot{\theta}(\tau_1) \curlywedge \dot{\theta}(\tau_2) \qquad [\text{R'-MERGE}]$$

$$(\mu, \Phi' \cup \{\ell : \{\tau_1\,;\,\tau_2\}\}).\,C \gg (\mu, \Phi' \cup \{\ell : \{\tau\}\}).\,C \quad si\ \forall\dot{\theta}.\,\dot{\theta}(\tau) = \dot{\theta}(\tau_1) \curlywedge \dot{\theta}(\tau_2) \qquad [\text{R'-SIMPL}]$$

Unification

$$\exists\bar{\dot{\alpha}}.\,\tau_1 = \tau_2 \wedge C \gg \dot{\theta}(C) \quad si\ \dot{\theta}\ est\ le\ m.g.u\ de\ \tau_1\ et\ \tau_2\ et\ si\ dom(\dot{\theta}) = \bar{\dot{\alpha}} \qquad [\text{R'-UNIF}]$$

$$\exists\bar{\dot{\alpha}}.\,\tau_1 = \tau_2 \wedge C \gg \texttt{False} \quad si\ \tau_1\ et\ \tau_2\ n'admettent\ pas\ d'unificateur \qquad [\text{R'-NUNIF}]$$

Paramétricité

$$\exists\bar{\dot{\alpha}}_i\bar{\dot{\beta}}\bar{\dot{\gamma}}.\ \overline{\Diamond\{\dot{\alpha}_{1..n}\}} \wedge \overline{[\tau'\,;\,\tau''] \bigtriangledown \dot{\beta}} \wedge \overline{i : \{\dot{\gamma}\}} \gg \texttt{True} \qquad [\text{R'-UNIV}]$$

Autres

$$C \wedge \texttt{False} \gg \texttt{False} \qquad [\text{R'-FALSE}]$$

$$C \wedge \texttt{True} \gg C \qquad [\text{R'-TRUE}]$$

$$(\mu, \Phi).\,C \gg C \quad si\ fl(C) \#\ dom(\mu) \qquad [\text{R'-HEAP}]$$

- une contrainte d'homogénéité $\Diamond\{\tau_{1..n}\}$ est satisfaite par une substitution $\dot{\theta}$ si, après application de la substitution, on vérifie que les types $\dot{\theta}(\tau_1)$ à $\dot{\theta}(\tau_n)$ partagent le même constructeur de types de tête ;
- une contrainte de type $w : \{\tau_{1..n}\}$ est satisfaite par un environnement Ψ et une substitution $\dot{\theta}$ si w est compatible sous les hypothèses Ψ avec l'anti-unificateur des types $\dot{\theta}(\tau_1)$ à $\dot{\theta}(\tau_n)$;
- et une contrainte $(\mu, \Phi').\, C$ est satisfaite par un environnement Ψ et une substitution $\dot{\theta}$ s'il existe un environnement Ψ'' compatible avec μ sous les hypothèses $\Psi \oplus (\curlywedge\{\dot{\theta}(\Phi')\} \curlywedge \Psi'')$ et tel que ces mêmes hypothèses permettent de satisfaire la contrainte C. Comme précédemment, pour faciliter la preuve de correction de la règle de réécriture [R'-Heap], la satisfaisabilité d'une contrainte $(\mu, \Phi').\, C$ n'impose pas la compatibilité de μ avec les hypothèses de types mémorisées dans $\dot{\theta}(\Phi')$.

Lemme 3.4.3. *Soit un tas μ, une valeur w et un type τ. On a $\emptyset, \emptyset \models (\mu, \emptyset).\, (w : \tau)$ ssi il existe un environnement de typage Ψ'' tel que $\mu : \Psi''$ et $\Psi'' \vdash w : \tau$.*

Preuve Trivial avec les règles [C'-OrigHeap] et [C'-OrigQuestion]. □

Lemme 3.4.4. *Pour toute contrainte C, on vérifie $\dot{\theta}, \Psi \models C$ ssi $\emptyset, \Psi \models \dot{\theta}(C)$.*

Preuve Similaire à la preuve du lemme 3.3.4. □

Corollaire 3.4.5. *Soit une contrainte C, une substitution $\dot{\theta}$ et un environnement de typage Ψ. Si $\emptyset, \Psi \models \dot{\theta}(C)$ alors $\emptyset, \Psi \models \exists\bar{\alpha}.\, C$ où $\bar{\alpha} = dom(\dot{\theta})$.*

Lemme 3.4.6. *Soit trois ensembles de types $\tau_{1..n}$, $\tau'_{1..n}$ et $\tau''_{1..n}$ tels que pour $i = 1..n$, $[\tau'_i; \tau''_i] \bigtriangledown \tau_i$. Si $\Diamond\{\tau_{1..n}\}$ alors*

$$[\curlywedge\{\tau'_{1..n}\}; \curlywedge\{\tau''_{1..n}\}] \bigtriangledown \curlywedge\{\tau'_{1..n}\}$$

Preuve La relation $\bigtriangledown$ étant définie par cas sur le constructeur de types, si $\Diamond\{\tau_{1..n}\}$ alors chaque relation $[\tau'_i\,;\, \tau''_i] \bigtriangledown \tau_i$ est issue du même cas de la définition de $\bigtriangledown$. Autrement dit, il existe un type $\tau = \mathtt{T}(\alpha, \ldots, \delta)$ tel que $[\tau'\,;\, \tau''] \bigtriangledown \tau$ et tel pour chaque τ_i, il existe une substitution θ_i telle que $\tau_i = \theta_i(\tau)$, $\tau'_i = \theta_i(\tau')$ et $\tau''_i = \theta_i(\tau'')$. Par définition de l'anti-unification, en posant $\theta(\alpha) = \curlywedge\{\theta_{1..n}(\alpha)\}$, alors on vérifie $\theta(\tau) = \curlywedge\{\theta_{1..n}(\tau)\} = \curlywedge\{\tau_{1..n}\}$ et $\theta(\tau') = \curlywedge\{\tau'_{1..n}\}$ et $\theta(\tau'') = \curlywedge\{\tau''_{1..n}\}$. Autrement dit, $[\curlywedge\{\tau'_{1..n}\}; \curlywedge\{\tau''_{1..n}\}] \bigtriangledown \curlywedge\{\tau'_{1..n}\}$. □

Lemme 3.4.7. *Soit une fermeture $\mu(\ell) = \langle \forall\bar{\alpha}.\, (\Gamma \Rightarrow \tau' \to \tau''), \rho\rangle$, deux instances de son schéma de type $\Gamma_1 \Rightarrow \tau'_1 \to \tau''_1$ et $\Gamma_2 \Rightarrow \tau'_2 \to \tau''_2$ et un environnement de typage simple Ψ. Si $\Psi \vdash \rho : \Gamma_1 \curlywedge \Gamma_2$ alors $\Psi \vdash \mu(\ell) : (\tau'_1 \to \tau''_1) \curlywedge (\tau'_2 \to \tau''_2)$.*

Preuve En posant $\Gamma_0 = \Gamma_1 \curlywedge \Gamma_2$ et $\tau'_0 \to \tau''_0 = (\tau'_1 \to \tau''_1) \curlywedge (\tau'_2 \to \tau''_2)$. Par principalité de l'anti-unification (lemme 3.4.1), on vérifie $\Gamma \Rightarrow \tau' \to \tau'' \preccurlyeq \Gamma_0 \Rightarrow \tau'_0 \to \tau''_0$. On en déduit par [H-Clos] et le lemme d'instanciation (lemme 2.3.1) que si $\Psi \vdash \rho : \Gamma_0$, alors $\Psi \vdash \mu(\ell) : \tau'_0 \to \tau''_0$.

Lemme 3.4.8. *Pour chacune des règles de réécriture $C_1 \gg C_2$ de la figure 3.2, on vérifie pour tout environnement Ψ et toute substitution $\dot\theta$, si $\dot\theta, \Psi \models C_2$ alors $\dot\theta, \Psi \models C_1$.*

Preuve Les preuves de correction des règles de réécriture sont détaillées dans l'ordre de la figure 3.2. Les preuves identiques à celles de la section 3.3.3 ne sont pas détaillées.

– Entiers

Cas [R'-INT]. Vérifions que pour tout $\dot\theta$ et Ψ, si $\dot\theta, \Psi \models \Diamond\{\tau_{1..n}\} \wedge i : \{\tau_1\}$ alors $\dot\theta, \Psi \models i : \{\tau_{1..n}\}$. Par définition de $\dot\theta, \Psi \models \Diamond\{\tau_{1..n}\}$ à l'aide de [C'-$\Diamond$], l'ensemble $\{\dot\theta(\tau_{1..n})\}$ est homogène. Ainsi, les types $\dot\theta(\tau_{1..n})$ partagent un même constructeur de types de tête `T` et leur anti-unificateur est de la forme $\mathtt{T}(\ldots)$. La relation $\vartriangle$ étant définie par cas sur le constructeur de types, l'hypothèse $i \vartriangle \dot\theta(\tau_1)$ permet alors de conclure $i \vartriangle \curlywedge\{\dot\theta(\tau_{1..n})\}$, c'est-à-dire $\dot\theta, \Psi \models i : \{\tau_{1..n}\}$,

– Étiquettes

Cas [R'-MERGE']. Vérifions, lorsque $\forall\dot\theta.\ \curlywedge\{\dot\theta(\Phi'(\ell))\} \preccurlyeq \curlywedge\{\dot\theta(\tau_{1..n})\}$, que pour tout $\dot\theta$ et Ψ, si $\dot\theta, \Psi \models (\mu, \Phi').\, C$ alors $\dot\theta, \Psi \models (\mu, \Phi').\, (\ell : \{\tau_{1..n}\} \wedge C)$. La condition d'application permet à l'aide des règles [C'-ORIGQUESTION] et [W-LABEL], de vérifier systématiquement $\dot\theta, \Psi \models (\mu, \Phi').\, (\ell : \{\tau_{1..n}\})$, pour tout $\dot\theta$ et Ψ. Ainsi on conclut simplement, si $\dot\theta, \Psi \models (\mu, \Phi').\, C$, alors $\dot\theta, \Psi \models (\mu, \Phi').\, (\ell : \{\tau_{1..n}\} \wedge C)$.

Cas [R'-BLK]. Vérifions, lorsque $\mu(\ell) = \mathtt{Blk}(w', w'')$ et $\{\dot\alpha_{1..m}\,;\,\dot\beta_{1..m}\} \# fv(\{\tau_{1..m}\}) \cup fv(\Phi') \cup fv(C)$ que pour tout $\dot\theta$ et Ψ, si :

$$\dot\theta, \Psi \models \exists\dot\alpha_{1..m}.\,\dot\beta_{1..m}.\, \left\{ \begin{array}{l} \Diamond\{\tau_{1..m}\} \\ \bigwedge_{i=1..m}[\dot\alpha_i\,;\,\dot\beta_i] \bigtriangledown \tau_i \\ \\ (\mu, \Phi' \cup \{\ell : \{\tau_{1..n}\}\}).\, \left\{ \begin{array}{l} w_1 : \{\dot\alpha_{1..m}\} \\ w_2 : \{\dot\beta_{1..m}\} \\ C \end{array} \right. \end{array} \right.$$

alors $\dot\theta, \Psi \models (\mu, \Phi').\, (\ell : \{\tau_{1..n}\} \wedge C)$.

Par définition du premier jugement à l'aide des règles [C'-EXISTS], [C'-$\Diamond$], [C'-$\bigtriangledown$], [C'-ORIGHEAP] et [C'-ORIGQUESTION], il existe deux ensembles de types $\tau'_{1..m}$ et $\tau''_{1..m}$ sans variables existentielles et un environnement de typage Ψ'' tel que :

(a) $\Diamond\{\dot\theta(\tau_{1..m})\}$

(b) $\forall i = 1..m.\, [\tau'_i\,;\,\tau''_i] \bigtriangledown \dot\theta(\tau_i)$

(c) $\Psi''' = \curlywedge\{\dot\theta(\Phi' \cup \{\ell : \curlywedge\{\tau_{1..n}\})\} \curlywedge \Psi''$ et $dom(\Psi''') = dom(\mu)$;

(d) $\forall \ell \in dom(\Psi'').\, \Psi \oplus \Psi''' \vdash \mu(\ell) : \Psi''(\ell)$;

(e) $\dot\theta, \Psi \oplus \Psi''' \models C$

(f) $\Psi \oplus \Psi''' \vdash w' : \curlywedge\{\tau'_{1..m}\}$

(g) $\Psi \oplus \Psi''' \vdash w'' : \curlywedge\{\tau''_{1..m}\}$

Les hypothèses (a), (b), permettent d'appliquer le lemme 3.4.6 pour vérifier $[\curlywedge\{\tau'_{1..m}\}; \curlywedge\{\tau''_{1..m}\}] \triangledown \curlywedge\{\dot\theta(\tau_{1..m})\}$. On en déduit à l'aide des hypothèses (f) et (g) et de la règle [H-Blk] :

$$\Psi \oplus \Psi''' \vdash \mathtt{Blk}(w', w'') : \curlywedge\{\dot\theta(\tau_{1..m})\}$$

Comme par hypothèse d'application $\mu(\ell) = \mathtt{Blk}(w', w'')$, ce jugement de typage permet d'étendre l'hypothèse (d) avec $\Psi \oplus \Psi''' \vdash \mu(\ell) : \curlywedge\{\dot\theta(\tau_{1..m})\}$. En remarquant ensuite que par associativité et commutativité de l'anti-unification $\Psi''' = \curlywedge\{\dot\theta(\Phi')\} \curlywedge (\Psi'' \oplus \{\ell : \curlywedge\{\dot\theta(\tau_{1..m})\})$, on en déduit à l'aide de l'hypothèse (c) et des règles [C'-OrigHeap] et [C'-OrigQuestion] : $\dot\theta, \Psi \models (\mu, \Phi').(\ell : \{\tau_{1..m}\})$. On en déduit à l'aide de l'hypothèse (e) : $\dot\theta, \Psi \models (\mu, \Phi').(\ell : \{\tau_{1..m}\} \wedge C)$. On conclut en particulier :

$$\dot\theta, \Psi \models (\mu, \Phi').(\ell : \{\tau_{1..n}\} \wedge C)$$

Cas [R'-Clos]. Vérifions lorsque $\mu(\ell) = \langle \forall \bar\alpha.\, (\Gamma \Rightarrow \tau' \to \tau''), \rho\rangle$ et $\bar\alpha_{1..m} \# ftv(\tau_{1..m}) \cup ftv(\Phi') \cup ftv(C)$, que pour tout $\dot\theta$ et Ψ, si :

$$\dot\theta, \Psi \models \exists \bar{\dot\alpha}_{1..m}. \begin{cases} \bigwedge_{i=1..m} \tau_i = \dot\tau'_i \to \dot\tau''_i \\ (\mu, \Phi' \cup \{\ell : \{\tau_{1..n}\}\}).\, (C \wedge \bigwedge_{x \in dom(\rho)} (\rho(x) : \{\dot\Gamma_{1..m}(x)\})) \end{cases}$$

alors : $\dot\theta, \Psi \models (\mu, \Phi').(\ell : \{\tau_{1..n}\} \wedge C)$.

Par définition du premier jugement à l'aide des règles [C'-Exists], [C'-Equal], [C'-OrigHeap] et [C'-OrigQuestion], il existe un ensemble de type $\tau'''_{1..m}$ et un environnement de typage Ψ'' tel que :

(a) $\dot\theta' = \dot\theta \oplus \{\dot\alpha_i \mapsto \tau'''_i\}_{i=1..m}$

(b) $\forall i = 1..n,\, \dot\theta(\tau_i) = \dot\theta'(\dot\tau'_i) \to \dot\theta'(\dot\tau''_i)$

(c) $\Psi''' = \curlywedge\{\dot\theta(\Phi' \cup \{\ell : \curlywedge\{\tau_{1..n}\})\} \curlywedge \Psi''$ et $dom(\Psi''') = dom(\mu)$;

(d) $\forall \ell \in dom(\Psi'').\, \Psi \oplus \Psi''' \vdash \mu(\ell) : \Psi''(\ell)$;

(e) $\dot\theta', \Psi \oplus \Psi''' \models C$

(f) $\forall x \in dom(\rho).\, \Psi \oplus \Psi''' \vdash \rho(x) : \curlywedge\{\dot\theta'(\dot\Gamma_{1..m}(x))\}$

L'hypothèse (b) permet, pour tout i de 1 à m, de vérifier $\Gamma \Rightarrow \tau' \to \tau'' \preccurlyeq \dot\theta'(\dot\Gamma_i) \Rightarrow \dot\theta'(\dot\tau'_i) \to \dot\theta'(\dot\tau''_i)$. L'hypothèse (f) et le lemme 3.4.7 permettent d'en déduire :

$$\Psi \oplus \Psi''' \vdash \langle \forall \bar\alpha.\, (\Gamma \Rightarrow \tau' \to \tau''), \rho\rangle : \curlywedge\{\dot\theta'(\tau_{1..m})\}$$

Comme par hypothèse d'application $\mu(\ell) = \langle \forall \bar\alpha.\, (\Gamma \Rightarrow \tau' \to \tau''), \rho\rangle$, ce jugement permet d'étendre l'hypothèse (d) avec $\Psi \oplus \Psi''' \vdash \mu(\ell) : \curlywedge\{\dot\theta'(\tau_{1..m})\}$. En remarquant ensuite, comme dans le cas [R'-Blk], que par associativité et commutativité de l'anti-unification $\Psi''' = \curlywedge\{\dot\theta(\Phi')\} \curlywedge (\Psi'' \oplus \{\ell : \curlywedge\{\dot\theta(\tau_{1..m})\})$,

on en déduit à l'aide de l'hypothèse (c) et des règles [C'-OrigHeap] et [C'-OrigQuestion] : $\dot{\theta}', \Psi \models (\mu, \Phi').(\ell : \{\tau_{1..m}\})$. Comme par hypothèse d'application $\bar{\dot{\alpha}}_{1..m} \# ftv(\tau) \cup ftv(\Phi') \cup ftv(C)$, on peut restreindre le domaine de $\dot{\theta}'$, dans le jugement précédent et dans l'hypothèse (e) pour en déduire : $\dot{\theta}, \Psi \models (\mu, \Phi').(\ell : \{\tau_{1..m}\} \wedge C)$. On conclut en particulier :

$$\dot{\theta}, \Psi \models (\mu, \Phi').(\ell : \{\tau_{1..n}\} \wedge C)$$

– Décomposition et homogénéité

Cas [R'-∇-True]. La relation ∇ est stable par substitution (propriété 2.3.2).

Cas [R'-∇-False]. La contrainte `False` étant trivialement invalide, la correction de cette règle est immédiate.

Cas [R'-◊]. Vérifions, que pour tout $\dot{\theta}$ et Ψ, si

$$\dot{\theta}, \Psi \models \exists \dot{\alpha}_{(1,1)..(n,m)}. \bigwedge_{i=1..m} (\tau'_i = \mathtt{T}(\dot{\alpha}_{(1,i)}, \ldots, \dot{\alpha}_{(n,i)}))$$

alors $\dot{\theta}, \Psi \models \Diamond\{\mathtt{T}(\tau_1, \ldots, \tau_n) ; \tau'_{1..m}\}$. Par définition du premier jugement à l'aide des règles [C'-Exists] et [C'-Equal], il existe un ensemble de types $\tau_{(1,1)..(n,m)}$ tel que pour tout i de 1 à m, on vérifie $\dot{\theta}(\tau'_i) = \mathtt{T}(\tau_{(1,i)}, \ldots, \tau_{(n,i)})$. L'ensemble $\{\dot{\theta}(\tau'_{1..m})\}$ est donc homogène en partageant le constructeur de types `T`. On en conclut :

$$\dot{\theta}, \Psi \models \Diamond\{\mathtt{T}(\tau_1 , \ldots , \tau_n) ; \tau'_{1..m}\}$$

– Anti-unification

La correction des règles [R'-Join], [R'-Merge] et [R'-Simpl] découle directement des propriétés de l'anti-unification (lemme 3.4.1).

– Paramétricité

Cas [R-Univ]. Il suffit d'utiliser le type universel $\star$ pour instancier $\bar{\dot{\alpha}}_i$, $\bar{\dot{\beta}}$, $\bar{\dot{\gamma}}$. □

Théorème 3.4.9 (Correction). *Soit un tas μ, clos et ne contenant pas de valeur modifiable. Soit une adresse-mémoire $\ell \in dom(\mu)$ et un type τ. Si en utilisant les règles de la figure 3.2, la contrainte $(\mu, \emptyset).(\ell : \tau)$ se réécrit vers* `True` *alors il existe un environnement de typage simple Ψ tel que $\mu : \Psi$ et $\Psi \vdash \ell : \tau$.*

Preuve Le lemme 3.4.8 permet de vérifier que si une contrainte initiale $(\mu, \emptyset).(\ell : \tau)$ se réécrit vers `True` alors cette contrainte est valide. La validité de cette contrainte lorsque que le tas μ est clos implique l'existence d'un environnement de typage simple Ψ tel que $\mu : \Psi$ et $\Psi \vdash \ell : \tau$ (lemme 3.4.3). □

3.4.4 Semi-complétude

Cette section va détailler la preuve de semi-complétude du système de réécriture décrit à la figure 3.2. Comme pour le système précédent (voir section 3.3.4), pour effectuer les preuves de semi-complétude des règles [R'-Blk] et [R'-Clos] dans le cas général, il est nécessaire de caractériser le sous-ensemble des contraintes satisfaisables dont les hypothèses de type sont effectivement compatibles avec le tas. Dans le système inductif, ce sous-ensemble peut être caractérisé en remplaçant la règle [C'-OrigHeap] par la règle ci-dessous.

$$\text{[CS'-OrigHeap]}\quad \frac{\begin{array}{c} dom(\curlywedge\{\dot\theta(\Phi')\} \curlywedge \Psi'') = dom(\mu) \\ \mu : \Psi \oplus (\curlywedge\{\dot\theta(\Phi')\} \curlywedge \Psi'') \qquad \dot\theta, \Psi \oplus (\curlywedge\{\dot\theta(\Phi')\} \curlywedge \Psi'') \models^s C \end{array}}{\dot\theta, \Psi \models^s (\mu, \Phi').\, C}$$

Les autres règles ne sont pas modifiées. Avec cette définition, les notions de satisfaisabilité et de semi-satifaisabilité d'une contrainte initiale $(\mu, \emptyset).\,(w : \tau)$ sont identiques.

Lemme 3.4.10. *Pour chacune des règles de réécriture $C_1 \gg C_2$ de la figure 3.2, on vérifie pour tout environnement Ψ et toutes substitutions $\dot\theta$, si $\dot\theta, \Psi \models^s C_1$ alors $\dot\theta, \Psi \models^s C_2$.*

Preuve Les preuves de semi-complétude des règles de réécriture sont détaillées dans l'ordre de la figure 3.2. Les preuves identiques à celles de la section 3.4.3 ne sont pas détaillées.

– Entiers

Cas [R'-Int]**.** Vérifions que pour tout $\dot\theta$ et Ψ, si $\dot\theta, \Psi \models^s i : \{\tau_{1..n}\}$ alors $\dot\theta, \Psi \models^s \Diamond\{\tau_{1..n}\} \wedge i : \{\tau_1\}$. Par définition à l'aide des règles [C'-OrigQuestion] et [W-Int] de :

$$\dot\theta, \Psi \models^s i : \{\tau_{1..n}\}$$

on vérifie $i \vartriangle \curlywedge\{\dot\theta(\tau_{1..n})\}$. La relation $i \vartriangle \tau$ n'étant jamais vérifiée lorsque $\tau \equiv \alpha$, on peut en déduire $\curlywedge\{\dot\theta(\tau_{1..n})\} \not\equiv \alpha$, autrement dit par définition de l'anti-unification $\Diamond\{\dot\theta(\tau_{1..n})\}$ et $\dot\theta, \Psi \models^s \Diamond\{\tau_{1..n}\}$. Par ailleurs, la relation $\vartriangle$ étant stable par instanciation (propriété 2.3.2), on vérifie trivialement $i \vartriangle \dot\theta(\tau_1)$ et $\dot\theta, \Psi \models^s i \vartriangle \{\tau_1\}$. On conclut :

$$\dot\theta, \Psi \models^s \Diamond\{\tau_{1..n}\} \wedge i : \{\tau_1\}$$

– Étiquettes

Cas [R'-Blk]**.** Vérifions, lorsque $\mu(\ell) = \texttt{Blk}(w', w'')$ et $\Phi'(\ell) = \{\tau_{n+1..m}\}$ et $\{\dot\alpha_{1..m}\,;\,\dot\beta_{1..m}\} \# fv(\{\tau_{1..m}\}) \cup fv(\Phi') \cup fv(C)$ que, pour tout $\dot\theta$ et Ψ, si $\dot\theta, \Psi \models^s (\mu, \Phi').\,(\ell : \{\tau_{1..n}\} \wedge C)$, alors :

$$\dot\theta, \Psi \models^s \exists \dot\alpha_{1..m}.\, \dot\beta_{1..m}.\, \left\{ \begin{array}{l} \Diamond\{\tau_{1..m}\} \\ \bigwedge_{i=1..m}[\dot\alpha_i\,;\,\dot\beta_i] \bigtriangledown \tau_i \\ \\ (\mu, \Phi' \cup \{\ell : \{\tau_{1..n}\}\}).\, \left\{ \begin{array}{l} w_1 : \{\dot\alpha_{1..m}\} \\ w_2 : \{\dot\beta_{1..m}\} \\ C \end{array} \right. \end{array} \right.$$

Par définition à l'aide des règles [CS'-OrigHeap] et [C'-OrigQuestion] de :

$$\dot{\theta}, \Psi \models^{s} (\mu, \Phi').\, (\ell : \{\tau_{1..n}\} \wedge C)$$

il existe un environnement de typage Ψ'' tel que :

(a) $\Psi''' = \curlywedge\{\dot{\theta}(\Phi')\} \curlywedge \Psi''$ et $dom(\Psi''') = dom(\mu)$

(b) $\mu : \Psi \oplus \Psi'''$

(c) $\Psi \oplus \Psi''' \vdash \ell : \curlywedge\{\dot{\theta}(\tau_{1..n})\}$

(d) $\dot{\theta}, \Psi \oplus \Psi''' \models^{s} C$

Par définition à l'aide de la règle [W-Label] de l'hypothèse (c), on vérifie $(\Psi \oplus \Psi''')(\ell) \preccurlyeq \curlywedge\{\dot{\theta}(\tau_{1..n})\}$. L'hypothèse (a) permet alors de vérifier :

(e) $\Psi'''(\ell) \preccurlyeq \curlywedge\{\dot{\theta}(\tau_{1..m})\}$

Par hypothèse d'application $\mu(\ell) = \texttt{Blk}(w', w'')$, l'hypothèse (b) permet alors de vérifier en particulier $\Psi \oplus \Psi''' \vdash \texttt{Blk}(w', w'') : \Psi'''(\ell)$. Le lemme d'instanciation (lemme 2.3.6) et l'hypothèse (e) permettent alors de vérifier :

(f) $\Psi \oplus \Psi''' \vdash \texttt{Blk}(w', w'') : \curlywedge\{\dot{\theta}(\tau_{1..m})\}$

Par définition de ce jugement de typage à l'aide de la règle [H-Blk], il existe deux types τ' et τ'' tels que :

$$[\tau' ; \tau''] \triangledown \curlywedge\{\dot{\theta}(\tau_{1..m})\} \;\; ; \quad \Psi \oplus \Psi''' \vdash w' : \tau' \quad \text{et} \quad \Psi \oplus \Psi''' \vdash w'' : \tau''$$

La relation $\triangledown$ étant stable par substitution (propriété 2.3.3), il existe deux ensembles de types $\tau'_{1..m}$ et $\tau''_{1..m}$ tels que pour tout i de 1 à m, on vérifie $[\tau'_i ; \tau''_i] \triangledown \dot{\theta}(\tau_i)$ et $\tau' \preccurlyeq \tau'_i$ et $\tau'' \preccurlyeq \tau''_i$. Par principalité de l'anti-unification on vérifie alors $\tau' \preccurlyeq \curlywedge\{\tau'_{1..m}\}$ et $\tau'' \preccurlyeq \curlywedge\{\tau''_{1..m}\}$. On en déduit :

$$\Psi \oplus \Psi''' \vdash w' : \curlywedge\{\tau'_{1..m}\} \quad \text{et} \quad \Psi \oplus \Psi''' \vdash w'' : \curlywedge\{\tau''_{1..m}\}$$

Par ailleurs, la relation $\triangledown$ n'étant pas satisfaisable lorsque le type de droite est une variable de type universelle, on vérifie $\curlywedge\{\dot{\theta}(\tau_{1..m})\} \not\equiv \alpha$. Autrement dit, par définition de l'anti-unification, l'ensemble $\dot{\theta}(\tau_{1..m})$ est homogène. On a montré jusqu'à maintenant :

$$\dot{\theta}, \Psi \models^{s} \begin{cases} \Diamond\{\tau_{1..m}\} \\ \bigwedge_{i=1..m} [\tau'_i ; \tau''_i] \triangledown \tau_i \\ (\mu, \Phi').\, (w_1 : \{\tau'_{1..m}\} \wedge w_2 : \{\tau''_{1..m}\} \wedge C) \end{cases}$$

L'hypothèse (f) permet alors d'étendre l'ensemble des hypothèses mémorisées dans Φ' avec les types $\tau_{1..n}$, et le corollaire 3.4.5 permet ensuite de conclure en remplaçant les types $\tau'_{1..m}$ et $\tau''_{1..m}$ par des variables existentielles.

$$\dot{\theta}, \Phi \models^{s} \exists \dot{\alpha}_{1..m}.\, \dot{\beta}_{1..m}.\, \begin{cases} \Diamond\{\tau_{1..m}\} \\ \bigwedge_{i=1..m} [\dot{\alpha}_i ; \dot{\beta}_i] \triangledown \tau_i \\ \\ (\mu, \Phi' \cup \{\ell : \{\tau_{1..n}\}\}).\, \begin{cases} w_1 : \{\dot{\alpha}_{1..m}\} \\ w_2 : \{\dot{\beta}_{1..m}\} \\ C \end{cases} \end{cases}$$

Cas [R'-CLOS]. Vérifions lorsque $\mu(\ell) = \langle \forall \bar{\alpha}.\, (\Gamma \Rightarrow \tau' \to \tau''), \rho \rangle$ et $\Phi'(\ell) = \{\tau_{n+1..m}\}$ et $\bar{\dot{\alpha}}_{1..m} \# ftv(\tau_{1..m}) \cup ftv(\Phi') \cup ftv(C)$, que pour tout $\dot{\theta}$ et Ψ, si $\dot{\theta}, \Psi \models^{s} (\mu, \Phi').(\ell : \{\tau_{1..n}\} \wedge C)$ alors :

$$\dot{\theta}, \Psi \models^{s} \exists \bar{\dot{\alpha}}_{1..m}. \left\{ \begin{array}{l} \bigwedge_{i=1..m} (\tau_i = \dot{\tau}'_i \to \dot{\tau}''_i) \\ (\mu, \Phi' \cup \{\ell : \{\tau_{1..n}\}\}).\, (C \wedge \bigwedge_{x \in dom(\rho)} (\rho(x) : \{\dot{\Gamma}_{1..m}(x)\})) \end{array} \right.$$

Par définition à l'aide de la règle [CS'-ORIGHEAP] de :

$$\dot{\theta}, \Psi \models^{s} (\mu, \Phi').(\ell : \{\tau_{1..n}\} \wedge C)$$

il existe un environnement de typage Ψ'' tel que :

(a) $\Psi''' = \lambda\!\!\!\lambda\{\dot{\theta}(\Phi')\} \curlywedge \Psi''$ et $dom(\Psi''') = dom(\mu)$

(b) $\mu : \Psi \oplus \Psi'''$

(c) $\Psi \oplus \Psi''' \vdash \ell : \lambda\!\!\!\lambda\{\dot{\theta}(\tau_{n+1..m})\}$

(d) $\dot{\theta}, \Psi \oplus \Psi''' \models^{s} C$

Par définition à l'aide de la règle [W-LABEL] de l'hypothèse (c), on vérifie $(\Psi \oplus \Psi''')(\ell) \preccurlyeq \lambda\!\!\!\lambda\{\dot{\theta}(\tau_{1..n})\}$. L'hypothèse (a) permet alors de vérifier :

(e) $\Psi'''(\ell) \preccurlyeq \lambda\!\!\!\lambda\{\dot{\theta}(\tau_{1..m})\}$

Par hypothèse d'application $\mu(\ell) = \langle \forall \bar{\alpha}.\, (\Gamma \Rightarrow \tau' \to \tau''), \rho \rangle$, l'hypothèse (b) permet alors de vérifier en particulier $\Psi \oplus \Psi''' \vdash \langle \forall \bar{\alpha}.\, (\Gamma \Rightarrow \tau' \to \tau''), \rho \rangle : \Psi'''(\ell)$. Le lemme d'instanciation (lemme 2.3.6) et l'hypothèse (e) permettent alors de vérifier :

(f) $\Psi \oplus \Psi''' \vdash \langle \forall \bar{\alpha}.\, (\Gamma \Rightarrow \tau' \to \tau''), \rho \rangle : \lambda\!\!\!\lambda\{\dot{\theta}(\tau_{1..m})\}$

Par définition de ce jugement de typage à l'aide de la règle [H-CLOS], il existe un environnement de typage des termes Γ''' tel que :

$$\Gamma \Rightarrow \tau' \to \tau'' \preccurlyeq \Gamma''' \Rightarrow \lambda\!\!\!\lambda\{\dot{\theta}(\tau_{1..m})\} \quad \text{et} \quad \Psi \oplus \Psi''' \vdash \rho : \Gamma'''$$

Par instanciation, on obtient alors des environnements de typage $\Gamma_{1..m}$ tels que pour $i = 1$ à m, on vérifie :

$$\Gamma''' \Rightarrow \lambda\!\!\!\lambda\{\dot{\theta}(\tau_{1..m})\} \preccurlyeq \Gamma_i \Rightarrow \dot{\theta}(\tau_i) \;\; ; \quad \Psi \oplus \Psi''' \vdash \rho : \Gamma_i \quad \text{et} \quad \Gamma''' \equiv \lambda\!\!\!\lambda\{\Gamma_{1..m}\}$$

Autrement dit, en utilisant les notations $\dot{\Gamma}_i$ et $\dot{\tau}'_i$, il existe des substitutions $\dot{\theta}_{1..m}$ telles que pour $i = 1$ à m, on vérifie $\dot{\theta}_i(\dot{\Gamma}_i \Rightarrow \dot{\tau}'_i \to \dot{\tau}''_i) \equiv \Gamma_i \Rightarrow \dot{\theta}(\tau_i)$. Autrement dit, on a montré jusqu'à maintenant :

$$\dot{\theta} \oplus \dot{\theta}_{1..n}, \Psi \models^{s} \left\{ \begin{array}{l} \bigwedge_{i=1..m} (\tau_i = \dot{\tau}'_i \to \dot{\tau}''_i) \\ (\mu, \Phi').\, (C \wedge \rho : \{\dot{\Gamma}_{1..m}\}) \end{array} \right.$$

L'hypothèse (f) permet alors d'étendre l'ensemble des hypothèses mémorisées dans Φ' avec les types $\tau_{1..n}$ et la règle [C'-EXISTS] permet ensuite de conclure en remplaçant quantifiant existentiellement le domaine des substitutions $\dot{\theta}_{1..m}$.

$$\dot{\theta}, \Psi \models^{s} \exists \bar{\dot{\alpha}}_{1..m}. \left\{ \begin{array}{l} \bigwedge_{i=1..m} (\tau_i = \dot{\tau}'_i \to \dot{\tau}''_i) \\ (\mu, \Phi' \cup \{\ell : \{\tau_{1..n}\}\}).\, (C \wedge \bigwedge_{x \in dom(\rho)} (\rho(x) : \{\dot{\Gamma}_{1..m}(x)\})) \end{array} \right.$$

– Décomposition et homogénéité

Cas [R'-∇-TRUE]**.** La décomposition d'un type τ par la relation ∇ est unique (propriété 2.3.4) et stable par substitution (propriété 2.3.3).

Cas [R'-∇-FALSE]**.** S'il n'existe aucun type τ_1 et τ_2 tels que $[\tau_1; \tau_2] \nabla \tau$, alors aucune substitution $\dot{\theta}$ ne validera $[\tau_1; \tau_2] \nabla \tau$.

Cas [R'-$\Diamond$]**.** Si pour une substitution $\dot{\theta}$ on vérifie :

$$\Diamond\{\mathtt{T}(\dot{\theta}(\tau_1), \ldots, \dot{\theta}(\tau_n)) ; \dot{\theta}(\tau'_{1..m})\}$$

alors les types $\dot{\theta}(\tau'_{1..m})$ partagent `T` comme constructeur de types.

– Anti-unification

La semi-complétude des règles [R'-JOIN], [R'-MERGE] et [R'-SIMPL] découle directement des propriétés de l'anti-unification (lemme 3.4.1).

□

Théorème 3.4.11 (Semi-complétude)**.** *Soit un tas μ, clos et ne contenant pas de valeur modifiable. Soit une adresse-mémoire $\ell \in dom(\mu)$ et un type τ. Si en utilisant les règles de la figure 3.2, la contrainte $(\mu, \emptyset).(\ell : \tau)$ se réécrit vers* `False` *alors il n'existe aucun environnement de typage simple Ψ tel que $\mu : \Psi$ et $\Psi \vdash \ell : \tau$.*

Preuve Le lemme 3.4.10 permet de vérifier que si une contrainte initiale $(\mu, \emptyset).(\ell : \tau)$ se réécrit vers `False`, alors cette contrainte est invalide. L'invalidité de cette contrainte lorsque que la tas μ est clos implique par contraposition du lemme 3.4.3 l'inexistence d'un environnement de typage simple Ψ tel que $\mu : \Psi$ et $\Psi \vdash \ell : \tau$. □

3.5 Système de réécriture paramétré par le système de types

Le système de réécriture basé sur le système de types généralisé (section 3.3, figure 3.1) et celui basé sur le système de types original (section 3.4, figure 3.2) ne partagent pas la même syntaxe de contraintes. L'objet de cette section est de montrer que le premier de ces systèmes de réécriture peut s'exprimer en utilisant la syntaxe du second. On obtient alors un système de réécriture dit paramétré proche de celui de la figure 3.2, où certaines conditions d'application dépendent du système de types sous-jacent. Cette paramétrisation permettra notamment de simplifier la prise en compte des blocs modifiables à la section 3.6 et de ne définir qu'une seule stratégie de réécriture à la section 3.7.

Dans la suite du document, lorsque les conditions d'application d'une règle de réécriture seront distinctes selon le système de types sous-jacent, la règle sera dédoublée : la règle applicable pour le système de types généralisé sera notée $C_1 \gg_\Phi C_2$, celle pour le système de types original sera notée $C_1 \gg_\Psi C_2$. Les règles notées $C_1 \gg C_2$ seront donc applicable indépendamment du système de types.

Ensemble de types homogène La contrainte d'homogénéité a été introduite dans le système de réécriture restreint au système de type original pour garantir la compatibilité d'un entier ou d'un bloc avec l'anti-unificateur des types attendus. Lorsque le système de types sous-jacent est le système généralisé, les types attendus pour une valeur partagée peuvent être vérifiés indépendamment et la contrainte d'homogénéité peut être ignorée. Pour cela, la règle [R'-◊] est remplacée dans le système de réécriture paramétré par les deux règles suivantes.

$$\Diamond\{\tau_{1..n}\} \gg_\Phi \texttt{True} \qquad [\text{R'-}\Diamond_\Phi]$$

$$\Diamond\{\texttt{T}(\tau_1,\ldots,\tau_n)\,;\,\tau'_{1..m}\} \gg_\Psi \exists\dot\alpha_{(1,1)..(n,m)}.\ \textstyle\bigwedge_{i=1..m}(\tau'_i = \texttt{T}(\dot\alpha_{(1,i)},\ldots,\dot\alpha_{(n,i)})) \quad si\ \dot\alpha_{(1,1)..(n,m)}\ \#\ fv(\{\tau'_{1..m}\}) \qquad [\text{R'-}\Diamond_\Psi]$$

Les règles de réécriture [R'-Blk] et [R'-Clos] s'appliquant dans le cas de contraintes de types multiples portant respectivement sur un bloc ou sur une fermeture n'ont alors pas besoin d'être modifiées. Par contre, la règle [R'-Int] s'appliquant dans le cas d'une contrainte de types multiples portant sur un entier doit être adaptée au système de types généralisé : il est nécessaire de vérifier la compatibilité de l'entier avec chacun des types attendus. Dans le cas du système de types original, ces vérifications multiples sont superflue mais correcte. La règle [R'-Int] peut alors être remplacée dans le système de réécriture paramétré par la règle ci-dessous.

$$i : \{\tau_{1..n}\} \gg \Diamond\{\tau_{1..n}\} \wedge i : \{\tau_1\} \wedge \ldots \wedge i : \{\tau_n\} \qquad si\ n \geq 2 \qquad [\text{R'-Int}_{\Psi,\Phi}]$$

Les règles [R'-Int-True] et [R'-Int-False] s'appliquant respectivement dans le cas des contraintes de type singleton portant sur un entier n'ont pas besoin d'être modifiées.

Anti-unification et instanciation Les règles effectuant un calcul d'anti-unification dans une contrainte de types multiples ou dans un ensemble d'hypothèse de types, à savoir les règles [R'-Merge] et [R'-Simpl], doivent être adaptées au système de type généralisé en suivant le principe de la règle [R-Merge] définit à la section 3.3.2.

$$\ell : \{\tau_1\,;\,\tau_2\,;\,\tau_{3..n}\} \gg_\Phi \ell : \{\tau_1\,;\,\tau_{3..n}\} \qquad si\ \forall\dot\theta.\,\dot\theta(\tau_1) \preccurlyeq \dot\theta(\tau_2) \qquad [\text{R'-Merge}_\Phi]$$

$$\ell : \{\tau_1\,;\,\tau_2\,;\,\tau_{3..n}\} \gg_\Psi \ell : \{\tau\,;\,\tau_{3..n}\} \qquad si\ \forall\dot\theta.\,\dot\theta(\tau) = \dot\theta(\tau_1) \curlywedge \dot\theta(\tau_2) \qquad [\text{R'-Merge}_\Psi]$$

$$(\mu, \Phi' \cup \{\ell : \{\tau_1\,;\,\tau_2\}\}).\,C \gg_\Phi (\mu, \Phi' \cup \{\ell : \{\tau_1\}\}).\,C \quad si\ \forall\dot\theta.\,\dot\theta(\tau_1) \preccurlyeq \dot\theta(\tau_2) \qquad [\text{R'-Simpl}_\Phi]$$

$$(\mu, \Phi' \cup \{\ell : \{\tau_1\,;\,\tau_2\}\}).\,C \gg_\Psi (\mu, \Phi' \cup \{\ell : \{\tau\}\}).\,C \quad si\ \forall\dot\theta.\,\dot\theta(\tau) = \dot\theta(\tau_1) \curlywedge \dot\theta(\tau_2) \qquad [\text{R'-Simpl}_\Psi]$$

De même, dans le cas du système de types généralisé, la condition d'application de la règle [R'-Merge'] doit adaptée en suivant le principe de la règle [R-Merge'].

$$(\mu, \Phi').\,(\ell : \{\tau_{1..n}\} \wedge C) \gg_\Phi (\mu, \Phi').\,C \quad si\ \ell \in dom(\Phi')\ et\ \forall\dot\theta.\,\forall i = 1..n,\ \exists\tau \in \Phi'(\ell).\,\dot\theta(\tau') \preccurlyeq \dot\theta(\tau_i) \qquad [\text{R'-Merge'}_\Phi]$$

$$(\mu, \Phi').\,(\ell : \{\tau_{1..n}\} \wedge C) \gg_\Psi (\mu, \Phi').\,C \quad si\ \ell \in dom(\Phi')\ et\ \forall\dot\theta.\ \curlywedge\{\dot\theta(\Phi'(\ell))\} \preccurlyeq \curlywedge\{\dot\theta(\tau_{1..n})\} \qquad [\text{R'-Merge'}_\Psi]$$

Comme pour la règle [R-Merge'], la condition d'application de la règle [R'-Merge'$_\Phi$] n'est pas toujours facile à vérifier ; la section 3.7.2 détaillera le principe de mise-en-œuvre de cette règle.

Propriétés du système de réécriture paramétré Les propriétés de correction, de semi-complétude et de (non) terminaison du système de réécriture paramétré sont, suivant le paramètre, les mêmes que dans les systèmes de réécriture non paramétrés des sections 3.3 et 3.4. Dans le cas de la restriction au système de types simple, les preuves sont celles décrites précédemment. Dans le cas du système de types généralisé, il faut dans un premier temps définir la notion de satisfaisabilité d'une contrainte de types multiples comme équivalente à la satisfaisabilité d'une conjonction de contrainte de type singleton pour chacun des types attendus ; et définir le prédicat d'homogénéité comme trivialement satisfaisable. Dans le système inductif, cela peut être décrit par les règles ci-dessous.

$$\frac{\Phi \vdash w : \tau_n \qquad \ldots \qquad \Phi \vdash w : \tau_n}{\dot{\theta}, \Phi \models w : \{\tau_{1..n}\}}\ [\text{C-QuestionSet}] \qquad\qquad \frac{}{\dot{\theta}, \Phi \models \Diamond\{\tau_{1..n}\}}\ [\text{C-}\Diamond]$$

Dans un second temps, les preuves de la section 3.3 s'adaptent directement à la nouvelle syntaxe.

3.6 Valeurs modifiables

Les systèmes de réécriture décris précédemment ne considèrent pas le cas des tas contenant des valeurs modifiables. L'objet de cette section est d'ajouter des règles de réécriture au système de réécriture paramétré permettant la vérification d'un bloc modifiable représentant une référence.

La preuve de correction du typage décrite au chapitre 2 a montré que les valeurs modifiables effectivement allouées dans le tas sont monomorphes. Ce principe a été transcrit dans les systèmes de typage simple et généralisé aux sections 3.1.1 et 3.1.2 par les conditions suivantes :

- si $\mu : \Psi$ alors $\forall \ell \in dom(\Psi)$ si $\Psi(\ell) = \texttt{Ref}(\tau)$ alors $fv(\tau) = \emptyset$;
- si $\mu : \Phi$ alors $\forall \ell \in dom(\Phi)$ si $\texttt{Ref}(\tau) \in \Phi(\ell)$ alors $fv(\tau) = \emptyset$ et $|\Phi(\ell)| = 1$.

Par ailleurs, comme détaillé à la section 3.1, la représentation mémoire utilisée par le compilateur OCaml ne permet pas de distinguer explicitement les blocs modifiables des blocs non modifiables. Du point de vue de l'algorithme de vérification, seul le type attendu va permettre de les distinguer. Une possibilité pour prendre en compte les références dans le système de réécriture est alors de vérifier lorsqu'un des types attendus pour un bloc est une référence que tous les types attendus sont égaux et sans variable universelle. Pour ne pas dupliquer la règle [R'-Blk], ce test peut être reporté dans le prédicat d'homogénéité, en considérant un ensemble de types contenant un type $\texttt{Ref}(\tau)$ comme homogène uniquement lorsque cet ensemble est un singleton et que τ est clos. Dans la définition inductive de satisfaisabilité du système de réécriture basé sur le système de types simple, ce principe peut se traduire en remplaçant la règle [C'-$\Diamond$] par les

règles suivantes.

$$\frac{[\text{C'-}\Diamond]\quad \dot\theta(\tau_1) \equiv \mathtt{T}(\tau_1^1,\ldots,\tau_1^m) \quad \ldots \quad \dot\theta(\tau_n) \equiv \mathtt{T}(\tau_n^1,\ldots,\tau_n^m) \quad \mathtt{T} \not\equiv \mathtt{Ref}}{\dot\theta, \Psi \models \Diamond\{\tau_{1..n}\}}$$

$$\frac{[\text{C'-}\Diamond\text{-Ref}]\quad \dot\theta(\tau_1) \equiv \mathtt{Ref}(\tau) \quad \ldots \quad \dot\theta(\tau_n) \equiv \mathtt{Ref}(\tau) \quad fv(\tau) = \emptyset}{\dot\theta, \Psi \models \Diamond\{\tau_{1..n}\}}$$

Dans le cas du système de types général, il se traduit en remplaçant la règle [C-$\Diamond$] par les règles suivantes.

$$\frac{[\text{C-}\Diamond]\quad \dot\theta(\tau_1) \equiv \mathtt{T_1}(\tau_1^1,\ldots,\tau_1^{m_1}) \quad \mathtt{T_1} \not\equiv \mathtt{Ref} \quad \ldots \quad \dot\theta(\tau_n) \equiv \mathtt{T_n}(\tau_n^1,\ldots,\tau_n^{m_n}) \quad \mathtt{T_n} \not\equiv \mathtt{Ref}}{\dot\theta, \Phi \models \Diamond\{\tau_{1..n}\}}$$

$$\frac{[\text{C-}\Diamond\text{-Ref}]\quad \dot\theta(\tau_1) \equiv \mathtt{Ref}(\tau) \quad \ldots \quad \dot\theta(\tau_n) \equiv \mathtt{Ref}(\tau) \quad fv(\tau) = \emptyset}{\dot\theta, \Phi \models \Diamond\{\tau_{1..n}\}}$$

Règles de réécriture Ainsi, dans le système de réécriture paramétré, un prédicat d'homogénéité contenant un type de la forme $\mathtt{Ref}(\_)$ peut-être résolu par les règles suivantes.

$$\Diamond\{\mathtt{Ref}(\tau)\,;\,\tau_{1..n}\} \gg \Diamond\{\mathtt{Ref}(\tau)\} \wedge \textstyle\bigwedge_{i=1..n}(\tau_i = \mathtt{Ref}(\tau)) \qquad [\text{R'-}\Diamond\text{-Ref}]$$
$$\Diamond\{\mathtt{Ref}(\tau)\} \gg \mathtt{True} \quad si\ fv(\tau) = \emptyset \qquad [\text{R'-}\Diamond\text{-Ref-True}]$$
$$\Diamond\{\mathtt{Ref}(\tau)\} \gg \mathtt{False} \quad si\ fuv(\tau) \neq \emptyset \qquad [\text{R'-}\Diamond\text{-Ref-False}]$$

Avec ces règles de réécriture, une contrainte de la forme $\Diamond\{\mathtt{Ref}(\tau)\}$ où τ n'est pas clos, ne peut pas se réécrire tant que les variables libres de τ sont uniquement des variables existentielles.

Dans le cas du système de types original, la règle de résolution du prédicat d'homogénéité par propagation d'un constructeur de types de tête doit être complétée avec une condition excluant la propagation du constructeur $\mathtt{Ref}$.

$$\Diamond\{\mathtt{T}(\tau_1,\ldots,\tau_n)\,;\,\tau'_{1..m}\} \gg_\Psi \exists\dot\alpha_{(1,1)..(n,m)}.\ \textstyle\bigwedge_{i=1..m}(\tau'_i = \mathtt{T}(\dot\alpha_{(1,i)},\ldots,\dot\alpha_{(n,i)})) \qquad [\text{R'-}\Diamond_\Psi]$$
$$si\ \dot\alpha_{(1,1)..(n,m)}\ \#\ fv(\{\tau'_{1..m}\})\ et\ \mathtt{T} \not\equiv \mathtt{Ref}$$

Dans le cas du système de types généralisé, la résolution triviale du prédicat d'homogénéité vers $\mathtt{True}$ doit être complétée avec la condition d'application : tous les constructeurs de types de tête doivent être connus et distincts de $\mathtt{Ref}$.

$$\Diamond\{\tau_{1..n}\} \gg_\Phi \mathtt{True} \quad si\ \forall i = 1..n.\ \tau_i \not\equiv \dot\alpha\ et\ \tau_i \not\equiv \mathtt{Ref}(\_) \qquad [\text{R'-}\Diamond_\Phi]$$

Avec cette dernière règle, une contrainte de la forme $\Diamond\{\mathtt{T}(\tau_1,\ldots,\tau_n)\,;\,\dot\alpha\}$ ne peut pas se réécrire tant que la variable $\dot\alpha$ n'est pas instanciée. Néanmoins, dans ce cas comme dans le cas $\Diamond\{\mathtt{Ref}(\dot\alpha)\}$, si de telles contraintes ne peuvent toujours pas se réécrire après

que toutes les autres contraintes aient été résolues, il est possible de les résoudre en instanciant les variables libres restantes avec le type universel $\star$. Pour traduire cela dans le système de réécriture, on adapte la règle [R'-Univ].

$$\exists \bar{\beta}\bar{\gamma}.\ \overline{\Diamond\{\tau_{1..n}\}} \wedge \overline{\Diamond\{\mathtt{Ref}(\tau)\}} \wedge \overline{[\tau' ; \tau''] \triangledown \dot{\beta}} \wedge \overline{i : \{\dot{\gamma}\}} \gg_\Phi \mathtt{True} \quad si\ \overline{\forall i = 1..n.\ \tau_i \not\equiv \mathtt{Ref}(\_)}\ et\ \overline{fuv(\tau) = \emptyset} \qquad [\text{R'-Univ}_\Phi]$$

$$\exists \bar{\alpha}_i\bar{\beta}\bar{\gamma}.\ \overline{\Diamond\{\dot{\alpha}_{1..n}\}} \wedge \overline{\Diamond\{\mathtt{Ref}(\tau)\}} \wedge \overline{[\tau' ; \tau''] \triangledown \dot{\beta}} \wedge \overline{i : \{\dot{\gamma}\}} \gg_\Psi \mathtt{True} \quad si\ \overline{fuv(\tau) = \emptyset} \qquad [\text{R'-Univ}_\Psi]$$

Avec ces deux règles, les formes normales sont à nouveau restreintes à `True` et `False`.

3.7 Stratégies de réécriture et terminaison

La section 3.3.2 a montré que le système de réécriture basé sur le système de types généralisé peut ne pas terminer en présence de cycles dans le tas. Les sections 3.4.1 et 3.4.2 ont montré ensuite que la restriction du système de réécriture au système de types original permet de retrouver la propriété de terminaison en l'absence de fermeture polymorphe nécessitant l'explicitation d'inconnues de type. L'objet de cette section est de faire le lien avec le prototype décrit au chapitre suivant, en décrivant un compromis permettant de garantir la terminaison de l'algorithme dans tous les cas, en renonçant le moins possible à la propriété de semi-complétude.

La section 3.7.1 définit une stratégie de réécriture basée sur un tri topologique permettant dans un premier temps un gain d'efficacité. Dans un second temps, cette stratégie de réécriture sera utilisée à la section 3.7.2 pour décrire le compromis garantissant la terminaison.

3.7.1 Tri topologique

La section 3.2 avait permis la description d'une première version du système de réécriture ne considérant que les tas sans fermeture, ni valeur modifiable, ni cycle lorsque le système de types sous-jacent est le système généralisé. Dans ce système, il était possible de définir une stratégie de réécriture basée sur un tri topologique du tas qui améliore l'efficacité de l'algorithme : pour ne pas vérifier plusieurs fois avec le même type un bloc partagé, on attend d'avoir collecté l'ensemble des types attendus pour ce bloc avant de le vérifier. Le tri du tas suivant un ordre topologique a pu être décrit dans le système à l'aide de la règle [R-Sort] reproduite ci-dessous.

$$\mu.\, C \gg \mu_1.\, \mu_2.\, C \qquad si\ \mu_1 \uplus \mu_2 = \mu\ et\ fl(img(\mu_1))\ \#\ dom(\mu_2) \qquad [\text{R-Sort}]$$

Une fois le tas trié, l'ordre de vérification des blocs était alors contraint par les règles [R-Blk-...] — ces règles ne peuvent s'appliquer qu'à un bloc appartenant au fragment du tas le plus profond dans la syntaxe d'une contrainte — et par la règle [R-Heap] reproduite ci-dessous, qui permet de faire disparaître d'une contrainte un bloc sur lequel ne porte plus aucune contrainte et ainsi permettre la vérification du bloc partagé suivant.

$$\mu.\, C \gg C \qquad si\ fl(C)\ \#\ dom(\mu) \qquad [\text{R-Heap}]$$

L'objet de cette section est d'étudier s'il est possible d'appliquer une stratégie similaire dans le système de réécriture paramétré.

L'ajout d'hypothèses de type dans la syntaxe des contraintes à la section 3.3.2, pour permettre la vérification de tas contenant des cycles, ne remet pas en cause ce principe : la règle ci-dessous permet de décomposer le tas en composantes fortement connexes triées selon un ordre topologique.

$$(\mu, \Phi).\, C \gg (\mu_1, \Phi_1).\, (\mu_2, \Phi_2).\, C \qquad \text{[R'-Sort]}$$
$$si\ \mu_1 \uplus \mu_2 = \mu\ et\ fl(img(\mu_1))\ \#\ dom(\mu_2)\ et\ \Phi_i = \Phi_{|dom(\mu_i)}$$

Suivant le même principe que précédemment, la vérification du tas se fait alors composante par composante. Par contre, la syntaxe de contraintes utilisée ici ne semble par permettre de détailler plus en avant une stratégie de réécriture à l'intérieur d'une composante fortement connexe. Le prototype décrit au chapitre 4 utilise une stratégie « récursive » : la stratégie utilisée pour le tas global sera appliquée à l'intérieur d'une composante fortement connexe, en triant topologiquement le sous graphe de la composante privée de ses racines [9].

Variables existentielles L'ajout d'une quantification existentielle dans la syntaxe des contraintes à la section 3.3.1 pour prendre en compte les valeurs fonctionnelles complique la définition d'une stratégie de réécriture basée sur un tri topologique, comme le montre l'exemple suivant. En nommant $\mathtt{T}$ le type récursif $\mathtt{T} \equiv \mathtt{Option}(\mathtt{T} \to \mathtt{Int})$, la fonction ci-dessous admet le type $\mathtt{T} \to \mathtt{Int}$:

$$\begin{array}{l}\mathtt{let}\ f = \lambda t.\\ \quad \mathtt{case}\ t\ \mathtt{of}\\ \quad |\ \mathtt{None} \to 0\\ \quad |\ \mathtt{Some}(g) \to g\ \mathtt{None}\end{array}$$

Et le programme suivant a pour type $\mathtt{Unit} \to \mathtt{Int}$:

$$\begin{array}{l}\mathtt{let}\ delay = \lambda f.\lambda x.\lambda y.\ f\ x\ \mathtt{in}\\ delay\ f\ \mathtt{Some}(f)\end{array}$$

Le tas obtenu par évaluation de ce programme est :

$$\mu \equiv \{\ \begin{array}{lcl}\ell_0 & \mapsto & \langle \lambda y.\ f\ x, \{f \mapsto \ell_2; x \mapsto \ell_1\}\rangle\\ \ell_1 & \mapsto & \mathtt{Blk}(\ell_2, 0)\\ \ell_2 & \mapsto & \langle \lambda t.\, \mathtt{case}\ t\ \mathtt{of}\ \ldots, \emptyset\rangle\end{array}\ \}$$

Comme précédemment, le type associé au pointeur de code représentant $\lambda y.\ f\ x$ est $\forall \alpha\beta\gamma.\, (\{x : \alpha\,;\, f : \alpha \to \beta\} \Rightarrow \gamma \to \beta)$. En utilisant les règles de réécriture de la figure 3.1

9. Dans ce contexte, les racines de la composante sont les valeurs de la composante sur lesquelles la vérification du contexte a propagé des contraintes de types.

(page 101) et en décomposant préalablement le tas en $\mu_2.\mu_1.\mu_0$ où $\mu_i \equiv \{\ell_i \mapsto \mu(\ell_i)\}$, la vérification de compatibilité du tas μ avec $\texttt{Unit} \to \texttt{Int}$ est bloquée[10] :

$$\begin{array}{rcl}
 & & \mu.\,(\ell_0 : \{\texttt{Unit} \to \texttt{Int}\}) \\
[\text{R-Sort}][\text{R-Sort}] & \gg & \mu_2.\mu_1.\mu_0.\,(\ell_0 : \{\texttt{Unit} \to \texttt{Int}\}) \\
[\text{R-Clos}] & \gg & \exists\dot\alpha\dot\beta\dot\gamma.\, \left\{ \begin{array}{l} \texttt{Unit} \to \texttt{Int} = \dot\gamma \to \dot\beta \\ \mu_2.\mu_1.\mu_0.\,(\ell_2 : \{\dot\alpha \to \dot\beta\} \wedge \ell_1 : \{\dot\alpha\}) \end{array} \right. \\
[\text{R-Unif}] & \gg & \exists\dot\alpha.\,\mu_2.\mu_1.\mu_0.\,(\ell_2 : \{\dot\alpha \to \texttt{Int}\} \wedge \ell_1 : \{\dot\alpha\}) \\
[\text{R-Heap}] & \gg & \exists\dot\alpha.\,\mu_2.\mu_1.\,(\ell_2 : \{\dot\alpha \to \texttt{Int}\} \wedge \ell_1 : \{\dot\alpha\})
\end{array}$$

À cette étape, la vérification de $\mu(\ell_1)$ semble impossible car la variable $\dot\alpha$ n'est pas encore instanciée et la décomposition du tas en fragments triés topologiquement interdit la vérification de $\mu(\ell_2)$ qui aurait permis cette instanciation.

Heureusement, l'explicitation dans le système paramétré de contraintes représentant le prédicat d'homogénéité et le propagateur de type permet de débloquer la situation ci-dessus sans renoncer au tri topologique. Pour cela, la règle [R'-Blk], reproduite ci-dessous, peut commencer la vérification d'un bloc dont tous les types attendus sont des variables existentielles : elle introduit alors deux ensembles de variables existentielles $\dot\alpha_{1..m}$ et $\dot\beta_{1..m}$ représentant respectivement les types attendus pour les deux éléments du bloc et elle explicite un prédicat d'homogénéité et des propagateurs de type à vérifier ultérieurement.

$$\begin{array}{l}
(\mu, \Phi').\,(\ell : \{\tau_{1..n}\} \wedge C) \gg \qquad\qquad [\text{R'-Blk}] \\
\quad \exists\dot\alpha_{1..m}.\,\dot\beta_{1..m}.\, \left\{ \begin{array}{l} \Diamond\{\tau_{1..m}\} \\ \bigwedge_{i=1..m}[\dot\alpha_i\,;\,\dot\beta_i] \,\triangledown\, \tau_i \\ (\mu, \Phi' \cup \{\ell : \{\tau_{1..n}\}\}).\,(w' : \{\dot\alpha_{1..m}\} \wedge w'' : \{\dot\beta_{1..m}\} \wedge C) \\ \qquad si\ \mu(\ell) = \texttt{Blk}(w', w'')\ et\ \Phi'(\ell) = \{\tau_{n+1..m}\} \\ \qquad et\ \{\dot\alpha_{1..m}\,;\,\dot\beta_{1..m}\} \,\#\, fv(\{\tau_{1..m}\}) \cup fv(\Phi') \cup fv(C) \end{array} \right.
\end{array}$$

Pour continuer l'exemple ci-dessus, on obtient alors la suite de réécriture suivante :

$$\begin{array}{rcl}
 & & \exists\dot\alpha.\,\mu_2.\mu_1.\,(\ell_2 : \{\dot\alpha \to \texttt{Int}\} \wedge \ell_1 : \{\dot\alpha\}) \\
[\text{R'-Blk}] & \gg & \exists\dot\alpha\dot\beta\dot\gamma.\, \left\{ \begin{array}{l} \Diamond\{\dot\alpha\} \wedge [\dot\beta\,;\,\dot\gamma] \,\triangledown\, \dot\alpha \\ \mu_2.\mu_1.\,(\ell_2 : \{\dot\alpha \to \texttt{Int}\} \wedge \ell_2 : \{\dot\beta\} \wedge 0 : \{\dot\gamma\}) \end{array} \right. \\
[\text{R'-Join}] & \gg & \exists\dot\alpha\dot\beta\dot\gamma.\, \left\{ \begin{array}{l} \Diamond\{\dot\alpha\} \wedge [\dot\beta\,;\,\dot\gamma] \,\triangledown\, \dot\alpha \\ \mu_2.\mu_1.\,(\ell_2 : \{\dot\alpha \to \texttt{Int}; \dot\beta\} \wedge 0 : \{\dot\gamma\}) \end{array} \right. \\
[\text{R'-Heap}][\text{R'-Clos}] & \gg & \exists\dot\alpha\dot\beta\dot\gamma.\, \left\{ \begin{array}{l} \Diamond\{\dot\alpha\} \wedge [\dot\beta\,;\,\dot\gamma] \,\triangledown\, \dot\alpha \\ (\dot\alpha \to \texttt{Int} = \texttt{T} \to \texttt{Int}) \wedge (\dot\beta = \texttt{T} \to \texttt{Int}) \\ \mu_2.\,(0 : \{\dot\gamma\}) \end{array} \right. \\
[\text{R'-Unif}] & \gg & \exists\dot\gamma.\, \left\{ \begin{array}{l} \Diamond\{\texttt{T}\} \wedge [\texttt{T} \to \texttt{Int}\,;\,\dot\gamma] \,\triangledown\, \texttt{T} \\ \mu_2.\,(0 : \{\dot\gamma\}) \end{array} \right. \\
[\text{R'-}\triangledown\text{-True}][\text{R'-Unif}] & \gg & \Diamond\{\texttt{T}\} \wedge \mu_2.\,(0 : \{\texttt{Unit}\}) \\
[\text{R'-}\Diamond_\Phi] & \gg_\Phi & \mu_2.\,(0 : \{\texttt{Unit}\}) \\
[\text{R'-Heap}][\text{R'-Int-True}] & \gg & \texttt{True}
\end{array}$$

10. Le tas μ ne contenant pas de cycle, on ignore dans cet exemple les hypothèses de type mémorisées.

Pour ne pas introduire trop de variables existentielles, une stratégie d'application efficace retardera autant que possible l'application d'une règle [R'-Blk] lorsque l'ensemble des types attendus est un ensemble de variables existentielles. Si comme dans l'exemple une telle application est nécessaire, on dira que l'on *force* la vérification du bloc.

Tri topologique paresseux Si l'utilisation d'un système de réécriture pour modéliser l'algorithme laisse une certaine liberté dans l'ordre d'application des règles et a permis la définition d'une stratégie de réécriture retardant la vérification de certaines valeurs, le décomposition explicite du tas en fragments triés topologiquement contraint parfois trop fortement l'ordre de parcours. En effet, une fois le tas trié, le système de réécriture proposé ici ne permet plus de retarder la vérification d'un bloc au bénéfice d'un bloc appartenant à une autre composante fortement connexe, même si les deux composantes sont indépendantes les unes des autres. En pratique, pour éviter cette contrainte, le prototype décrit au chapitre 4 utilise, en l'absence de cycle, un tri topologique « paresseux » basé sur un compteur de références (voir en particulier la **??**).

3.7.2 Terminaison ou semi-complétude

Lors de la vérification d'une contrainte de types sur une adresse-mémoire pour laquelle des hypothèses de type ont déjà été mémorisées — par exemple, dans une mise en œuvre suivant une stratégie de réécriture basée sur un tri topologique, lorsqu'une contrainte de types sur une racine d'une composante fortement connexe provient de l'intérieur de la composante — il est nécessaire de décider d'utiliser soit l'une des règles de simplification [R'-Merge'$_\Psi$] ou [R'-Merge'$_\Phi$], soit l'une des règles de décomposition [R'-Blk] ou [R'-Clos]. Une stratégie efficace tentera naturellement d'éviter de vérifier plusieurs fois une valeur avec le même type en appliquant en priorité l'une des règles simplification [R'-Merge'$_\Psi$] ou [R'-Merge'$_\Phi$], selon le système de types choisi. Mais, en présence de variables existentielles, la condition d'application de ces règles n'est pas toujours facile à vérifier :

- dans le cas du système de types simple, il faut vérifier pour toutes les instanciations possibles des variables existentielles que l'anti-unificateur des types attendus est une instance de l'anti-unificateur des hypothèses de type mémorisées ;
- dans le cas du système de types généralisé, il faut vérifier pour toutes les instanciations possibles des variables existentielles que chacun des types attendus est une instance d'une des hypothèses de type mémorisées.

En pratique, on ne peut évidemment pas tester toutes les instanciations possibles des variables existentielles. On utilise alors une même condition d'application pour les deux systèmes de types plus faible mais plus simple à décider :

- si l'hypothèse de type ne contient pas de variables de type existentielles, on effectue un simple test d'instanciation, par exemple, si $\alpha \to \alpha$ est une hypothèse de type, on acceptera $\dot{\beta} \to \dot{\beta}$ ou `Int` $\to$ `Int` ;
- si l'hypothèse de type contient des variables existentielles, on exige alors l'égalité, par exemple, si $\alpha \to \dot{\beta}$ est une hypothèse de type, on acceptera le type $\alpha \to \dot{\beta}$ mais on refusera `Int` $\to \dot{\beta}$.

Le test d'égalité utilisé en présence des variables existentielles pourrait sans doute être assoupli, par exemple en exigeant l'égalité uniquement sur les variables existentielles. Cet assouplissement permettrait par exemple d'accepter le type $\texttt{Int} \to \dot{\beta}$, lorsque $\alpha \to \dot{\beta}$ est une hypothèse de type. Cependant, le test proposé n'est pas directement inclus dans la condition d'application des règles [R'-MERGE'$_\Psi$] et [R'-MERGE'$_\Phi$]. En effet, le mécanisme de substitution utilisé par la règle [R'-UNIF] permet d'instancier une variable existentielle avec une variable universelle et il suffit pour cela de considérer la substitution $\dot{\theta} \equiv \{\dot{\beta} \mapsto \alpha\}$ pour remarquer que $\dot{\theta}(\texttt{Int} \to \dot{\beta})$ n'est pas une instance de $\dot{\theta}(\alpha \to \dot{\beta})$. Néanmoins, les variables universelles ayant été initialement introduites pour représenter le type vide produit notamment par anti-unification, une telle substitution signifie implicitement $\{\dot{\beta} \mapsto \forall\alpha.\,\alpha\}$ et l'hypothèse de type est implicitement $\forall\alpha.\,\alpha \to \dot{\beta}$. On peut alors imaginer de renommer une des variables lors de la substitution, mais le travail de formalisation de cet argument n'a pas encore été effectué.

Une autre difficulté liée aux conditions d'application des règles de simplification [R'-MERGE'$_\Psi$] et [R'-MERGE'$_\Phi$] est le cas où le test d'instanciation est vrai pour certaines substitutions et faux pour d'autres. Dans ce cas, pour ne pas prendre le risque d'appliquer inutilement une des règles de décomposition [R'-BLK] et [R'-CLOS], une stratégie efficace retardera le plus possible le choix entre les règles de simplification et les règles de décomposition lorsque le type attendu ou l'une des hypothèses contiendra des variables existentielles et, si en fin de compte il est impossible de faire autrement, l'algorithme appliquera l'une des règles de décomposition. On dira alors que l'on force une nouvelle vérification de la valeur partagée. Pour ne pas prendre le risque de boucler, l'algorithme bornera le nombre de nouvelles vérifications forcées de chaque valeur partagée.

De manière plus générale, le prototype décrit au chapitre 4 bornera le nombre d'application d'une des règles de décomposition pour lesquelles le nombre de constructeurs de types apparaissant dans les hypothèses de types ne décroît pas explicitement, c'est-à-dire lors d'une nouvelle vérification forcée, mais aussi en présence de certains types cycliques.

Cette stratégie de réécriture bornée permet de garantir la terminaison, mais oblige à renoncer à la propriété de semi-complétude. Pour diminuer le nombre de programmes rejetés à tort, le prototype tente, lorsque la borne maximale est atteinte, d'instancier les variables existentielles restantes avec des types permettant de valider la contrainte. Par exemple, en reprenant la suite infinie de réécritures décrite à la section 3.4.2 pour le tas :

$$\mu \equiv \{\ell_0 \mapsto \langle\lambda().\,f\ x, \{f : \ell_0\,;\, x : 0\}\rangle\}$$

où le type associé à $\lambda y.\,f\ x$ est $\forall\alpha\beta\gamma.\,(\{f : \alpha \to \beta\,;\, x : \alpha\} \Rightarrow \gamma \to \beta)$.

$$\begin{array}{rcl}
 & & (\mu, \emptyset).\,(\ell_0 : \{\texttt{Unit} \to \texttt{Int}\}) \\
\text{[R'-CLOS][R'-UNIF]} & \gg & \exists\dot{\alpha}.\,(\mu, \{\ell_0 : \{\texttt{Unit} \to \texttt{Int}\}\}).\,(\ell_0 : \{\dot{\alpha} \to \texttt{Int}\} \wedge 0 : \{\dot{\alpha}\}) \\
\text{[R'-CLOS][R'-UNIF]} & \gg & \exists\dot{\alpha}\dot{\beta}\dot{\gamma}.\,(\mu, \{\ell_0 : \{\texttt{Unit} \to \texttt{Int}\,;\, \dot{\alpha} \to \texttt{Int}\}). \\
 & & \qquad (\ell_0 : \{\dot{\beta} \to \texttt{Int}; \dot{\gamma} \to \texttt{Int}\} \wedge 0 : \{\dot{\alpha}\} \wedge 0 : \{\dot{\beta}\,;\, \dot{\gamma}\}) \\
 & \gg & \ldots
\end{array}$$

Dans cet exemple, une fois la borne maximale atteinte, le prototype unifiera les types attendus pour ℓ_0 avec les hypothèses de type déjà mémorisées pour cette adresse-mémoire.

Les variables existentielles restantes seront alors instanciées avec `Unit` et la vérification réussira.

En pratique, la valeur actuelle de la borne maximale dans le prototype est 0, il reste à voir à l'usage quelles sont parmi les valeurs contenant des cycles et des fermetures nécessitant l'explicitation d'inconnues de types, celles qui sont encore rejetées à tort.

Chapitre 4

Mise en œuvre dans le compilateur Objective Caml

L'algorithme de vérification formalisé au chapitre 3 a été appliqué au langage Objective Caml et ce chapitre décrit cette réalisation. Elle prend la forme de deux bibliothèques d'introspection, l'une pour les graphes mémoire et l'autre pour les types, et d'une fonction réalisant le test de compatibilité d'un graphe mémoire avec un type. Plus précisement, la fonction de vérification de compatibilité d'un graphe mémoire a été écrite directement en OCaml, et pour cela il a été nécessaire d'utiliser des mécanismes d'instropection permettant :

- de manipuler explicitement la représentation mémoire d'une valeur ;
- de détecter efficacement les valeurs allouées qui sont effectivement partagées ;
- de manipuler les valeurs $\texttt{ty}(\tau)$ utilisées pour représenter explicitement le type attendu pour une valeur désérialisée (voir section 2.4) ;
- et, pour permettre la désérialisation de valeurs fonctionnelles, de conserver pour chaque pointeur de code le schéma de type de la fonction dont il est isse (voir section 3.3.1).

Graphe mémoire et partage La bibliothèque de base du compilateur OCaml contient déjà un module *Obj* qui permet de manipuler directement la représentation mémoire d'une valeur. Nous avons étendu ce module avec un mécanisme permettant d'associer à chaque valeur allouée des données de partage, telles le nombre de références sur la valeur ou la liste des types déjà vérifiés. Actuellement, ces données de partage sont stockées dans une table d'associations indexée par l'adresse de la valeur dans le tas. Pour cela, les valeurs manipulées doivent être allouées dans le tas *majeur*, et la table doit être reconstruite après chaque phase de compaction du GC. La section 4.1.1 décrit l'interface de cette bibliothèque, appelée *Sobj*, et détaille la représentation mémoire utilisée par le compilateur OCaml.

Représentation dynamique des types Il existe déjà plusieurs extensions de la syntaxe d'OCaml réalisées à l'aide de CamlP4 [De Rauglaudre, 2003] qui permettent d'ajouter la construction $\texttt{ty}(\tau)$. Néanmoins, la représentation dynamique des types devant être compatible avec celle des schémas de type associés à chaque pointeur de code, nous avons choisi de réaliser cette extension de syntaxe en modifiant directement le compilateur OCaml. Les mécanismes d'inférence de type utilisés par notre algorithme de vérification étant nettement plus simples que les mécanismes utilisés par le typeur d'OCaml, la représentation choisie pour les valeurs de type à l'exécution est une simplification de celle utilisée par le compilateur. La section 4.1.2 détaille l'interface du module *Ty* permettant de manipuler la représentation dynamique des types. Le prototype actuel ne prend pas encore en compte les foncteurs applicatifs [Leroy, 1995].

Type des pointeurs de code Conserver à l'exécution le schéma de type de la fonction dont est issu chaque pointeur de code d'un programme demande une modification plus conséquente du compilateur OCaml. En effet, le calcul des types statiques attendus pour l'environnement d'une fermeture doit être effectué au moment de la phase d'introduction des fermetures (voir par exemple Minamide *et al.* [1996]) ; or dans la version actuelle

du compilateur OCaml l'information de type est effacée avant cette étape. Aussi, le prototype écrit pendant cette thèse conserve l'information de typage uniquement pour la version code octet du compilateur et dans le cas de fonctions définies à l'extérieur d'un foncteur. Un travail plus général pourrait réutiliser le travail effectué par Montelatici [2007] pour compiler OCaml vers le code octet typé CLR de dotNet [Microsoft]. La section 4.1.3 revient plus en détail sur la représentation et la construction de la table d'associations entre un pointeur de code et son schéma de type.

Fonction(s) de vérification Les sections 4.2 et 4.3 décrivent la mise en œuvre de la fonction de test de compatibilité d'un graphe mémoire avec un type en détaillant quelques extraits de code. Cette description se concentre dans un premier temps sur une version ignorant les fermetures, puis dans un second temps sur une version complète. La section 4.4 dresse finalement le bilan du prototype réalisé en comparant d'un point de vue pratique le mécanisme de (dé)sérialisation sûre obtenu à ceux déjà existants et décrits au chapitre 1.

4.1 Introspection en OCaml

Cette section décrit les modifications apportées au compilateur OCaml pour pouvoir réaliser l'algorithme de vérification de type. Ces modifications se veulent le moins intrusives possible dans le code du compilateur. En particulier elles sont conçues pour de ne pas ralentir l'exécution d'un programme n'utilisant pas les fonctions de désérialisation sûre.

4.1.1 Parcours du graphe mémoire et informations de partage

Interface du module Sobj

Information de partage Le type $\alpha\ t$ représente une valeur OCaml associée à des informations de partage de type α.

type $\alpha\ t$

Les informations de partage peuvent être ajoutées à une valeur OCaml quelconque à l'aide la fonction *repr*. Cette fonction reçoit en argument une fonction permettant d'initialiser les informations de partage à partir du nombre de références sur une valeur partagée et de son numéro d'ordre dans un parcours en profondeur d'abord.

type $\alpha\ init\ =\ depth : int \rightarrow\ refs : int \rightarrow\ \alpha$
val $repr :\ \alpha\ init\ \rightarrow\ \beta\ \rightarrow\ \alpha\ t$

Ces mêmes informations de typage peuvent être directement produites par les fonctions de désérialisation non-sûre *from_channel* et *from_string*.

val $from\_channel :\ \alpha\ init\ \rightarrow\ in\_channel\ \rightarrow\ \alpha\ t$
val $from\_string :\ \alpha\ init\ \rightarrow\ string \rightarrow\ int \rightarrow\ \alpha\ t$

Récriproquement la fonction *obj* permet d'effacer les informations de partage. Le type de retour de cette fonction est arbitraire. Cette fonction permet de tromper le système de type.

val $obj : \alpha\ t \rightarrow \beta$

Le type *data* permet de récupérer les données de partage associées à une valeur.

```
type α data = {
    sh_obj : α t; (* La valeur partagée *)
    sh_refs : int; (* Le nombre de références sur la valeur partagée (minimum 2) *)
    sh_depth : int; (* La position du bloc dans un parcours en profondeur d'abord *)
    sh_data : α; (* Les données spécifiques allouées par la fonction d'initialisation *)
  }
val data : α t → α data option
```

Parcours du graphe mémoire La fonction *context* permet de rechercher dans le graphe mémoire les valeurs partagées accessibles directement depuis une valeur quelconque. Elle renvoie la liste des données de partage associées aux blocs trouvés.

val $context : \alpha\ t \rightarrow \alpha\ data\ list$

La fonction *successors* permet de rechercher la liste des blocs partagés accessibles directement depuis un bloc partagé.

val $successors : \alpha\ data \rightarrow \alpha\ data\ list$

La fonction *close_successors* permet de rechercher dans le graphe mémoire toutes les valeurs partagées accessibles directement et indirectement depuis une valeur quelquonque.

val $close\_successors : \alpha\ t \rightarrow \alpha\ data\ list$

Ces fonctions seront utiles pour effectuer le tri topologique du graphe mémoire.

Représentation uniforme des données La représentation uniforme des données choisie par le compilateur OCaml permet de distinguer les valeurs immédiates et les valeurs allouées. Cette information est stockée dans le dernier bit du mot-mémoire représentant la valeur. Les fonctions *is_int* et *is_block* permettent de discriminer entre les valeurs immédiates et les valeurs allouées.

val $is\_int : \alpha\ t \rightarrow bool$
val $is\_block : \alpha\ t \rightarrow bool$

Le premier mot mémoire des valeurs allouées dans le tas d'OCaml est un entête comprenant notamment la taille en mots de la zone mémoire, et un *tag* indiquant la nature des données allouées. Les fonctions *tag* et *size* permettent de récupérer ces informations.

val *size* : $\alpha\ t \rightarrow int$
val *tag* : $\alpha\ t \rightarrow int$

Si l'argument de la fonction *tag* est un entier, la constante *int_tag* est renvoyée. Si c'est une adresse mémoire en dehors du tas d'OCaml, la constante *out_of_heap_tag* est renvoyée.

val *int_tag* : *int*
val *out_of_heap_tag* : *int*

Les tags inférieurs à la constante *generic_tag_max* servent à représenter les blocs génériques : constructeurs non-constants d'un type somme, et dans le cas particulier du tag 0 : enregistrement, tableau, etc.

val *generic_tag_max* : *tag*

Les éléments d'un bloc générique peuvent être extraits à l'aide de la fonction *fields*. La fonction *field* permet d'en extraire un élément particulier.

val *fields* : $\alpha\ t \rightarrow \alpha\ t\ list$
val *field* : $\alpha\ t \rightarrow int \rightarrow \alpha\ t$

Fermeture Le tag *closure_tag* sert à distinguer les blocs représentant les fermetures. Le premier élément du bloc est le pointeur de code et les éléments suivants constituent l'environnement de la fermeture.

val *closure_tag* : *int*

Les fonctions mutuellement récursives partagent une fermeture commune de tag *Closure*. Les premiers éléments du bloc sont les pointeurs de code de chacune des fonctions séparés par des pseudo-entête de tag *infix_tag*. Les éléments suivants constituent l'environnement commun aux fonctions mutuellement récursives. Les pseudo-entête permettent de représenter une fonction appartenant à un ensemble de fonctions mutuellement récursives comme un pointeur à l'intérieur du bloc représentant la fermeture commune.

val *infix_tag* : *int*

Les pseudo-entête contiennent leur position dans la fermeture à laquelle ils appartiennent. Aussi, la fonction *closure_from_infix* permet de récupérer cette position et la fermeture englobante.

val *closure_from_infix* : $\alpha\ t \rightarrow \alpha\ t \times int$

La fonction *closure_env* calcule la liste des éléments de l'environnement d'une fermeture. Si la valeur reçue en argument n'est pas un bloc de tag *closure_tag*, l'exception *Invalid_arg* est renvoyée. Cette fonction prend en compte les fermetures représentant un ensemble de fonctions mutuellement récursives.

val *closure_env* : $\alpha\ t \rightarrow \alpha\ t\ list$

La fonction *closure_type* récupère le type statique associé au pointeur de code d'une fermeture ou d'une fermeture récursive (voir section 4.1.3). Si la valeur reçue en argument n'est pas un bloc de tag *closure_tag*, l'exception *Invalid_arg* est renvoyée.

val *closure_type* : $\alpha\ t \rightarrow Ty.scheme$

Bloc de base Le contenu des blocs dont le tag est supérieur à la constante *no_scan_tag* ne suit pas le principe de représentation uniforme des données. Certains des tags supérieurs à cette constante servent à représenter les valeurs de types prédéfinis, tel que les chaînes de caractères et les flottants. Dans le prototype, les valeurs appartenant à ces blocs de base seront vérifiées en suivant des principes similaires aux principes de vérification des entiers.

val *no_scan_tag* : *int*
val *string_tag* : *int*
val *double_tag* : *int*

Parmi les blocs de base on distingue aussi les séquences connexes de flottants permettant d'optimiser la représentation des tableaux de flottants ou les enregistrements dont tous les champs sont de type flottant.

val *double_array_tag* : *int*

Les derniers cas des blocs de base sont les entiers 32 ou 64 bits. Ils sont représentés par un bloc de tag *custom_tag*.

val *custom_tag* : *int*

type *custom_tag*
val *custom_tag* : $\alpha\ t \rightarrow custom\_tag$

val *custom_nativeint_tag* : *custom_tag*
val *custom_int32_tag* : *custom_tag*
val *custom_int64_tag* : *custom_tag*

val *custom_nativeint_size* : *int*
val *custom_int32_size* : *int*
val *custom_int64_size* : *int*

Autres tags Les constantes *lazy_tag* et *object_tag* servent respectivement à discrimer les blocs représentant les valeurs paresseuses non-évaluées et les objets. Ils ne sont pas considérés dans ce prototype.

val *lazy_tag* : *int*
val *object_tag* : *int*

Les tags *Forward* et *Abstract* sont eux aussi ignorés.

val *forward_tag* : *int*
val *abstract_tag* : *int*

4.1.2 Représentation dynamique des types

4.1.2.1 Interface du module Ty

Cette section décrit les parties de l'interface du module *Ty* utile à la réalisation de l'algorithme de vérification de compatibilité d'un graphe mémoire.

Extension de syntaxe La syntaxe des expressions est étendue avec une construction [^ *ty* ^] représentant une expression de type. Pour typer ces expressions, l'ensemble des types prédéfinis est étendu avec un type α *tyrepr*. Ainsi, les valeurs de type *ty tyrepr* sont les valeurs [^ *ty'* ^] telles que $gen(ty) \preccurlyeq ty'$ (voir section 2.4).

Expressions de type Pour manipuler ces expressions de type, il est nécessaire de les projeter vers le type non paramétré *expression* à l'aide de la méthode *dump*.

type *expression*
val *dump* : α *tyrepr* $\rightarrow$ *expression*

Les types abstraits *declaration* et *scheme* représentent respectivement des définitions de types et de schémas de types. Le type *description* définit l'algèbre des types que l'extension de syntaxe [^ *ty'* ^] est capable de représenter dans la version actuelle du prototype.

type *declaration*
type *scheme*
type *description* =

Types prédéfinis

| *Tunit* | *Tbool* | *Tint* | *Tnativeint* | *Tint32* | *Tint64*
| *Tchar* | *Tstring* | *Tfloat* | *Texn*
| *Tarray* of *expression* | *Tlist* of *expression* | *Toption* of *expression*
| *Tlazy* of *expression* | *Ttyrepr* of *expression*
| *Tformat6* of *expression* × *expression* × *expression* ×
expression × *expression* × *expression*
| *Ttuple* of *expression list*
| *Tarrow* of *string* × *expression* × *expression*

Constructeur de type

| *Tconstr* of *declaration* × *expression list*

Variant polymorphe

| *Tvariant* of *bool* × (*string* × *int* × *expression option*) *list*

Définition de type

| *Tsum* of *string list* × (*string* × *expression list*) *list*
| *Tdoublerecord* of (*string* × *bool*) *list*
| *Trecord* of (*string* × *bool* × *scheme*) *list*

Variable de type

| *Tvar*

Type abstrait

| *Tabstract*

La méthode *desc* permet de récupérer la description d'une expression de type.

val *desc* : *expression* → *description*

La fonction *freevars* calcule les variables libres d'une expression de type.

val *freevars* : *expression* → *expression list*

Alias de type et type de donnée La fonction *resolve_alias* permet de remplacer les contructeurs de type correspondant à un alias de type par leur définition. Cette opération de remplacement est récursive et s'effectue en profondeur dans l'expression de type. Autrement dit, aucun des constructeurs de type apparaissant dans l'expression renvoyée ne correspond à un alias de type.

val *resolve_alias* : *expression* → *expression*

La méthode *unfold* remplace l'éventuel constructeur de type de tête de son argument par sa définition. Ce remplacement rend possible à l'éxécution l'analyse de la définition effective d'un type abstrait statiquement par la signature d'un module.

val *unfold* : *expression* → *expression*

Unification La méthode *equal* permet de décider de l'égalité de deux types, y compris en présence d'expressions de type cycliques. L'égalité entre variables de type est l'égalité physique.

val *equal* : *expression* → *expression* → *bool*

La méthode *unify* réalise une opération d'unification destructrice. Elle prend en compte les expressions de type cycliques.

val *unify* : *expression* → *expression* → *bool*

La fonction *try_unify* tente de réaliser l'unification de ses paramètres. Si l'unification réussit, la substitution par effet de bord est effectuée. Sinon, les variables de type ne sont pas modifiées.

val *try_unify* : *Ty.expression* → *Ty.expression* → *bool*

Schéma de type La méthode *open_scheme* permet d'instancier un schéma de type. Les variables de type libres dans le schéma de type sont conservées ; les variables de types liées étant instanciées en de nouvelles variables.

val *open_scheme* : *scheme* → *expression list* × *expression*

4.1.2.2 Interface du module TyHelpers

Le module *TyHelpers* définit les opérations sur les types qui sont spécifiques à l'algorithme de vérification de compatibilité d'un graphe mémoire. Il introduit principalement la notion de variable de type universelle, et les fonctions de test d'instanciation et d'anti-unification qui correspondent.

Variable de type universelle et variable de type existentielle. Les variables de type universelles sont représentées à l'aide de contructeurs de type sans définition manifeste (autrement dit la constante *TAbstract*). Ainsi, ils correspondent à des types vides ; le test de compatibilité d'une valeur avec une variable universelle échoue systématiquement.

val *create_univ_var* : *unit* → *Ty.expression*
val *is_univ_var* : *Ty.expression* → *bool*

Les variables de type existentielles seront simplement représentées par une expression *Ty.create_expr Ty.Tvar*.

val *is_var* : *Ty.expression* → *bool*

La fonction *create_var_list n* permet de créer une liste de n nouvelles variables existentielles.

val *create_var_list* : *int* → *Ty.expression list*

La fonction *generalize vars e* permet de remplacer dans e les variables existentielles *vars* par des variables universelles.

val *generalize* : *Ty.expression list* → *Ty.expression* → *Ty.expression*

Le prédicat *is_monomorphic* est vrai si l'expression de type en paramètre ne contient ni variable universelle ni variable existentielle. Si elle contient des variables universelles, le prédicat est faux. Dans les autres cas, l'exception *Maybe* est levée.

exception *Maybe*
val *is_monomorphic* : *Ty.expression* → *bool*

Tests d'instanciation Le prédicat *is_instance ty1 ty2* est vrai si le type *ty1* est une instance du type *ty2*. Si des variables existentielles rendent le test d'instanciation indécidable, l'exception *Maybe* est levée.

val *is_instance* : *Ty.expression* → *Ty.expression* → *bool*

Le prédicat *is_instance_safe* est une variante du prédicat *is_instance* qui renvoie false lorsque le test est indécidable.

val *is_instance_safe* : *Ty.expression* → *Ty.expression* → *bool*

Le prédicat *fold_is_instance ty tys* renvoie *Some* true si *ty* est une instance d'au moins un des types de la liste *tys*. Elle renvoie *Some* false s'il est possible de décider que *ty* n'est instance d'aucun des types de la liste *tys*. Elle renvoie *None* sinon.

val *fold_is_instance* : *Ty.expression* → *Ty.expression list* → *bool option*

Anti-unification La fonction *antiunif* calcule l'anti-unificateur principal de deux types. Deux appels successifs n'utilisent pas la même mémoire. Si des variables existentielles rendent le calcul impossible, l'exception *Maybe* est levée. Cette fonction prend en compte les expressions de type cycliques.

val *antiunif* : *Ty.expression* → *Ty.expression* → *Ty.expression*

La fonction *size* compte le nombre de constructeurs de type apparaissant dans une expression de type, même en présence de type cyclique.

val *size* : *Ty.expression* → *int*

Prédicat d'homogénéité La fonction *check_constr* vérifie que deux types possèdent le même constructeur de tête.

val *check_constr* : *Ty.expression* → *Ty.expression* → *bool*

La fonction *duplicate_constr ty* construit une nouvelle expression de type ayant le même constructeur de type de tête que *ty* et dont les paramètres sont de nouvelles variables existentielles.

val *duplicate_constr* : *Ty.expression* → *Ty.expression*

4.1.3 Exporter le type des pointeurs de code

Propager l'information de typage Pour conserver à l'exécution le type statique de la fonction dont est issue un pointeur de code, il est nécessaire que le compilateur conserve les informations de type jusqu'à la phase d'introduction des fermetures. Propager cette information dans un cadre général demande d'importantes modifications de la version actuelle du compilateur OCaml. Le prototype réalisé durant cette thèse effectue des

modifications *a minima* permettant d'exporter cette information de type uniquement dans le cas de fonction simple et uniquement pour la version du compilateur produisant du code octet. Pour cela, le lambda-code est étendu avec un cas permettant, lorsque cela est possible simplement, de conserver le type d'une variable après la phase de traduction de l'arbre de syntaxe typé en lamba-code :

| *Ltvar* of *Ident.t* × *Types.type_expr*

De la même manière le cas *Lfunction* est étendu pour conserver, lorsque cela est possible simplement, le type de la fonction originale :

| *Lfunction* of *function_kind* × *Ident.t list* × *lambda* × *Types.type_expr option*

Lors de la phase d'introduction des fermetures — c'est-à-dire dans la version actuelle du compilateur lors de la compilation du lambda-code en code octet — si l'information de types est disponible pour l'ensemble des variables libres capturées, il est possible de calculer un type statique à associer au pointeur de code de la fermeture générée. Malheureusement, cette approche minimaliste fonctionne uniquement lorsque l'environnement de typage ne comporte pas de schéma de type ; la position exacte des quantifications universelles n'étant pas conservée.

Construire la table d'association La seconde difficulté à résoudre est l'association à l'exécution entre un pointeur de code et son type statique. Au lieu de construire une table d'association globale, le prototype stocke le type associé à un pointeur de code dans une variable globale distincte et insère une pseudo-instruction référençant cette variable globale juste avant l'instruction pointée.

4.2 Algorithme de vérification sans fermeture

Cette section décrit quelques extraits de code utiles à la mise en œuvre de l'algorithme présenté au chapitre 3 en l'absence de valeurs fonctionnelles. Une première étape repose sur le système de type généralisé ; elle réalise un parcours en profondeur du graphe mémoire en ignorant le partage et elle échoue en présence de valeurs modifiables. Une deuxième étape optimise le parcours des graphes sans cycle à l'aide d'un tri topologique basé sur un compteur de référence. Une troisième étape ajoute la gestion des valeurs modifiables. Une quatrième étape permet la vérification des graphes contenant des cycles en explicitant les composantes fortement connexes. Une cinquième et dernière étape restreint l'ensemble des valeurs acceptables à celles typables dans le système de type original.

Dans les extraits de code ci-dessous, quelques annotations de types ont été ajoutées pour faciliter la lecture.

Dans les extraits de code ci-dessous, quelques annotations de types ont été ajoutées pour faciliter la lecture.

Module Check

1. Le module *Check* décrit une première étape dans la mise en œuvre de l'algorithme de vérification. Il ne considère que les valeurs entières, les blocs de base et les blocs génériques ; il ignore les fermetures et les valeurs modifiables. Le système de types sous-jacent est le système de types généralisé, ce qui permet d'ignorer le partage et d'utiliser une stratégie de réécriture basée sur un simple parcours en profondeur du graphe-mémoire. En présence de cycle, cette première version ne termine pas.

2. Le prototype dépend des modules *Sobj* et *Ty* décrits aux sections 4.1.1 et 4.1.2.

```
open Sobj
open Ty
```

3. Dans le système de réécriture, les cas d'échec sont représentés par la contrainte `False`. Dans ce prototype, ils seront représentés par l'exception *TypeError*.

```
exception TypeError
let fail () = raise TypeError
```

4. La fonction *check_int* réalise les règles de réécriture [R'-Int-True] et [R'-Int-False]lorsque le type attendu n'est pas une variable existentielle. Elle est définie par cas sur la structure du type attendu.

```
let rec check_int (i : int) (ty : Ty.expression) =
  match Ty.desc (Ty.unfold ty) with
  | Tint → ()
  | Tchar → if i < 0 ∨ i > 255 then fail ()
  | Tunit | Tlist _ | Toption _ → if i ≢ 0 then fail ()
  | Tbool → if i ≢ 0 ∧ i ≢ 1 then fail ()
  | Tsum (const, _) → if i ≥ List.length const then fail ()
  | Tnativeint | Tint32 | Tint64 | Tstring | Tfloat
  | Tarray _ | Ttuple _ | Tdoublerecord _ | Trecord _
  | Tformat6 _ | Tvariant _ | Tarrow _ | Ttyrepr _ | Texn
  | Tabstract → fail ()
  | Tvar → invalid_arg "Check.check_int"
  | Tlazy ty → check_int i ty
  | Tconstr _ → assert false
```

Dans le langage de type utilisé au chapitre 3, le seul type *a priori* compatible avec un entier négatif ou supérieur à 255 est le type `Int`. Il serait donc possible d'étendre le système de récriture avec la règle ci-dessous :

$$i : \tau \gg \tau = \texttt{Int} \qquad si\ i < 0\ ou\ i > 255 \qquad \text{[R-Int-Inst]}$$

Cette remarque n'est plus valable dans les versions actuelles du compilateur OCaml. Par exemple, le mécanisme de compilation des valeurs paresseuses ne permet pas de distinguer la représentation d'une valeur de type `Int` d'une valeur de type `Lazy(Int)` si celle-ci a déjà été évaluée. Un entier supérieur à 255 peut donc appartenir aux deux types.

5. La fonction *check_basic_block* réalise dans le cas des blocs de base, c'est-à-dire ceux dont le *tag* est supérieur à la constante `No_scan_tag`, des règles de réécriture similaires aux règles [R-Int-True] et [R-Int-False]. Cette fonction ne doit pas être appelée lorsque le type attendu est une variable existentielle. Dans le cas particulier des blocs de base représentant une séquence de flottants (cas *Tdoublerecord* et *Tarray Tfloat*), la fonction *check_basic_block* renvoie une liste représentant le caractère modifiable ou non des éléments du bloc.

```
let rec check_basic_block (obj : α Sobj.t) (ty : Ty.expression) : bool list =
  let tag = Sobj.tag obj
  and size = Sobj.size obj in
  match Ty.desc (Ty.unfold ty) with
  | Tstring → if tag ≢ Obj.string_tag then fail () else []
  | Tfloat → if tag ≢ Obj.double_tag then fail () else []
  | Tdoublerecord l →
      if tag ≢ Obj.double_array_tag then fail ();
      if ¬ (List.length l = size) then fail ();
      List.map snd l
  | Tarray ty when Ty.desc ty = Tfloat →
      if tag ≢ Obj.double_array_tag then fail ();
      TyHelpers.create_list size true
  | Tnativeint →
      check_custom obj Sobj.custom_nativeint_tag Sobj.custom_nativeint_size
  | Tint32 → check_custom obj Sobj.custom_int32_tag Sobj.custom_int32_size
  | Tint64 → check_custom obj Sobj.custom_int64_tag Sobj.custom_int64_size
  | Tchar | Tint | Tbool | Tunit | Toption _ | Tlist _
  | Trecord _ | Tsum _ | Tvariant _ | Tarrow _ | Ttuple _
  | Ttyrepr _ | Texn | Tabstract | Tformat6 _ | Tarray _ → fail ()
  | Tlazy ty → check_basic_block obj ty
  | Tvar → invalid_arg "Check.check_basic_block"
  | Tconstr _ → assert false

and check_custom obj custom_tag size =
  if Sobj.tag obj ≢ Obj.custom_tag then fail ();
  if Sobj.custom_tag obj ≢ custom_tag then fail ();
  if Sobj.size obj ≢ size then fail ();
  []
```

6. La fonction *get_generic_block_tys* calcule à partir du type attendu pour un bloc les types attendus pour les éléments de ce bloc, ainsi que leur caractère modifiable ou non ; si le type n'est pas acceptable pour un bloc, l'exception *TypeError* est levée. Cette fonction sera utile pour réaliser les règles de réécriture [R'-∇-True] et [R'-∇-False].

```
let immutable tys = TyHelpers.create_list (List.length tys) false, tys
```

```
let rec get_generic_block_tys tag size ty : bool list × Ty.expression list =
  match Ty.desc (Ty.unfold ty) with
  | Toption ty when tag = 0 ∧ size = 1 → immutable [ty]
  | Tlist ty' when tag = 0 ∧ size = 2 → immutable [ty'; ty]
  | Ttuple tys when tag = 0 ∧ size = List.length tys → immutable tys
  | Tsum (_, nconst) →
      (try let tys = snd (List.nth nconst tag) in
           if size ≠ List.length tys then fail();
           immutable (List.map Ty.resolve_alias tys)
       with Failure "nth" → fail ())
  | Trecord ntys when tag = 0 ∧ size = List.length ntys →
      let get_field (_, mut, sty) =
        let (vars, expr) = Ty.open_scheme sty in
        (mut, Ty.resolve_alias (TyHelpers.generalize vars expr)) in
      List.split (List.map get_field ntys)
  | Tarray ty when Ty.desc ty ≠ Tfloat ∧ tag = 0 →
      (TyHelpers.create_list size true, TyHelpers.create_list size ty)
  | Tlazy ty → get_generic_block_tys tag size ty
  | Tunit | Tint | Tbool | Tchar | Tstring | Tfloat | Tformat6 _
  | Tdoublerecord _ | Tnativeint | Tint32 | Tint64 | Tabstract
  | Texn | Tarrow _ | Ttyrepr _ | Tvariant _ → fail ()
  | Tvar → invalid_arg "Check.get_generic_block_tys"
  | Tconstr _ → assert false
  | Toption _ | Tlist _ | Ttuple _ | Trecord _ | Tarray _ → fail ()
```

L'utilisation de la fonction *Ty.resolve_alias* (voir 4.1.2) dans le cas des types sommes et dans celui des types enregistrements permet de supposer dans le reste le l'algorithme que des les expressions de types manipulées ne contiennent pas d'alias de types. Cela simplifiera notamment les tests d'égalité ou les calculs d'anti-unificateur.

7. La fonction *get_generic_block_tyss* permet d'itérer la fonction *get_generic_block_tys* sur un ensemble de type attendus. Elle échoue si le caractère modifiable d'un champ n'est pas le même pour tous les types attendus.

```
let rec get_generic_block_tyss tag size tys : bool list × Ty.expression list list =
  match tys with
  | [] → assert false
  | [ty] →
      let (muts, tys) = get_generic_block_tys tag size ty in
      (muts, List.map (fun ty → [ty]) tys)
  | ty :: tys →
      let (muts, tyss) = get_generic_block_tyss tag size tys in
      let (muts', tys') = get_generic_block_tys tag size ty in
      if muts ≠ muts' then fail ();
      (muts, List.map2 (fun ty tys → ty :: tys) tys' tyss)
```

8. Le parcours de vérification d'une valeur se réalise naturellement à l'aide d'un ensemble de fonctions mutuellement récursives. Néanmoins pour présenter notre prototype par étape, nous le décrivons ici à l'aide d'une classe. Les versions successives de l'algorithme seront alors définies par héritage et redéfinition de méthode.
La classe *check* définie ici réalise le parcours de vérification. Elle est paramétrée par le type des données associées aux blocs partagés. Cette première version ignorant le partage, aucune contrainte ne porte sur le paramètre de type.

```
class ['data] check = object (check)
```

9. La méthode principale de vérification d'une valeur est la méthode *check#sobj*. Elle réalise la stratégie de vérification d'une contrainte $w : \{\tau_{1..n}\}$. Dans ce premier module, chaque contrainte générée est vérifiée immédiatement en appelant la méthode *check#obj*. Autrement, la vérification est un simple parcours récursif en profondeur du graphe mémoire.

```
  method sobj (obj : 'data Sobj.t) (tys : Ty.expression list) : unit =
    check#obj obj tys
```

10. Pour cette première étape, c'est la méthode *check#obj* qui débute réellement la vérification d'une contrainte $w : \{\tau_{1..n}\}$, en déléguant, selon la nature de la valeur w (entier, bloc de base, bloc générique, ...), la vérification à une méthode spécifique.

```
  method obj obj tys =
    if Sobj.is_int obj
    then check#int (Sobj.obj obj : int) tys
    else
      let tag = Sobj.tag obj in
      if tag ≥ Obj.no_scan_tag then ignore (check#basic_block obj tys)
      else if tag = Obj.closure_tag then
        check#closure (Sobj.closure_type obj) (Sobj.closure_env obj) tys
      else if tag = Obj.lazy_tag then check#lazy_ obj tys
      else if tag = Obj.object_tag then check#object_ obj tys
      else if tag = Obj.infix_tag then check#infix obj tys
      else if tag = Obj.forward_tag then check#forward obj tys
      else check#block tag (Sobj.fields obj) tys
```

Ici, la vérification des fermetures simples et récursives échoue systématiquement.

```
  method closure = fail ()
  method infix = fail()
```

Les cas particuliers des objets et des valeurs paresseuses non évaluées ne seront jamais prises en compte dans ce prototype.

```
  method lazy_ = fail ()
  method object_ = fail ()
  method forward = fail ()
```

11. Dans le cas des entiers ou des blocs de base, les méthodes spécifiques *check#int* et *check#basic_bloc* se contentent respectivement d'itérer les fonctions *check_int* (4) et *check_basic_block* (5) pour chacun des types attendus.

```
method int i tys =
  List.iter (check_int i) tys

method basic_block obj tys =
  let muts = List.map (check_basic_block obj) tys in
  check#basic_block_mutability muts
```

12. Dans le cas particulier des blocs de base représentant une séquence de flottants, la méthode *check#basic_block_mutability* vérifie qu'aucun des éléments n'est modifiable. La valeur renvoyée par la méthode est ignorée dans cette première version.

```
method basic_block_mutability (muts : bool list list) : bool list option =
  if List.exists (List.exists (fun x → x)) muts then fail ();
  None
```

Autrement dit, puisque nous sommes dans le cadre du système de types généralisé et sans valeurs modifiables, ces méthodes réalisent en fait une extension aux blocs de base des règles de réécriture [R'-INT$_{\Psi,\Phi}$], et implicitement [R'-$\Diamond_\Phi$].

13. Dans le cas des blocs génériques, la méthode *check#block* calcule d'abord les types attendus pour chacun des éléments du bloc à l'aide de la fonction *get_generic_block_tyss* (7). Elle appelle ensuite récursivement la méthode *check#sobj* sur chacun des éléments du bloc et des types attendus.

```
method block tag fields tys =
  let size = List.length fields in
  let (muts, tyss) = get_generic_block_tyss tag size tys in
  ignore (check#block_mutability (muts, tyss));
  if size ≠ List.length tyss then fail ();
  List.iter2 check#sobj fields tyss
```

La méthode auxiliaire *check#block_mutability* vérifie pour l'instant que les types attendus n'imposent pas à un des éléments du bloc d'avoir un caractère modifiable. La valeur renvoyée est ici ignorée.

```
method block_mutability (muts, _) =
  if List.exists (fun mut → mut) muts then fail ();
  ([] : Ty.expression option list)
```

Autrement dit, toujours dans le cadre du système de type généralisé et sans valeurs modifiables, ces méthodes réalisent les règles de réécriture [R'-BLK], puis [R'-∇-TRUE] ou [R'-∇-FALSE], et implicitement [R'-$\Diamond_\Phi$]. Ces règles de réécriture s'appliquent systématiquement car aucun des types attendus ne contient de variable de type existentielle.

14. La méthode *check#check* est le point d'entrée de la fonction de vérification. Elle commence par remplacer toutes les variables de type libres dans les types attendus par

des constantes représentant le type vide (voir 4.1.2.2) et toutes les *alias* de type par leur définition concrète (voir 4.1.2). Elle appelle ensuite la méthode *check#sobj*.

```
  method check (obj : 'data Sobj.t) (tys : Ty.expression list) : unit =
    let tys = List.map (fun ty → TyHelpers.generalize (freevars ty) ty) tys in
    let tys = List.map Ty.resolve_alias tys in
    check#sobj obj tys
end (* class check *)
```

Module Sharings

15. Le module *Sharings* étend la première version de l'algorithme de vérification décrit par le module *Check* en ajoutant un mécanisme de gestion du partage visant l'efficacité. Pour cela, ce module applique la stratégie de vérification basée sur un tri topologique décrit à la section 3.7.1. Le système de type sous-jacent reste le système de type généralisé.

16. En l'absence de cycle, le tri topologique est effectué paresseusement à l'aide d'un compteur de références :

- à chaque bloc partagé est initialement associé le nombre de référence sur ce bloc ;
- à chaque fois qu'un bloc partagé est rencontré au cours du parcours récursif du graphe mémoire, ce compteur est décrémenté, et :
  - si le compteur n'est pas nul, les types attendus pour ce bloc sont mémorisés et le parcours récursif est interrompu ;
  - si le compteur est nul, l'ensemble des types attendus a été collectés, et le bloc est vérifié immédiatement.

Ainsi, le type α *data* défini ci-dessous représente le type des données associé à chaque bloc partagé et la fonction *init* permet d'initialiser ces données.

```
type α data = {
    mutable remaining_refs : int; (* Nombre de références restantes *)
    mutable tys : Ty.expression list; (* Ensembles des types collectés *)
    data : α;
  }
and α sobj_data = α data Sobj.data
and α sobj = α data Sobj.t

let init ~depth ~refs = {
  remaining_refs = refs;
  tys = [];
  data = ();
}
```

17. En présence de cycle, cette deuxième étape dans la description du prototype termine prématurément en levant l'exception *Unexpected_cycle*. Une gestion plus complète des cycles sera décrite avec le module *Cycle*.

```
exception Unexpected_cycle
```

18. La classe *check* effectuant la stratégie de parcours basé sur un tri topologique hérite de la classe *Check.check* en contraignant le type des données associées aux blocs partagés au type α *data*. Le paramètre de type *'data* reste un paramètre de la classe *check* pour permettre des extensions futures.

```
class ['data] check = object (check) inherit ['data data] Check.check as super
```

19. La méthode *Check.check#sobj* est redéfinie pour permettre un traîtement spécifique des blocs partagés à l'aide de la méthode *check#shared*. Les blocs non partagés restent vérifiés immédiatement en suivant le parcours en profondeur d'abord décrit par la méthode *Check.check#obj*.

```
method sobj obj tys =
  match Sobj.data obj with
  | Some sh → check#shared sh tys
  | None → check#obj obj tys
```

20. La méthode *check#shared* réalise le tri topologique paresseux décrit au point 16. Elle maintient en plus la liste des blocs partagés déjà rencontrés et dont le compteur de références n'est pas encore nul. Cette liste sera utile pour la détection des cycles et leur prise en compte par le module *Cycle*.

```
val mutable postponed_block : 'data sobj_data list = []
method shared sh tys =
  let data = sh.Sobj.sh_data and refs = sh.Sobj.sh_refs in
  data.tys ← tys @ data.tys;
  if data.remaining_refs = refs then postponed_block ← sh :: postponed_block;
  data.remaining_refs ← data.remaining_refs − 1;
  if data.remaining_refs = 0 then
      ( postponed_block ← List.filter ((≢) sh) postponed_block;
        data.tys ← check#simplify data.tys;
        check#obj sh.Sobj.sh_obj data.tys )
```

La méthode annexe *check#simplify* permet de simplifier l'ensemble des types collectés avant de vérifier un bloc partagé. Elle suit le principe des règles de réécriture [R'-MERGE$_\Phi$] et [R'-SIMPL$_\Phi$].

```
method simplify tys = List.fold_right check#insert tys []
method insert ty tys = match tys with
  | [] → [ty]
  | ty′ :: _ when TyHelpers.is_instance_safe ty ty′ → tys
  | ty′ :: tys when TyHelpers.is_instance_safe ty′ ty → check#insert ty tys
  | ty′ :: tys → check#merge_insert ty′ ty tys
method merge_insert ty′ ty tys = ty′ :: check#insert ty tys
```

21. La méthode *Check#check* est redéfinie pour détecter la présence de cycle : si à la fin du parcours du graphe mémoire il reste des blocs dont le compteur de références n'est pas nul, ces blocs participent à des cycles.

```
method check (obj : 'data sobj) (tys : Ty.expression list) : unit =
  super#check obj tys;
  check#schedule_postponed

method schedule_postponed =
  if postponed_block ≠ [] then check#cycle

method cycle = raise Unexpected_cycle
```

L'utilisation explicite des méthodes *cycle* et *schedule_postponed* facilitera l'extension du prototype pour prendre en compte respectivement les cycles puis les fermetures.

```
end (* class check *)
```

Module References

22. Le module *References* permet d'ajouter la gestion des valeurs modifiables au prototype. Il repose sur les principes décrit à la section 3.6. La classe *check* effectue cette extension en héritant simplement de la classe *Sharings.check*.

```
class ['data] check = object (check) inherit ['data] Sharings.check as super
```

23. La méthode *Check.check#block_mutability* (13) est redéfinie pour garantir que le caractère modifiable d'un élément d'un bloc est le même pour tous les types attendus pour ce bloc. De plus, lorsqu'un élément d'un bloc est modifiable, les types attendus doivent être égaux et monomorphes (voir la règle [R'-◊-Ref]).

```
method block_mutability
    ((muts, tyss) : bool list × Ty.expression list list) : Ty.expression option list =
  let check_field mut tys =
    if mut then Some (check#equal_and_monomorphic tys) else None in
  List.map2 check_field muts tyss

method equal_and_monomorphic tys = match tys with
  | [] → assert false
  | ty :: tys →
      if ¬ (List.for_all (Ty.equal ty) tys) then fail ();
      if ¬ (TyHelpers.is_monomorphic ty) then fail ();
      ty
```

La méthode renvoie un couple composé d'une liste de booléens représentant le caractère modifiable des éléments d'un bloc et de la liste des types attendus pour chacun de ces éléments. Cette information sera utile en présence de fermetures.

24. Pour effectuer la même vérification en présence de blocs de base représentant des séquences de flottants, la méthode *Check.check#basic_block_mutability* (12) est elle aussi redéfinie. La valeur renvoyée est ici ignorée.

```
method basic_block_mutability (d : bool list list) : bool list option =
  match d with
  | [] → assert false
  | muts :: mutss → if ¬ (List.for_all ((=) muts) mutss) then fail () else Some muts
end (* class check *)
```

Module Cycle

25. Le module *Cycle* permet la vérification des graphes-mémoire dont les cycles peuvent être vérifiés en un seul passage. Il étend pour cela la stratégie de réécriture fondée sur un tri topologique paresseux en effectuant un tri explicite du sous-graphe restant à parcourir lorsque des cycles ont été découverts. Chaque composante fortement connexe est alors vérifiée en suivant la même stratégie que le graphe-mémoire complet : dans un premier temps parcours topologique paresseux basé sur un compteur de références, puis dans un second temps si des cycles ont été découverts, tri topologique explicite du sous-graphe de la composante restant à parcourir.

26. Une fois le graphe-mémoire trié explicitement, les blocs partagés vont être annotés avec un identifiant représentant la composante fortement connexe à laquelle ils appartiennent. Pour mémoriser l'historique des tris topologiques, cet identifiant sera constitué d'une liste d'entiers. Le premier entier de la liste correspond à l'identifiant unique de la composante fortement connexe issue du premier tri topologique explicite du graphe-mémoire. Le second entier de la liste correspond à l'identifiant unique de la composante fortement connexe issue d'un éventuel tri topologique à l'intérieur de la composante identifiée par le premier entier, et ainsi de suite. Bien que cet identifiant complexe ne soit pas strictement nécessaire pour cette version de l'algorithme, il simplifie la réalisation d'un algorithme pouvant vérifier les graphes-mémoire nécessitant plusieurs parcours d'un cycle (voir l'exemple 3.4.2 et la section **??**). Pour mémoriser cet identifiant, le type *Sharings.data* des données associées aux blocs de partage est étendu avec le type *data* ci-dessous, dans lequel :

- le champ *scc_id* représente l'identifiant de composante fortement connexe ;
- le champ *tag* est utile pour effectuer le tri topologique explicite ;
- le champ *succ* permet de mémoriser la liste des blocs partagés accessibles directement à partir de ce bloc partagé.

```
type α data = {
    mutable scc_id : int list;
    mutable tag : bool;
    mutable succ : α sobj_data list option;
    data : α;
```

```
  }
and α sobj_data = α data Sharings.data Sobj.data
and α sobj = α data Sharings.data Sobj.t
```

27. La fonction *in_scc id sh* permet de tester l'appartenance d'un bloc partagé *sh* à la composante fortement connexe *id* ou à l'une de ses composantes internes.

```
let in_scc id (sh : α sobj_data) : bool =
  let rec prefix xs ys = match xs, ys with
  | [], _ → true
  | _, [] → false
  | x :: xs, y :: ys → x = y ∧ prefix xs ys in
  prefix id sh.Sobj.sh_data.Sharings.data.scc_id
```

La fonction *get_successors* ci-dessous permet de mémoriser les successeurs d'un bloc partagé calculé par la fonction *Sobj.successors* (voir 4.1.1).

```
let get_successors (sh : α sobj_data) : α sobj_data list =
  match sh.Sobj.sh_data.Sharings.data.succ with
  | Some succ → succ
  | None →
      let succ = Sobj.successors sh in
      sh.Sobj.sh_data.Sharings.data.succ ← Some succ;
      succ
```

28. Le tri topologique explicite est effectué à l'aide de l'algorithme de Tarjan [1972]. La mise en œuvre de l'algorithme n'est pas détaillée ici ; elle est dissimulée dans un foncteur paramétré par le module *Shared* ci-dessous. Ce module contient le type des données à trier, les accesseurs correspondants (non détaillés ici) et la fonction de calcul des successeurs. Dans notre cas, si un tri topologique a déjà eu lieu, les successeurs d'un bloc sont restreints aux successeurs inclus dans la même composante fortement connexe lors du tri précédent.

```
module Shared(T : sig type t end) = struct

  type t = T.t sobj_data

  let get_successors sh =
    let get_remainings_refs sh = sh.Sobj.sh_data.Sharings.remaining_refs in
    List.filter
      (fun succ → get_scc_id succ = get_scc_id sh ∧ get_remainings_refs succ > 0)
      (get_successors sh)

end

let tsort (type a) shs =
  let module Sort = Tarjan.Make(Shared(struct type t = a end)) in
  Sort.tsort shs
```

29. La classe *check* vérifie les graphes-mémoire pour lesquels il suffit de parcourir les cycles une seule fois. Elle hérite de la classe *References.check* en contraignant les données associées aux blocs partagés à être de type α *data*. Une fois encore, le paramètre du type *data* reste un paramètre dédié à des extensions futures.

```
class ['data] check = object (check) inherit ['data data] References.check as super
```

30. La méthode *Sharings.check#cycle* (21) est redéfinie. Au lieu d'échouer, elle effectue maintenant le tri topologique explicite du sous-graphe-mémoire non encore vérifié, puis elle vérifie successivement à l'aide de la méthode *scc* chacune des composantes fortement connexes obtenues.

```
method cycle =
  let roots = postponed_block in postponed_block ← [];
  List.iter check#scc (tsort roots)
```

31. Pour vérifier une composante fortement connexe, la méthode *scc* vérifie l'ensemble des racines de la composante — c'est-à-dire les blocs de la composante pour lesquels des types ont déjà été collectés. De la même manière que précédemment les blocs partagés, elle simplifie d'abord l'ensemble des types collectés à l'aide de la méthode *Sharing.check#simplify*, puis elle effectue la vérification avec la méthode *Check.check#obj* (10). Pour pouvoir restreindre cette vérification à la composante fortement connexe actuelle, son identifiant est préalablement mémorisé dans une variable d'instance. La méthode *scc* force également à 0 le nombre de références restant à vérifier pour les racines, permettant dans la suite du parcours de détecter les références arrières d'un bloc de la composante vers une de ses racines.

```
val mutable current_scc = []
method scc (id, shs) =
  current_scc ← id;
  let roots = List.filter (fun sh → sh.Sobj.sh_data.Sharings.tys ≠ []) shs in
  List.iter (fun sh → sh.Sobj.sh_data.Sharings.remaining_refs ← 0) roots;
  let check_root sh =
    let data = sh.Sobj.sh_data in
    data.Sharings.tys ← check#simplify data.Sharings.tys;
    check#obj sh.Sobj.sh_obj data.Sharings.tys in
  List.iter check_root roots;
  check#schedule_postponed
```

32. La méthode *Sharings.check#shared* (20) est redéfinie pour restreindre la vérification à la composante actuelle. Lorsqu'au cours du parcours une référence à une des racines est rencontrée, la vérification est déléguée à la méthode *back_ref*. Lorsqu'une référence à un bloc d'une autre composante est rencontrée, les types attendus pour ce bloc sont simplement mémorisés. Il n'est plus nécessaire de décrémenter le compteur de références, un tri topologique ayant déjà été effectué. Dans tous les autres cas, la stratégie n'est pas modifiée.

```
  method shared sh tys =
    let data = sh.Sobj.sh_data in
    if data.Sharings.remaining_refs = 0 then
      check#back_ref sh tys
    else if data.Sharings.data.scc_id ≠ current_scc then
      data.Sharings.tys ← tys @ data.Sharings.tys
    else
      super#shared sh tys
```

33. À cette étape de description du prototype — c'est-à-dire dans le système de type généralisé, en l'absence de variable existentielle et en refusant les graphes contenant des composantes nécessitant d'être vérifiées plusieurs fois — la vérification d'une référence arrière est un simple test d'instanciation : on vérifie que chacun des types attendus est une instance d'un des types mémorisés pour la racine.

```
  method back_ref sh tys =
    let data = sh.Sobj.sh_data in
    let tys = check#simplify tys in
    data.Sharings.tys ← check#simplify data.Sharings.tys;
    let test_instantiation ty =
      List.exists (TyHelpers.is_instance_safe ty) data.Sharings.tys in
    let tys = List.filter (fun ty → ¬ (test_instantiation ty)) tys in
    if tys ≠ [] then check#recheck sh tys
```

34. Si l'un des types attendus n'est pas une instance d'un des types mémorisés, la vérification échoue.

```
  method recheck sh tys = fail ()

end (* class check *)
```

Module Restriction

35. Pour terminer la description du prototype en l'absence de valeur fonctionnelle, on s'intéresse ici au module *Restriction* qui restreint l'ensemble des graphes-mémoire acceptés à ceux typables dans le système de type original. La section 3.4.1 a montré que cette restriction permet en l'absence de type cyclique d'autoriser les vérifications multiples d'un cycle sans perdre la propriété de terminaison. Cette possibilité de vérification multiple est nécessaire pour obtenir la propriéte de complétude en présence de récursion polymorphe (voir l'exemple 3.4.2). En présence de type cyclique, l'argument de décroissance utilisé à la section 3.4.1 pour montrer la terminaison de l'algorithme n'est plus valable. Pour garantir la terminaison de ce prototype, nous bornons le nombre de vérifications d'un bloc partagé ne faisant pas décroître la taille du type attendu par la valeur initiale du champ *recheck_count*.

```
type α data = {
    mutable recheck_count : int;
    data : α;
  }
and α sobj_data = α data Cycle.data Sharings.data Sobj.data
and α sobj = α data Cycle.data Sharings.data Sobj.t

class ['data] check = object (check) inherit ['data data] Cycle.check as super
```

36. La méthode *Sharings.check#merge_insert* — méthode auxiliaire de la méthode *Sharings.check#simplify* (20) — est redéfinie en suivant le principe des règles de réécriture [R'-MERGE$_\Psi$] et [R'-SIMPL$_\Psi$]. En l'absence de variable existentielle, l'ensemble des types attendus pour un bloc partagé peut être systématiquement anti-unifié et réduit à un singleton.

```
method merge_insert ty' ty tys = check#insert (TyHelpers.antiunif ty' ty) tys
```

37. Pour ne pas interférer avec le mécanisme de tri topologique paresseux, la deuxième vérification d'un bloc partagé ne peut avoir lieu avant que ne soit terminé le premier parcours de vérification de la composante fortement connexe à la quelle appartient le bloc. La méthode *Cycle.check#recheck* (34) est redéfinie pour mémoriser dans la variable d'instance *postponed_recheck* l'ensemble des blocs qu'il faudra vérifier à nouveau.

```
val mutable postponed_recheck = []
method recheck sh tys =
  let rec insert (sh, tys as c) l = match l with
  | [] → [c]
  | (sh', tys') :: l when sh' ≡ sh → (sh, tys' @ tys) :: l
  | c' :: l → c' :: insert c l in
  postponed_recheck ← insert (sh, tys) postponed_recheck
```

Ainsi, un bloc de partage n'apparaît qu'une seule fois dans la liste *postponed_recheck*.

38. La méthode *Cycle.check#scc* (31) est redéfinie pour amorcer la vérification des blocs partagées pour lesquelles plusieurs passage sont nécessaires.

```
method scc (id, shs) =
  super#scc (id, shs);
  check#schedule_recheck id
```

39. La méthode *schedule_recheck* applique une stratégie de réécriture fondée sur un parcours en largeur des blocs partagés et un parcours en profondeur des blocs non partagés. Dans tous les cas, le parcours est restreint à la composante fortement connexe actuelle, y compris ses éventuelles composantes internes (voir 26 et 27).

```
method schedule_recheck id =
  let shs, upper_shs =
    List.partition (fun (sh, _) → Cycle.in_scc id sh) postponed_recheck in
  postponed_recheck ← upper_shs;
  if shs ≠ [] then (List.iter check#process_recheck shs; check#schedule_recheck id)
```

40. Lorsqu'un bloc partagé a déjà été vérifié, le champ *Sharings.remaining_refs* des données de partage qui lui sont associées vaut 0. Ainsi, à partir du deuxième parcours de vérification, la méthode *Cycle.check#shared* (32) délègue systématiquement la vérification des blocs déjà vérifiés une première fois à la méthode *Sharings.check#back_ref* (33). Si nécessaire, cette méthode délèguera ensuite, via les méthodes *check#recheck* (37) et *check#schedule_recheck* (39), les vérifications supplémentaires du bloc à la méthode *process_recheck*.

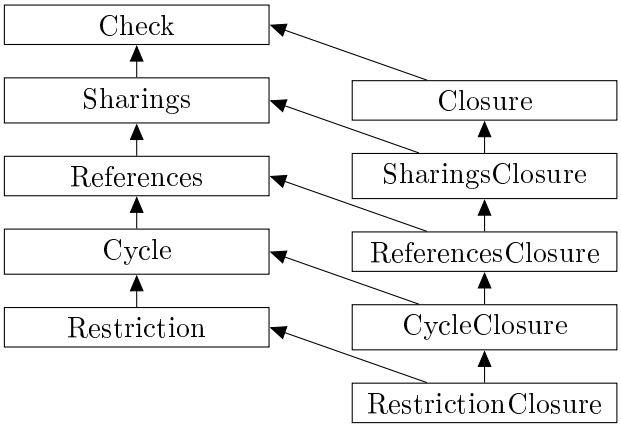
```
method process_recheck (sh, tys) =
  let data = sh.Sobj.sh_data in
  let get_ty tys = match tys with [ty] → ty | _ → assert false in
  let ty = get_ty data.Sharings.tys in
  let ty' = get_ty (check#simplify (ty :: tys)) in
  if TyHelpers.size ty' ≥ TyHelpers.size ty then begin
    let data = data.Sharings.data.Cycle.data in
    data.recheck_count ← data.recheck_count − 1;
    if data.recheck_count < 0 then fail ()
  end ;
  data.Sharings.tys ← [ty'];
  check#obj sh.Sobj.sh_obj data.Sharings.tys
```

En l'absence de type cyclique, le type ty' contient systématiquement moins de constructeurs de type que ty et le corps de la conditionnelle au milieu de la méthode *process_recheck* n'est jamais executé.

```
end (* class check *)
```

4.3 Algorithme de vérification avec fermeture

Cette section étend les cinq classes précédentes en leur ajoutant un mécanisme de vérification des fermetures. L'arbre d'héritage est alors le suivant :



Module Closure

41. Le module *Closure* permet d'ajouter au module *Check* un mécanisme de vérification des fermetures. Ce mécanisme nécessite de prendre en compte les variables existentielles pouvant apparaître dans les types attendus (voir section 3.3.1). Pour cela, les contraintes de type ne pouvant pas être vérifiées immédiatement seront explicitées et mémorisées dans une liste de contraintes restant à vérifier (45). Dans un second temps, un ordonnanceur minimaliste (49) permettra de reprendre la vérification de ces contraintes.

Cette première étape dans la description du prototype avec fermetures repose sur le système de type généralisé, ce qui permet d'ignorer le partage. Elle échoue en présence de champs modifiables.

42. La fonction *get_closure_tys* calcule les types attendus pour les éléments de l'environnement, à partir du type attendu pour une fermeture et du schéma de type associé au pointeur de code de cette fermeture. Elle échoue si le type attendu n'est pas compatible avec le type du pointeur de code. Cette fonction sera utile pour réaliser la règle de réécriture [R'-Clos].

```
let rec get_closure_tys sty ty : Ty.expression list =
  match Ty.desc ty with
  | Tlazy ty → get_closure_tys sty ty
  | _ →
      let (vars, ty_fun) = Ty.open_scheme sty in
      let ty_fun = Ty.resolve_alias ty_fun in
      match Ty.desc ty_fun with
      | Tarrow ("^env", ty_env, ty_ext) → begin
          match Ty.desc ty_env with
          | Ttuple tys_env →
              if ¬ (Ty.unify ty ty_ext) then fail () else
              tys_env
          | _ → assert false
```

```
    end
  | _ → if ¬(Ty.unify ty ty_fun) then fail () else []
```

43. La fonction *get_closure_tyss* permet d'itérer la fonction *get_closure_tys* (42) sur un ensemble de types attendus.

```
let rec get_closure_tyss sty tys : Ty.expression list list =
  match tys with
  | [] → assert false
  | [ty] → List.map (fun ty → [ty]) (get_closure_tys sty ty)
  | ty :: tys →
      let tyss = get_closure_tyss sty tys in
      let tys' = get_closure_tys sty ty in
      List.map2 (fun ty tys → ty :: tys) tys' tyss
```

44. La classe *check* étend la classe *Check.check* avec un mécanisme de vérification de fermetures. Elle est paramétrée par le type des données associées aux blocs partagés et par le type décrivant les contraintes qu'il est possible de rendre explicite.

```
class ['data, 'constraints] check = object (check) inherit ['data] Check.check as super
```

Les contraintes rendues explicites afin de retarder leur vérification seront représentées à l'aide d'un type variant. Ainsi, ce type pourra être étendu au fur et à mesure de la description du prototype. Pour cette première étape, il est suffisant de pouvoir retarder la vérification d'un entier, d'un bloc de base ou d'un bloc générique.

```
  constraint 'constraints = [>
  | `Int of int × Ty.expression list
  | `BasicBlock of 'data Sobj.t × Ty.expression list
  | `Block of int × 'data Sobj.t list × Ty.expression list
  ]
```

45. Les contraintes rendues explicites seront mémorisées dans une liste référencée par la variable d'instance *delayed_constraints*.

```
  val mutable delayed_constraints : 'constraints list = []
```

La méthode *Check.check#int* vérifie immédiatement la compatibilité d'un l'entier avec l'ensemble des types attendus qui ne sont pas des variables existentielles ; la vérification des variables existentielles étant retardées.

```
  method int i tys =
    let (vars, tys) = List.partition TyHelpers.is_var tys in
    if vars ≠ [] then delayed_constraints ← `Int (i, vars) :: delayed_constraints;
    if tys ≠ [] then super#int i tys
```

Les méthodes *Check.check#basic_block* et *Check.check#block* sont redéfinies en suivant le même principe.

```
method basic_block obj tys =
  let (vars, tys) = List.partition TyHelpers.is_var tys in
  if vars ≠ [] then
    delayed_constraints ← `BasicBlock (obj, vars) :: delayed_constraints;
  super#basic_block obj tys

method block tag fields tys =
  let (vars, tys) = List.partition TyHelpers.is_var tys in
  if vars ≠ [] then
    delayed_constraints ← `Block (tag, fields, vars) :: delayed_constraints;
  if tys ≠ [] then super#block tag fields tys
```

46. La méthode *closure* effectue la vérification d'une fermeture. Pour cela, elle calcule les types attendus pour l'environnement d'une fermeture à l'aide de la fonction *get_closure_tyss* (43) ; puis, à l'aide des types obtenus, elle appelle récursivement la méthode *check#sobj* sur chacun des éléments de l'environnement. La variable d'instance *unification* permet de se souvenir qu'un calcul d'unification a été effectué par la fonction *get_closure_tyss*. Ce calcul a éventuellement permis d'instancier des variables de types et la vérification de certaines contraintes retardées peut être reprise.

```
val mutable unification = false
method closure sty env tys =
  let tyss = get_closure_tyss sty tys in
  if List.length tyss ≠ List.length env then fail ();
  unification ← true;
  List.iter2 check#sobj env tyss
```

47. À la fin du parcours récursif, la méthode *Check.check#check* réamorce la vérification des contraintes retardées.

```
method check (obj : 'data Sobj.t) tys =
  super#check obj tys;
  check#schedule_postponed
```

La vérification des contraintes retardées est effectuée conjointement par les méthodes *check#schedule_postponed* (49) et *check#constraint_* (48).

48. La méthode *constraint_* permet de reprendre la vérification d'une contrainte.

```
method constraint_ c = match c with
| `Int (i, tys) → check#int i tys
| `BasicBlock (obj, tys) → ignore (check#basic_block obj tys)
| `Block (tag, fields, tys) → check#block tag fields tys
| _ → assert false
```

49. La méthode *check#schedule_postponed* ordonnance la vérification des contraintes retardées. Pour cela, elle reprend la vérification de toutes les contraintes retardées en

itérant la méthode *check#constraint_* sur la liste *delayed_constraints*. Les contraintes dont la vérification n'est toujours pas possible sont immédiatement réinsérées dans la liste *delayed_constraints* par l'une des méthodes *check#int*, *check#basic_block* ou *check#block* (45). Lorsqu'au cours d'un parcours de la liste des contraintes retardées aucune unification n'a eu lieu, aucune des contraintes restantes ne peut plus être vérifiées. Ces contraintes sont alors ignorées, conformément à la règle [R-UNIV]. Si au contraire une unification a eu lieu, la liste des contraintes retardées est à nouveau parcourue.

```
  method schedule_postponed =
    while unification do
      let delayed = delayed_constraints in
      unification ← false;
      delayed_constraints ← [];
      List.iter check#constraint_ delayed
    done
end (* class check *)
```

Module SharingsClosure

50. Le module *SharingsClosure* étend la classe *Closure.check* pour lui ajouter un mécanisme de gestion du partage. Il reprend le mécanisme de tri topologique paresseux réalisé par la classe *Sharings.check* à l'aide de compteurs de références. Cette étape dans la description du prototype échoue en présence de cycle ou de valeurs modifiables.

51. Comme l'a montré la section 3.7.1, pour réaliser un parcours topologique du graphe mémoire en présence de variables de type existentielles, il est parfois nécessaire de pouvoir commencer le vérification d'un bloc dont l'ensemble des types attendus est un ensemble de variables existentielles. Suivant les principes de la règle [R'-BLK], il est alors possible de vérifier un bloc dont le type attendu est $\dot{\beta}$ en vérifiant les éléments du bloc à l'aide de nouvelles variables existentielles $\dot{\alpha}_1, \ldots, \dot{\alpha}_n$ et en explicitant une contrainte $[\dot{\alpha}_1, \ldots, \dot{\alpha}_n] \triangledown \dot{\beta}$ — voir la contrainte '*Triangle* (53) et la méthode *force_block* (55). Pour ne pas introduire trop de variables existentielles, la vérification d'un bloc pour lequels l'ensemble des types attendus contient des variables existentielles est retardée le plus possible — voir les méthodes *block* (54), *force_constraint* (57) et *schedule_postponed* (56).

52. La classe *check* hérite doublement des classes *Closure.check* et *Sharings.check*.

```
class ['data, 'constraints] check = object (check)
  inherit ['data] Sharings.check as check_sharings
  inherit ['data Sharings.data, 'constraints] Closure.check as check_closure
```

La méthode *sobj* commune aux classes *Sharings.check* et *Closure.check* a été redéfinie par la classe *Sharings.check*. Pour ne pas masquer cette redéfinition, on délègue explicitement les appels à la méthode *sobj* à la méthode redéfinie dans la classe *Sharings.check*.

```
method sobj = check_sharings#sobj
```

53. La contrainte ‘*Triangle* représente une liste de contraintes $[\tau_1, \ldots, \tau_n]\ \nabla_{tag}\ \tau$ indicée par le même *tag*.

```
constraint 'constraints = [>
| ‘Triangle of int × (Ty.expression list × Ty.expression) list
]
```

Dès que l'un des types à droite d'un triangle ∇ a été instancié, il est possible de vérifier partiellement une contrainte ‘*Triangle* en unifiant les variables de types à gauche du triangle avec les types attendus calculés par la fonction *get_generic_block_tys* (6).

```
method constraint_ c = match c with
| ‘Triangle (tag, tyss) → ignore (check#triangle tag tyss)
| c → check_closure#constraint_ c

method triangle tag (tyss : (Ty.expression list × Ty.expression) list) =
  let (varss, tyss) = List.partition (fun (_, ty) → TyHelpers.is_var ty) tyss in
  if tyss ≠ [] then
    ( if varss ≠ [] then
        delayed_constraints ← ‘Triangle (tag, varss) :: delayed_constraints )
  else
    let (ftyss, tys) = List.split tyss in
    let size = List.length (List.hd ftyss) in
    let (muts', ftyss') = Check.get_generic_block_tyss tag size tys in
    ignore (check#block_mutability (muts', ftyss'));
    List.iter2
      (fun tys tys' → if ¬ (List.for_all2 Ty.unify tys tys') then fail ())
      ftyss ftyss';
    unification ← true;
    if varss ≠ [] then
      delayed_constraints ← ‘Triangle (tag, varss) :: delayed_constraints
```

54. Contrairement à la méthode *Closure.check#block*, et afin ne pas invalider les invariants des compteurs de références utilisés par le tri topologique, il est n'est pas possible de vérifier partiellement l'ensemble des types attendus en vérifiant immédiatement les types qui ne sont pas des variables. Ainsi, la méthode *Closure.check#block* (45) est redéfinie pour retarder la vérification d'un bloc dont l'un des types attendus est une variable de type.

```
method block tag fields tys =
  if List.exists TyHelpers.is_var tys then
    delayed_constraints ← ‘Block (tag, fields, tys) :: delayed_constraints
  else
    check_sharings#block tag fields tys
```

55. À l'opposé de la méthode *block* (54), la méthode *force_block* permet de commencer la vérification d'un bloc dont certains des types attendus sont des variables de types. Elle suit les principes décrits au point 51.

```
method force_block tag fields tys =
  let size = List.length fields in
  let (vars, tys) = List.partition TyHelpers.is_var tys in
  let fvarss = List.map (fun _ → TyHelpers.create_var_list size) vars in
  let vars = List.combine fvarss vars in
  if tys = [] then
    ( if vars ≠ [] then
        delayed_constraints ← 'Triangle (tag, vars) :: delayed_constraints;
      List.iter2 check#sobj fields fvarss )
  else
    let (muts, tyss) = Check.get_generic_block_tyss tag size tys in
    ignore (check#block_mutability (muts, tyss));
    if vars ≠ [] then
      delayed_constraints ← 'Triangle (tag, vars) :: delayed_constraints;
    let tyss = List.fold_left (List.map2 (fun tys var → var :: tys)) tyss fvarss in
    List.iter2 check#sobj fields tyss
```

56. L'ordonnanceur minimaliste décrit par la méthode *Closure.schedule_postponed* (49) est étendu : lorsqu'aucune contrainte retardée ne peut plus être vérifiée, la vérification des blocs restants est forcée à l'aide des méthodes *force_constraint* (57) et *force_block* (55). Une fois tous les blocs vérifiés — c'est-à-dire lorsqu'il ne reste plus que des contraintes '*Int*, '*BasicBlock* ou '*Triangle* pour lesquels les types attendus sont des variables de types — la méthode *Sharings.schedule_postponed* (21) est appelée pour détecter la présence de cycle.

```
method schedule_postponed =
  while unification do
    check_closure#schedule_postponed;
    let delayed = delayed_constraints in
    delayed_constraints ← [];
    List.iter check#force_constraint delayed;
  done;
  check_sharings#schedule_postponed
```

57. La méthode *force_constraint* se contente de forcer la vérification d'une contrainte '*Block* à l'aide de la méthode *force_block* (55). En guise d'optimisation et en suivant les principes de la règle [R'-Univ], la vérification d'un bloc dont tous les types attendus sont des variables de type n'est pas forcée si aucune des contraintes de type ne porte sur les blocs partagés dont la vérification est retardée.

```
  method force_constraint c = match c with
  | `Block (tag, fields, tys)
    when ¬ (List.for_all TyHelpers.is_var tys ∧
              List.for_all
                (fun sh → List.for_all TyHelpers.is_var sh.Sobj.sh_data.Sharings.tys)
                postponed_block) →
      check#force_block tag fields tys
  | c → delayed_constraints ← c :: delayed_constraints

end (* class check *)
```

Module ReferencesClosure

58. Le module *ReferencesClosure* ajoute la gestion des valeurs modifiables à la classe *SharingsClosure.check* en adaptant les mécanismes réalisés par la classe *References.check*. Pour cela, la classe *check* hérite des classes *References.check* et *SharingsClosure.check*.

```
class ['data, 'constraints] check = object (check)

  inherit ['data] References.check as check_references
  inherit ['data, 'constraints] SharingsClosure.check as check_closure

  method block_mutability = check_references#block_mutability
  method basic_block_mutability = check_references#basic_block_mutability
```

59. Les méthodes *Closure.check#basic_block* (45) et *SharingsClosure.check#triangle* (53) permettent respectivement des vérifications partielles des contraintes `*BasicBlock* et `*Triangle*. En présence de valeurs modifiables, il faut imposer pour chaque étape de vérification partielle que le caractère modifiable des champs est le même qu'à l'étape de vérification précédente. Pour mémoriser le caractère modifiable des champs lors d'une vérification partielle, on introduit les contraintes `*BasicBlockMut* et `*TriangleMut*.

```
  constraint 'constraints = [>
  | `BasicBlockMut of 'data Sharings.sobj × bool list × Ty.expression list
  | `TriangleMut of
      int × Ty.expression option list × (Ty.expression list × Ty.expression) list
  ]
```

60. La méthode *Closure.check#basic_block* (45) est redéfinie pour mémoriser le caractère modifiable des types déjà vérifiés pour ce bloc de base.

```
  method basic_block obj tys = check#basic_block_mut obj None tys
```

La vérification d'une contrainte `*BasicBlockMut* est similaire à la vérification d'une contrainte `*BasicBlock*, avec la condition supplémentaire que si le caractère modifiable des champs imposé par les types vérifiés à cette étape n'est pas le même que le caractère mémorisé, la vérification échoue.

```
method basic_block_mut obj muts tys =
  let (vars, tys) = List.partition TyHelpers.is_var tys in
  if tys = [] then
    ( if vars ≠ [] then check#delay_basic_block obj muts vars;
      muts )
  else
    let muts' = check_references#basic_block obj tys in
    if muts ≠ None ∧ muts ≠ muts' then fail ();
    if vars ≠ [] then check#delay_basic_block obj muts' vars;
    muts'

method delay_basic_block obj muts vars = match muts with
| None →
    delayed_constraints ← 'BasicBlock (obj, vars) :: delayed_constraints
| Some muts →
    delayed_constraints ← 'BasicBlockMut (obj, muts, vars) :: delayed_constraints
```

61. De la même manière, la méthode *SharingsClosure.check#triangle* (53) est redéfinie pour mémoriser le caractère modifiable des types déjà vérifiés. Pour cela, la vérification d'une contrainte '*TriangleMut* est similaire à la vérification d'une contrainte '*Triangle*, avec la même condition supplémentaire que précédemment (60).

```
method triangle tag tyss = check#triangle_mut tag None tyss

method triangle_mut tag muts tyss =
  let (varss, tyss) = List.partition (fun (_, ty) → TyHelpers.is_var ty) tyss in
  if tyss = [] then
    (if varss ≠ [] then check#delay_triangle tag muts varss)
  else
    let ftyss, tys = List.split tyss in
    let size = List.length (List.hd ftyss) in
    let muts', ftyss' = Check.get_generic_block_tyss tag size tys in
    let muts' = check#block_mutability (muts', ftyss') in
    if ¬ (check#equal_mutability muts muts') then fail ();
    List.iter2
      (fun tys tys' → if ¬ (List.for_all2 Ty.unify tys tys') then fail ())
      ftyss ftyss';
    unification ← true;
    if varss ≠ [] then check#delay_triangle tag (Some muts') varss

method delay_triangle tag muts vars = match muts with
| None →
    delayed_constraints ← 'Triangle (tag, vars) :: delayed_constraints
| Some muts →
    delayed_constraints ← 'TriangleMut (tag, muts, vars) :: delayed_constraints
```

```
method equal_mutability muts muts' = match muts with
| None → true
| Some muts →
    let unify_opt ty ty' = match ty, ty' with
    | None, None → true
    | Some ty, Some ty' → Ty.unify ty ty'
    | _, _ → false in
    List.length muts = List.length muts' ∧ List.for_all2 unify_opt muts muts'
```

62. Sur le même modèle que la méthode *Closure.check#basic_block* (60), la méthode *SharingsClosure.check#force_block* (55) mémorise dans le cas des blocs le caractère modifiable des types éventuellement vérifiés à cette étape.

```
method force_block tag fields tys =
  let size = List.length fields in
  let (vars, tys) = List.partition TyHelpers.is_var tys in
  let fvarss = List.map (fun _ → TyHelpers.create_var_list size) vars in
  let vars = List.combine fvarss vars in
  if tys = [] then
    ( if vars ≠ [] then check#delay_triangle tag None vars;
      List.iter2 check#sobj fields fvarss )
  else
    let (muts, tyss) = Check.get_generic_block_tyss tag size tys in
    let muts = check#block_mutability (muts, tyss) in
    if vars ≠ [] then check#delay_triangle tag (Some muts) vars;
    let tyss = List.fold_left (List.map2 (fun tys var → var :: tys)) tyss fvarss in
    List.iter2 check#sobj fields tyss
```

63. La méthode *References.check#equal_and_monomorphic* (23) doit être redéfinie en présence de variables de type existentielles, pour effectuer un test d'unification à la place d'un simple test d'égalité. Le test de « monomorphicité » peut être retardé.

```
method equal_and_monomorphic tys = match tys with
  | [] → assert false
  | ty :: tys →
      if ¬ (List.for_all (Ty.unify ty) tys) then fail ();
      if tys ≠ [] then unification ← true;
      check#is_monomorphic ty;
      ty

method is_monomorphic ty =
  try if ¬ (TyHelpers.is_monomorphic ty) then fail ()
  with TyHelpers.Maybe →
    delayed_constraints ← 'Closed ty :: delayed_constraints
```

```
constraint 'constraints = [>
| `Monomorphic of Ty.expression
]
```

Lorsqu'à la fin de la vérification il reste des contraintes `Closed qui ne peuvent être vérifiées à cause de variables existentielles, elles sont ignorées.

```
method constraint_ c = match c with
| `BasicBlockMut (obj, muts, tys) →
    ignore(check#basic_block_mut obj (Some muts) tys)
| `TriangleMut (tag, muts, ftyss) → check#triangle_mut tag (Some muts) ftyss
| `Monomorphic ty → check#is_monomorphic ty
| c → check_closure#constraint_ c
end (* class check *)
```

Module CycleClosure

64. Le module *CycleClosure* ajoute un mécanisme de gestion des cycles à la classe *SharingsClosure.check*. Elle reprend le mécanisme de tri topologique explicite réalisé par le module *Cycle* en héritant des classes *Cycle.check* et *ReferencesClosure.check*.

```
class ['data, 'constraints] check = object (check)
  inherit ['data] Cycle.check as check_cycle
  inherit ['data Cycle.data, 'constraints] ReferencesClosure.check as check_closure
  method cycle = check_cycle#cycle
  method shared = check_cycle#shared
```

65. En présence de variables existentielles, il n'est pas toujours possible de décider si un type est instance ou non d'un autre type. La problème se pose pour la méthode *Cycle.check#back_ref* (33) qui teste l'instanciation de types attendus pour un bloc partagé ayant déjà été vérifié avec au moins un des types mémorisés pour ce bloc. Nous introduisons alors la contrainte `BackRef pour retarder ces tests.

```
constraint 'constraints = [>
| `BackRef of 'data Cycle.sobj_data × Ty.expression list
]
```

Cette nouvelle contrainte sera vérifiée à l'aide de la méthode *back_ref* (66).

```
method constraint_ c = match c with
| `BackRef (sh, tys) → check#back_ref sh tys
| c → check_closure#constraint_ c
```

66. La méthode *Cycle.check#back_ref* (33) est redéfinie pour mémoriser à l'aide de la contrainte `BackRef les tests d'instanciation pour lesquelles elle ne sait pas décider immédiatement du résultat.

```
  method back_ref sh tys =
    let data = sh.Sobj.sh_data in
    let tys = check#simplify tys in
    data.Sharings.tys ← check#simplify data.Sharings.tys;
    let delayed_tests =
      List.fold_left
        (fun acc ty →
          match TyHelpers.fold_is_instance ty data.Sharings.tys with
          | Some true → acc
          | Some false → check#recheck sh [ty]; acc
          | None → ty :: acc)
        [] tys in
    if delayed_tests ≠ [] then
      delayed_constraints ← 'BackRef (sh, delayed_tests) :: delayed_constraints
```

67. À la fin de ce parcours de vérification, la méthode *finalize* tente de résoudre les contraintes '*Back_ref* non résolues par unification des types attendus avec les types mémorisés. Cette heuristique est décrite à la section 3.7.2.

```
  method check (obj : 'data Cycle.sobj) (tys : Ty.expression list) : unit =
    check_closure#check obj tys;
    check#finalize

  method finalize =
    let try_unify sh tys =
      let data = sh.Sobj.sh_data in
      let tys = check#simplify tys in
      data.Sharings.tys ← check#simplify data.Sharings.tys;
      List.for_all
        (fun ty → List.exists (Ty.try_unify ty) data.Sharings.tys)
        tys in
    let filter_back_ref = function
      | 'BackRef (sh, tys) when try_unify sh tys → (unification ← true; false)
      | _ → true in
    delayed_constraints ← List.filter filter_back_ref delayed_constraints;
    if ¬ unification ∧
       List.exists (function 'BackRef _ → true | _ → false) delayed_constraints
    then fail ();
    if unification
    then (check#schedule_postponed; check#finalize)

end (* class check *)
```

Module RestrictionClosure

68. Le module *RestrictionClosure* restreint l'ensemble des graphes mémoire acceptés par le module *CycleClosure* à ceux typables dans le système de type original en reprennant les mécanismes d'anti-unification réalisés par le module *Restriction*. En présence de variables de type existentielles, il ne sera pas toujours possible de calculer l'anti-unificateur de deux types. Dans ces cas-là, il sera toujours possible de continuer le parcours de vérification tant que l'ensemble de types attendu est homogène — voir la section 3.4.2 et le point (73).

69. La fonction *check_homogeneity* vérifie que tous les types d'un ensemble possèdent le même constructeur de type de tête. Elle échoue si un des types de l'ensemble est une variable existentielle.

```
let rec check_homogeneity tys =
  match tys with
  | [] → assert false
  | [ty] → ty
  | ty :: tys →
      let ty' = check_homogeneity tys in
      if ¬ (TyHelpers.check_constr ty ty') then fail();
      ty'
```

70. La classe *check* hérite des classes *Restriction.check* et *CycleClosure.check*.

```
class ['data, 'constraints] check = object (check)

  inherit ['data] Restriction.check as check_restrition
  inherit ['data Restriction.data, 'constraints] CycleClosure.check as check_closure

  method scc = check_restrition#scc
  method recheck = check_restrition#recheck
```

71. Les prédicats d'homogénéité ne pouvant être vérifiés immédiatement seront représentés à l'aide de la contrainte ‘*Homogeneous*. Ces contraintes seront ensuite vérifiées à l'aide de la méthode *homogeneity* (72).

```
  constraint 'constraints = [>
  | ‘Homogeneous of Ty.expression list
  ]

  method constraint_ c = match c with
  | ‘Homogeneous tys → ignore(check#homogeneity tys)
  | c → check_closure#constraint_ c
```

72. La méthode *homogeneity* permet de contraindre un ensemble de types à être homogène. Si l'ensemble est constitué uniquement de variables existentielles la vérification est retardée en explicitant la contrainte ‘*Homogeneous* (71). Sinon, la vérification est immédiate. Dans le cas particulier d'un ensemble de types partagé entre un sous-ensemble

de types homogène et un sous-ensemble de variables de type, les variables de type sont instanciées en suivant principe de la règle de réécriture [R'-◊].

```
method homogeneity tys =
  let vars, tys = List.partition TyHelpers.is_var tys in
  match (vars, tys) with
  | ([], []) → []
  | (([_] as tys), [] | [], ([_] as tys)) → tys
  | (vars, []) →
      delayed_constraints ← 'Homogeneous vars :: delayed_constraints;
      vars
  | ([], tys) → ignore (check_homogeneity tys); tys
  | (vars, tys) →
      let ty = check_homogeneity tys in
      let vars' = List.map (fun _ → TyHelpers.duplicate_constr ty) vars in
      ignore (List.map2 Ty.unify vars vars');
      unification ← true;
      check#simplify (tys @ vars)
```

73. La méthode *Sharings.check#sobj* est redéfinie pour vérifier systématiquement l'homogénéité de l'ensemble des types attendus pour une valeur.

```
method sobj obj tys = check_closure#sobj obj (check#homogeneity tys)
```

74. La méthode *Sharings.check#merge_insert* — méthode auxiliaire de la méthode *Sharings.check#simplify* (20) — est redéfinie pour permettre d'anti-unifier deux types lorsque cela est possible.

```
method merge_insert ty' ty tys =
  try check_restrition#merge_insert ty' ty tys
  with TyHelpers.Maybe → check_closure#merge_insert ty' ty tys
```

75. La méthode *Restriction.check#process_recheck* (40) est redéfinie pour prendre en compte les blocs partagés pour lesquels l'ensemble des types mémorisés n'est pas réduit à un singleton.

```
method process_recheck (sh, tys) =
  let data = sh.Sobj.sh_data in
  let tys =
    match (data.Sharings.tys, check#simplify (data.Sharings.tys @ tys)) with
    | ([ty], [ty']) when TyHelpers.size ty' < TyHelpers.size ty → [ty']
    | (tys, tys') →
        let data = data.Sharings.data.Cycle.data in
        data.Restriction.recheck_count ← data.Restriction.recheck_count − 1;
        if data.Restriction.recheck_count < 0 then fail ();
        let back_refs, delayed =
```

```
        List.partition
          (function 'BackRef (sh', _) → sh ≡ sh' | _ → false)
          delayed_constraints in
      delayed_constraints ← delayed;
      let back_refs_tys =
        List.concat
          (List.map
             (function 'BackRef (_, tys) → tys | _ → assert false)
             back_refs) in
      check#simplify (tys @ tys' @ back_refs_tys) in
  data.Sharings.tys ← tys;
  check#obj sh.Sobj.sh_obj data.Sharings.tys;
  check#schedule_postponed
```

76. La méthode *CycleClosure.check#finalize* (67) peut réintroduire des blocs partagés dans la liste *postponed_recheck* (37). Il est alors nécessaire de redéfinir la méthode *finalize* pour effectuer la vérification de ces blocs.

```
method finalize =
  check_closure#finalize;
  if postponed_recheck ≠ [] then begin
    check#schedule_recheck []; (* identifiant du graphe complet *)
    check#finalize
  end

end (* class check *)
```

4.4 Bilan

Modifications du compilateur OCaml Par rapport aux mécanismes de (dé)sérialisation décrits au chapitre 1, la fonction de désérialisation construite dans cette thèse a pu être mise en œuvre dans le compilateur OCaml sans modifier la fonction de sérialisation. Le mécanisme proposé ici est indépendant de la représentation externe du graphe-mémoire. En particulier, il est indépendant de la représentation externe d'une fermeture et il serait toujours utilisable en présence d'un mécanisme automatique de chargement dynamique de code au moment de la désérialisation.

Seules les modifications du compilateur OCaml permettant de conserver le type statique de chaque pointeur de code (section 4.1.3) ont été nécessaires. Nous avons pris garde de les concevoir dans le souci de ne pas ajouter de surcoût à l'exécution d'un programme n'utilisant pas la désérialisation.

Ces modifications légères ne permettent pas la vérification des invariants parfois associés à un type abstrait par la signature d'un module, contrairement à la fonction de désérialisation du compilateur HashCaml [Billings *et al.*, 2006]. Notre fonction garantit uniquement la compatibilité de la valeur désérialisée avec la représentation concrète du

type. Ce choix à l'avantage d'offrir la sûreté d'exécution lors d'une communication avec un client non fiable, tout comme les mécanismes de génération de fonctions de (dé)sérialisation *ad-hoc* [Kennedy, 2004; Elsman, 2005; Yallop, 2007; Sutou et Sumii, 2007]. Dans ce cadre, notre fonction est la seule à prendre en compte les valeurs fonctionnelles.

Partage dans le graphe mémoire Le mécanisme de détection de partage dans une valeur utilisé par Kennedy [2004] ou Yallop [2007] repose sur un principe d'égalité spécifique à chaque type de données, généralement une égalité structurelle monomorphe, alors que la fonction de sérialisation d'OCaml utilise l'égalité physique. Le premier mécanisme optimise la taille de la représentation sérialisée en y introduisant plus de partage que dans le graphe-mémoire original. Cependant, la nécessité d'utiliser un mécanisme d'anti-unification dans notre algorithme — pour prendre en compte le partage polymorphe introduit par une construction `let _ = _ in _` — permet de construire des exemples où le mécanisme de détection de partage par égalité structurelle détecte moins de partage que l'égalité physique (voir le tas de l'exemple 3.4.1). Par ailleurs, la fonction de sérialisation générique du compilateur OCaml étant dépendante de la représentation des données, elle utilise des astuces assurant une détection du partage en temps constant ; elle est alors, du point de vue de la vitesse d'exécution, vraisemblablement très proche de l'optimale.

Conclusion

Cette thèse a proposé, dans le cadre d'un langage de programmation fonctionnel et impératif à la ML utilisant une représentation uniforme des données, un algorithme de vérification de compatibilité d'un graphe-mémoire avec un type. Cet algorithme permet notamment de définir un mécanisme de (dé)sérialisation typé qui refuse de désérialiser une valeur dont le graphe mémoire est incompatible avec le type attendu.

Graphes-mémoire compatibles et algorithme de vérification Plus précisément, cette thèse a caractérisé, pour les types de données algébriques usuels, les types fonctionnels et les références, l'ensemble des représentations-mémoire (entiers, blocs, fermetures) qui leurs sont compatibles. On a montré que cette caractérisation permet de définir un mécanisme de (dé)sérialisation — basé sur une simple linéarisation du graphe-mémoire et un test de compatibilité avec le type attendu lors de la désérialisation — qui préserve les résultats usuels de correction du typage (théorème 2.5.4). Le lemme de préservation du typage par évaluation (lemme 2.5.2) garantit notamment que cette notion de compatibilité avec un type inclut l'ensemble des représentations-mémoire productibles par un programme ML de même type.

Dans un second temps, cette caractérisation a permis la définition d'un algorithme testant l'appartenance d'un graphe-mémoire à l'ensemble des représentations-mémoire compatibles avec un type donné. Cet algorithme est décrit à l'aide d'un système de réécriture et d'une stratégie d'application visant l'efficacité. On a montré ensuite la correction de cet algorithme (lemme 3.4.8). En l'absence de certaines fermetures polymorphes ou de cycle dans le graphe-mémoire, il a été possible de vérifier aussi la complétude de l'algorithme (lemme 3.4.10). Mais, malheureusement, en présence de certains graphes-mémoire contenant des cycles et des fermetures polymorphes, le système de réécriture proposé contient des suites infinies de réécritures. Pour garantir en pratique la terminaison du test de compatibilité dans tous les cas, on borne le nombre de vérifications successives autorisées d'un cycle. L'algorithme obtenu n'est pas complet ; il reste à déterminer à l'usage si la borne proposée n'est pas trop restrictive.

Fonction de désérialisation sûre L'algorithme de vérification de compatibilité nous a permis d'étendre le compilateur OCaml avec un mécanisme de (dé)sérialisation sûre qui réutilise la mécanisme de (dé)sérialisation existant. Plus précisément, la fonction

de sérialisation n'est pas modifiée ; et notre fonction de désérialisation réutilise la fonction de désérialisation non sûre pour reconstruire le graphe mémoire avant de vérifier sa compatibilité avec le type attendu. Ce principe permet de ne pas dépendre de la représentation externe du graphe mémoire. En particulier, il est indépendant de la représentation externe d'une fermeture et il serait toujours utilisable en présence d'un mécanisme automatique de chargement dynamique de code au moment de la désérialisation.

Limitations et extensions Le prototype proposé au chapitre 4 a étendu le mécanisme de vérification du chapitre 3 à l'ensemble des types prédéfinis du compilateur OCaml (flottants, chaînes de caractères, ...) ainsi qu'à l'ensemble des définitions de types (types somme, types enregistrement, avec champs modifiables ou non). Ce paragraphe discute les extensions possibles de l'algorithme permettant de couvrir l'intégralité des valeurs sérialisables et en particulier les variants polymorphes, les modules de première classe, les objets, et les types algébriques généralisés.

Types algébriques généralisés Les types algébriques généralisés (GADT) [Xi *et al.*, 2003; Pottier et Régis-Gianas, 2006] sont une extension des types somme usuels, où notamment chacun des constructeurs de données d'un même type peut contraindre les paramètres du type d'une manière distincte. Après la vérification du typage, les valeurs appartenant à un GADT peuvent être compilées de la même manière que les valeurs des types algébriques usuels — c'est-à-dire par des entiers pour les constructeurs constants, et par des blocs *taggués* pour les constructeurs non-constants. Sous cette hypothèse de représentation mémoire des GADT, l'extension de notre algorithme à la vérification de telles valeurs semble directe en ajoutant des principes venant du mécanisme de vérification des fonctions au mécanisme de vérification des types algébriques. En effet, d'un côté la vérification d'un constructeur non-constant peut introduire des inconnues de types de même nature que les inconnues de type pouvant être introduites par la vérification d'une fermeture. Et d'un autre côté, le type produit par un constructeur de données doit être unifié avec le type attendu, comme le type d'un pointeur de code est unifié avec le type attendu pour une fermeture.

Module de première classe Les dernières versions du compilateur OCaml permettent l'utilisation de modules de première classe, et de telles valeurs peuvent donc être sérialisées au même titre que toutes les autres valeurs du langage. La représentation en mémoire d'un tel module est similaire à celle d'un n-uplet contenant l'ensemble des valeurs exportées par le module. En présence de module n'exportant pas de déclaration de type, vérifier la compatibilité d'une valeur désérialisée avec la signature d'un module semble alors similiaire à la vérification du type d'un n-uplet. Lorsque la signature attendue pour le module contient des définitions de type, la vérification semble moins évidente, mais les mécanismes proposés dans cette thèse et le prototype apportent déjà quelques pistes. En particulier, en présence d'une définition de type manifeste dans la signature attendue, il semble nécessaire de la comparer au type ayant servi à créer le module sérialisé : notre prototype modifie déjà la représentation d'un module pour conserver explicitement cette informa-

tion de type. D'un autre côté, les types non manifestes dans la signature attendue pourront être instanciés par l'information de typage conservée explicitement dans l'implémentation du module sérialisé.

Sous-typage structurel : variants polymorphes et objets Le compilateur OCaml contient une extension des types somme usuels appelée variants polymorphes [Garrigue, 1994]. Elle permet une forme de sous-typage structurel. Dans un premier temps, la vérification de variants polymorphes clos semble pouvoir s'effectuer comme une extension du mécanisme de vérification des types somme classiques : la relation de décomposition ∇ peut être étendue en l'indexant par le nom du variant polymorphe ; et l'opération d'anti-unification de deux variants polymorphes semble ensuite pouvoir se formaliser en position covariante (respectivement contravariante) comme l'intersection (resp. l'union) des deux types. Une question reste cependant à résoudre : le mécanisme d'inférence de type utilisé en la présence d'inconnues de type est-il suffisant pour prendre en compte les variants polymorphes ouverts [1] ?

L'extension objet d'OCaml [Rémy et Vouillon, 1998] repose elle aussi sur un mécanisme de sous-typage structurel. Du point de vue de la représentation mémoire, les objets sont représentés par des enregistrements comprenant la table des méthodes et les variables d'instance ; la table des méthodes est conceptuellement un tableau de fermetures ; chaque fermeture attend en premier argument l'enregistrement représentant l'objet sur lequel elle est appelée. Cette représentation permet de sérialiser un objet avec le mécanisme générique de sérialisation d'OCaml sans traitement spécifique. Étendre l'algorithme de vérification de type proposé dans cette thèse pour vérifier des types objets demanderait au préalable :

- de formaliser un mécanisme d'anti-unification dual du mécanisme proposé pour les variants polymorphes — c'est-à-dire effectuer l'union (resp. l'intersection) des méthodes en position covariante (resp. contravariante) ;
- et d'être capable d'associer un type à chacun des pointeurs de code contenus dans la table des méthodes.

Comme dans le cas des fonctions, l'information conservée sur le type statique des pointeurs de code aurait un double rôle : être confronté au type attendu pour la méthode représentée par la fermeture et apporter de l'information de typage pour les variables d'instance et les méthodes privées de l'objet contenant la méthode.

Autres utilisations de l'algorithme En plus d'être utile pour une fonction de désérialisation typée, les principes de l'algorithme de vérification de compatibilité d'un graphe-mémoire avec un type pourraient être réutilisés dans d'autres cadres. Par exemple, ils pourraient sans trop de modifications être utiles lors de l'écriture de programme OCaml inter-opérant avec un langage non typé à l'aide du mécanisme de fonction externe : l'algorithme proposé permettrait de vérifier la compatibilité des valeurs ML construites par une fonction externe.

1. Vraisemblablement le type attendu pour une valeur désérialisée est toujours clos, mais lorsqu'une valeur sérialisée contient une fermeture, le type associé à son pointeur de code peut contenir des variants polymorphes ouverts.

D'un autre point de vue, l'algorithme pourrait être réutilisé à la manière de Aditya et Caro [1993] pour reconstruire le type des valeurs allouées dans le tas à partir du type des racines. Cette propagation de type permettrait par exemple, comme l'outil `hp2ps` pour le compilateur GHC [GHC Team, 2010] ou l'outil `ocaml-memprof` [Le Fessant et Mimram, 2004] pour le compilateur OCaml, d'obtenir des statistiques sur l'utilisation du tas. Aucun de ces deux outils ne contient pour l'instant de mécanisme de propagation de type franchissant les fermetures.

Conclusion Parmi les différents mécanismes de sérialisation présentés dans l'introduction ou dans le chapitre 1, les mécanismes à la fois génériques et sûrs du point de vue du système de types du langage hôte reposent ou bien sur un mécanisme de génération de fonctions ad-hoc, ou bien sur des primitives typage dynamique nécessitant de définir une représentation externalisable des types. Cette thèse a montré qu'il est possible d'étendre un langage typé statiquement avec un mécanisme de sérialisation sûre reposant sur une notion de compatibilité d'un graphe mémoire avec un type, y compris en présence de valeurs fonctionelles. La définition de cette notion de compatibilité indépendamment du mécanisme de sérialisation permet entres autres : d'envisager d'étendre la notion à plus de types de haut-niveau : GADT, variant polymorphe, ... — une réutilisation dans d'autres cadres : débogueur, analyse du tas, ... — tout en n'imposant pas de surcoût aux programmes n'utilisant pas ces mécanismes.

Bibliographie

Martín Abadi, Luca Cardelli, Benjamin C. Pierce et Didier Rémy : Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, janvier 1995.

Shail Aditya et Alejandro Caro : Compiler-directed type reconstruction for polymorphic languages. *In FPCA '93 : Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 74–82, juin 1993.

Shail Aditya, Christine H. Flood et James E. Hicks : Garbage collection for strongly-typed languages using run-time type reconstruction. *In LFP '94 : Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 12–23, juin 1994.

Andrew W. Appel : Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2 (2):153–162, juillet 1989.

John W. Backus : *The Fortran automatic coding system for the IBM 704*. IBM, 1954.

Nick Benton, Andrew Kennedy et Claudio Russo : *The SML.NET 1.2 user guide*, 2006.

John Billings, Peter Sewell, Mark R. Shinwell et Rok Strnisa : Type-safe distributed programming for OCaml. *In ML'06 : Proceedings of the ACM Workshop on ML*, pages 20–31, septembre 2006.

Gilad Bracha, Martin Odersky, David Stoutamire et Philip Wadler : Making the future safe for the past : adding genericity to the java programming language. *In OOPSLA '98 : Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 183–200, octobre 1998.

Peter Canning, William Cook, Walter Hill, Walter Olthoff et John C. Mitchell : F-bounded polymorphism for object-oriented programming. *In FPCA '89 : Proceedings of the fourth international Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, septembre 1989.

Luca Cardelli et Peter Wegner : On understanding types, data abstraction, and polymorphism. *ACM Computing Survey*, 17(4):471–523, décembre 1985.

Eric Cartwright et Brian Stoler : Efficient implementation of run-time generic types for java. *In IFIP WG2.1 Working Conference on Generic Programming*, pages 207–236, 2002.

Robert Cartwright et Guy L. Steele, Jr. : Compatible genericity with run-time types for the java programming language. *In OOPSLA '98 : Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 201–215, octobre 1998.

Dominique Clément, Joëlle Despeyroux, Th. Despeyroux et Gilles Kahn : A simple applicative language : Mini-ML. *In LFP '86 : Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 13–27, août 1986.

Pascal Cuoq et Julien Signoles : Experience report : Ocaml for an industrial-strength static analysis framework. *In ICFP'09 : Proceedings of the 14th ACM International Conference on Functional Programming*, volume 44(9) de *SIGPLAN Not.*, pages 281–286, septembre 2009.

Olivier Danvy : A simple solution to type specialization. *In ICALP'98 : 25th International Colloquium on Automata, Languages and Programming*, volume 1443 de *Lecture Notes in Computer Science*, pages 908–917. Springer, 1998.

Roland Ducournau : Implementing statically typed object-oriented programming languages. *ACM Computing Surveys*, 4(43), 2011. To appear.

Martin Elsman : Type-specialized serialization with sharing. *In TFP'05 : Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming*, pages 47–62, septembre 2005.

Fabrice Le Fessant et Samuel Mimram : ocaml-memprof : Ocaml heap profiling. `http://www.pps.jussieu.fr/~smimram/`, 2004.

Robert B. Findler et Matthias Felleisen : Contracts for higher-order functions. *In ICFP'02 : Proceedings of the 7th ACM International Conference on Functional Programming*, volume 37(9) de *SIGPLAN Not.*, pages 48–59, septembre 2002.

Jun Furuse et Pierre Weis : Entrées-sorties de valeurs en Caml. *In JFLA'00 : Journées Francophones des Langages Applicatifs*. INRIA, janvier 2000.

Richard P. Gabriel : *Performance and Evaluation of Lisp Systems*. MIT Press, 1985.

Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides : *Design Patterns*. Addison-Wesley, 1995.

Jacques Garrigue : Programming with polymorphic variants. *In ML'98 : Proceedings of the ACM Workshop on ML*, septembre 1994.

Jim Gettys, Philip L. Karlton et Scott McGregor : The X Window System, Version 11. *Softw., Pract. Exper.*, 20(S2):35–67, 1990.

The GHC Team : *The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.0.1*, novembre 2010.

Adele Goldberg et David Robson : *Smalltalk-80 : The Language and Its Implementation*. Addison-Wesley, 1983.

Benjamin Goldberg et Michael Gloger : Polymorphic type reconstruction for garbage collection without tags. *SIGPLAN Lisp Pointers*, V(1):53–65, 1992.

James Gosling, Bill Joy, Guy Steele et Gilad Bracha : *The Java Language Specification (3rd Edition)*. Addison-Wesley, 2005.

Robert Harper et Greg Morrisett : Compiling polymorphism using intensional type analysis. *In POPL'95 : Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, pages 130–141, janvier 1995.

Grégoire Henry, Michel Mauny et Emmanuel Chailloux : Typer la désérialisation sans sérialiser les types. *Technique et Science Informatiques*, 26(9):1067–1090, 2007.

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

Gérard Huet : *Résolution d'équations dans des langages d'ordre 1,2,..., ω*. Thèse de doctorat, Université Paris 7, 1976.

IEEE 754 : *Standard for Floating-Point Arithmetic*. IEEE, 2008.

William Kahan : Why do we need a floating-point arithmetic standard ? Rapport technique, University of California, Berkeley, 1981.

Andrew J. Kennedy : Pickler combinators. *Journal of Functional Programming*, 14 (6):727–739, 2004.

Xavier Leroy : *Typage polymorphe d'un langage algorithmique*. Thèse de doctorat, Université Paris 7, 1992.

Xavier Leroy : Applicative functors and fully transparent higher-order modules. *In POPL'95 : Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, pages 142–153, janvier 1995.

Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy et Jérôme Vouillon : *The Objective Caml system release 3.12*, juin 2010.

Xavier Leroy et Hervé Grall : Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009.

Xavier Leroy et Michel Mauny : Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1993.

Sheng Liang et Gilad Bracha : Dynamic class loading in the java virtual machine. *In OOPSLA '98 : Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 36–44, octobre 1998.

John McCarthy : Recursive functions of symbolic expressions and their computation by machine, Part I. *Communication of the ACM*, 3(4):184–195, 1960.

John McCarthy : History of LISP. *SIGPLAN Notice*, 13(8):217–223, 1978.

Microsoft : The .NET Common Language Runtime. `http://msdn.microsoft.com/net/`.

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

Yasuhiko Minamide, Greg Morrisett et Robert Harper : Typed closure conversion. *In POPL'96 : Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 271–283, janvier 1996.

Raphaël Montelatici : *Functional languages, typing and interoperability : Objective Caml on .NET*. Thèse de doctorat, Université Paris 7, 2007.

Greg Morrisett, David Walker, Karl Crary et Neal Glew : From System F to Typed Assembly Language. *Transaction on Programming Languages and Systems*, 21 (3):527–568, mai 1999.

Alan Mycroft : Polymorphic type schemes and recursive definitions. *In ISP'84 : Proceedings of the 6th International Symposium on Programming*, volume 167 de *Lecture Notes in Computer Science*, pages 217–228. Springer, 1984.

Object Management Group : The Common Object Request Broker : Architecture and Specification, 1995.

Martin Odersky : The Scala language specification version 2.7 DRAFT, 2009.

Martin Odersky et Philip Wadler : Pizza into Java : translating theory into practice. *In POPL'97 : Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 146–159, janvier 1997.

Oracle : Java^{TM} object serialization. `http://download.oracle.com/javase/7/docs/technotes/guides/serialization/`, 2005.

Gordon Plotkin : A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.

François Pottier et Yann Régis-Gianas : Stratified type inference for generalized algebraic data types. *In POPL'06 : Proceedings of the 33rd ACM Symposium on Principles of Programming Languages*, pages 232–244, janvier 2006.

François Pottier et Didier Rémy : The Essence of ML Type Inference. *In* Benjamin C. Pierce, éditeur : *Advanced Topics in Types and Programming Languages*, chapitre 10, pages 389–489. MIT Press, 2005.

Daniel De Rauglaudre : *Camlp4 - Tutorial, version 3.07*, septembre 2003. URL `http://caml.inria.fr/pub/docs/tutorial-camlp4/`.

Didier Rémy et Jérôme Vouillon : Objective ML : An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, janvier 1998.

RFC 4506 : *XDR : External Data Representation Standard*, 2006.

RFC 5322 : *Internet Message Format*, 2008.

RFC 5531 : *RPC : Remote Procedure Call Protocol Specification Version 2*, 2009.

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

Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton Van straaten, Robby Findler et Jacob Matthews : Revised^6 report on the algorithmic language scheme. *Journal of Functional Programming*, 19(S1):1–301, 2009.

Guy L. Steele, Jr. : Data representation in PDP-10 MACLISP. MIT AI Memo 421, Massachusetts Institute of Technology, 1977.

Bjarne Stroustrup : *The C++ Programming Language*. Addison-Wesley, troisième édition, 2000.

Hisatotshi Sutou et Eijiro Sumii : Quicksilver/OCaml : A poor man's type-safe and abstraction-secure communication library. Unpublished, 2007.

David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper et Peter Lee : TIL : A type-directed optimizing compiler for ML. *In PLDI'96 : Proceedings of the ACM Conference on Programming Language Design and Implementation*, volume 31(5) de *SIGPLAN Not.*, pages 181–192, mai 1996.

David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper et Peter Lee : TIL : A type-directed optimizing compiler for ML. *In 20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation (1979-1999) : A Selection*, volume 39(4) de *SIGPLAN Not.*, pages 554–567. avril 2004.

R. E. Tarjan : Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

Sam Tobin-Hochstadt et Matthias Felleisen : Interlanguage migration : from scripts to programs. *In OOPSLA '06 : Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming, Systems, Languages, and Applications*, pages 964–974, octobre 2006.

Sam Tobin-Hochstadt et Matthias Felleisen : The design and implementation of Typed Scheme. *In POPL'08 : Proceedings of the 35th ACM Symposium on Principles of Programming Languages*, pages 395–406, janvier 2008.

Andrew P. Tolmach : Tag-free garbage collection using explicit type parameters. *In LFP '94 : Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 1–11, juin 1994.

The Unicode Consortium : *The Unicode Standard, Version 5.2.0*. The Unicode Consortium, 2009. URL `http://www.unicode.org/versions/Unicode5.2.0/`.

Stephen Weeks, Matthew Fluet, Henry Cejtin et Suresh Jagannathan : *The MLton Standard ML compiler*, 2007. URL `http://mlton.org/`.

Stephanie Weirich : Higher-order intensional type analysis. *In ESOP'02 : Proceedings of the 11th European Symposium on Programming*, pages 98–114, 2002.

Stephanie Weirich : Type-safe run-time polytypic programming. *Journal of Functional Programming*, 16(10):681–710, novembre 2006.

Andrew K. Wright : Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, décembre 1995.

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

Hongwei Xi, Chiyan Chen et Gang Chen : Guarded recursive datatype constructors. *In POPL'03 : Proceedings of the 30th ACM Symposium on Principles of Programming Languages*, page 224–23, janvier 2003.

Jeremy Yallop : Practical generic programming in OCaml. *In ML'07 : Proceedings of the ACM Workshop on ML*, pages 83–94, septembre 2007.

Zhe Yang : Encoding types in ML-like languages. *In ICFP'98 : Proceedings of the 3rd ACM International Conference on Functional Programming*, volume 34(1) de *SIGPLAN Not.*, pages 289–300, septembre 1998.

Index

Règles d'évaluation

Règles de typage

Système de réécriture de contraintes

Résumé

Le typage statique des langages de programmation garantit des propriétés de sûreté d'exécution des programmes et permet l'usage de représentations de données dénuées d'informations de types. En présence de primitives de (dé)sérialisation, ces données brutes peuvent invalider les propriétés apportées par le typage statique. Il est alors utile de pouvoir tester dynamiquement la compatibilité des données lues avec le type statique attendu.

Cette thèse définit, dans le cadre des langages de programmation basés sur un système de types avec polymorphisme paramétrique et utilisant une représentation uniforme des données, une notion de compatibilité d'un graphe mémoire (désérialisé) avec un type ; cette notion s'exprime sous la forme de contraintes de types sur les nœuds du graphe mémoire. Cette formalisation permet de construire un mécanisme de résolution de contraintes par réécriture, puis un algorithme de vérification de compatibilité d'un graphe mémoire avec un type. Les propriétés de correction et de complétude de l'algorithme obtenu sont étudiées en présence de types algébriques, de données modifiables, de cycles et de valeurs fonctionnelles. Cette thèse propose également un prototype pour le compilateur OCaml.

Abstract

Static typechecking in programming languages allows strong safety properties—*well-typed programs don't go wrong*—and avoids costly dynamic type checking and runtime type representation. In this setting, simple (un)marshalling primitives may break invariants assumed by the compiler and void the warranty. In order to preserve type-safety, some dynamic type checking of unmarshalled data is required.

This thesis defines, for programming languages based on parametric polymorphism and uniform data representation, a notion of compatibility between (unmarshalled) memory graphs and types. This notion is expressed as constraints over nodes of the memory graph. The constraint representation allows to build a constraint solver based on a rewriting system, and then to build an algorithm to check the compatibility of a memory graph with a type. Correction and completeness of the algorithm are studied in presence of algebraic data types, mutable data, cycles and functional values. This thesis also provides a prototype tailored for the OCaml compiler.