Un système de types pragmatique pour la vérification déductive des programmes

Léon Gondelman

NNT : 201XSACLSXXX

Thèse de doctorat de l'Université Paris-Saclay préparée à Université Paris-Sud

Ecole doctorale n°*XXX*
Sciences et Technologies de l'Information et de la Communication
Spécialité de doctorat : Informatique

par

M. Léon Gondelman

Un système de types pragmatique pour la vérification déductive des programmes

Thèse présentée et soutenue à Pôle Universitaire d'Ingénierie d'Orsay, le 13 décembre 2016.
Composition du Jury :

Mme.	Sandrine Blazy	Professeur Université Université de Rennes 1	(Rapporteure)
M.	Giuseppe Castagna	Directeur de recherche CNRS, Université de Paris 7	(Examinateur)
M.	Jean Goubault-Larrecq	Professeur LSV, CNRS, ENS Cachan	(Examinateur)
M.	Jean-Christophe Filliâtre	Directeur de recherche CNRS, Université Paris Sud	(Directeur de thèse)
M.	Andrei Paskevich	Maître de conférences Université Paris Saclay	(Directeur de thèse)
M.	Jorge Sousa Pinto	Professeur Universidade do Minho	(Rapporteur)
M.	François Pottier	Directeur de recherche INRIA Paris	(Examinateur)

Un système de types pragmatique pour la vérification déductive des programmes

Léon Gondelman

13 décembre 2016

À Samuel Bach

Table des matières

Table des figures

Remerciements

Mes remerciements vont avant tout à mes directeurs de thèse, Jean-Christophe Filliâtre et Andrei Paskevich. Leur présence dans ma vie académique et dans ma vie tout court pendant ces dernières années a été précieuse et je n'ai pas de mots pour exprimer ma gratitude. Je peux seulement dire, en toute humilité, que j'espère un jour me rapprocher de la hauteur tant de leurs qualités de chercheurs que de leurs qualités morales et humaines. Et si je devais dire quels sont les souvenirs les plus précieux de ma vie de thésard, ce seront certainement les discussions matinales avec Jean-Christophe au coin café devant le tableau avant que tout le monde arrive et les discussions avec Andrei au labo jusqu'à l'œil de nuit, quand, après avoir trouvé les bonnes définitions, on descendait par les forêts d'Orsay en parlant de la poésie russe.

Je tiens à remercier Sandrine Blazy et Jorge Sousa Pinto d'avoir accepté d'être les rapporteurs de cette thèse et de faire partie du jury lors de la soutenance.

Je remercie Giuseppe Castagna, François Pottier ainsi que Jean Goubault-Larrecq qui me font également cette honneur de faire partie de mon jury. La présence de M. Goubault-Larrecq me tient particulier au cœur, car ma vie scientifique a commencé en grimpant les théorèmes ardus de son cours du lambda-calcul sans lequel je ne serai jamais arrivé là où je suis aujourd'hui.

Je tiens à exprimer ma gratitude envers tous les membres de l'équipe VALS avec qui j'ai partagé tant de bons moments et qui m'ont appris tant de choses. Mes remerciements vont en particulier à mes deux co-bureaux, Martin Clochard, qui était toujours là pour m'expliquer une solution particulièrement astucieuse pour un problème du projet Euler et mon merveilleux petit frère académique Mário Pereira qui m'a tant soutenu pendant la rédaction de ma thèse et la préparation de la soutenance.

Un merci à part aux personnes du service administratif de l'ENS et du LRI qui m'ont aidé à gérer tous les dossiers, les papiers, les formulaires, etc, et cela malgré mon absence d'organisation. Je tiens particulièrement à remercier Isabelle Delais à l'ENS et Régine Bricquet au LRI pour leur accueil et leur soutien sans faille.

Enfin, mes remerciements vont à ma famille, à mes parents en Lettonie, à mon grand frère à New-York et ma grand-mère en Israël ainsi qu'à tous mes amis dont le soutient fut précieux pendant ces trois années de thèse.

Je tiens à remercier particulièrement mon amie Éléna Vladimirska qui m'a fait découvrir la langue française lorsque j'étais encore un lycéen à Riga. Grâce à elle, mon rêve d'écrire un jour un livre en français s'est enfin réalisé.

Je remercie également mon ami Bernard Randé dont l'aide précieuse m'a permis de découvrir et d'avancer dans l'apprentissage des mathématiques.

Enfin, je tiens à remercier mon ami Samuel Bach à qui j'ai le bonheur de dédier cette thèse et sans qui rien de tout cela n'aurait été possible.

1 Introduction

1.1 Question de confiance

Lorsque j'ai assisté au cours de compilation donné à l'École Normale Supérieure, j'ai été bouleversé par l'idée qu'un compilateur, outil traduisant un programme informatique depuis un langage de haut niveau vers un langage plus élémentaire, est lui-même un programme. Cette mise en abîme m'a fait prendre conscience de l'importance de s'assurer que les programmes que l'on écrit soient corrects. Qu'un programme traduit par le compilateur de `C` vers un langage assembleur comporte un bug, est, certes, embêtant, mais on peut y remédier facilement : il suffit de débugguer le programme `C`. Imaginons maintenant que c'est le compilateur lui-même qui comporte un bug : il traduit, par exemple, l'opération d'addition `C` en l'opération de soustraction assembleur. Un tel bug est d'une autre envergure, car il sera une source éternelle d'erreurs compromettant un grand nombre des programmes `C`, qu'ils soient corrects ou pas. On pourrait croire que l'exemple est trop naïf, mais la réalité est que les compilateurs sont parmi les programmes les plus sophistiqués autant par le nombre de lignes de code que par la complexité de leur conception. Ainsi peut-on lire dans l'introduction du *Livre au dragon* [1] :

> « *Il est en fait si difficile d'écrire un compilateur optimisant que nous osons affirmer qu'il n'existe aucun compilateur optimisant complètement sans erreur ! C'est pourquoi l'objectif principal lors de l'écriture d'un compilateur doit être qu'il soit correct.* »

L'une des raisons pour lesquelles il est difficile de produire des logiciels corrects est que la suite des instructions que le programmeur envoie à l'ordinateur sera exécutée par celui-ci exactement telle quelle, ne laissant place à aucune interprétation ambigüe. La moindre modification dans le texte du programme risque de modifier le résultat. Lors d'une conférence au Collège de France à laquelle j'ai assisté pendant ma thèse, l'informaticien Gérard Berry a donné une illustration merveilleusement parlante de ce propos : supposons que l'on souhaite déplacer un robot sur une grille d'un point à un autre en évitant des obstacles. Pour faire actionner le robot, on dispose d'un langage très simple de quatre lettres « `B` », « `D` », « `G` », « `H` » permettant chacune de déplacer le robot de sa position courante dans dans l'une des quatre directions (en bas, à droite, à gauche, en haut). Pour déplacer un robot d'un point **A** au point **B** sur la grille, il

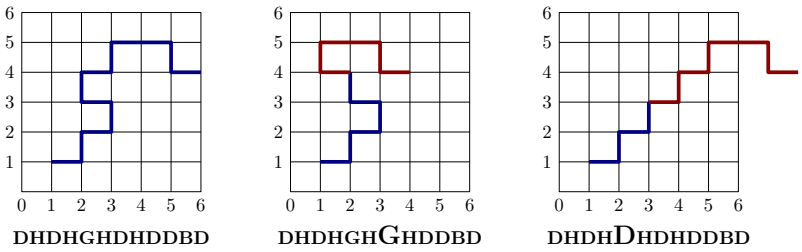


FIGURE 1.1 – Déplacement du robot sur une grille

suffit d'écrire une séquence de déplacements élémentaires et de rentrer les coordonnées la position initiale **A**. Par exemple, imaginons que l'on souhaite déplacer le robot du point $(1, 1)$ vers le point $(6, 4)$ et que le chemin optimal est donné par la séquence de douze lettres du programme `DHDHGHDHDDBD` (dessin à gauche dans la figure 1.1). Voyons ce qui se passe si l'on modifie une seule lettre, par exemple en remplaçant la septième lettre (`D`) par `G` (dessin au centre dans la figure 1.1). On voit que ce petit changement a pour conséquence d'envoyer le robot dans un endroit qui n'est pas le point de destination. Pire encore, si l'on remplace la cinquième lettre (`G`) par `D` (dessin à droite dans la figure 1.1), le robot sort complètement de la grille et on n'est même pas en mesure de décrire son comportement qui peut avoir des effets imprévisibles. Mais dans les deux cas, on voit très bien à quel point il est facile d'introduire un bug et à quel point il peut être difficile de le remarquer *a posteriori*.

L'exemple du petit robot qu'il s'agit de déplacer sur la grille pourrait faire sourire le lecteur, tant l'exemple est simple, mais lorsque ce robot devient une fusée, qu'il s'agit d'envoyer dans l'espace et que cette fusée explose quarante secondes après le décollage suite à un dépassement d'entier dans les registres mémoire du programme de pilotage, on comprend que le bug informatique peut avoir des conséquences néfastes et réelles. L'échec du vol numéro 501 de l'Ariane 5 est seulement un exemple parmi tant d'autres : il suffit de consulter la page Wikipédia anglaise en question [1] pour voir la liste tristement longue des bugs informatiques majeurs causant des pertes de grandes sommes d'argent et de vies humaines. La question comment développer des logiciels corrects est donc cruciale. Peut-on obtenir des programmes dont on est certain qu'ils ne contiennent aucun bug ? Si oui, comment ? Sinon, quels sont des niveaux de confiance que l'on peut espérer avoir en des logiciels ?

1. https://en.wikipedia.org/wiki/List_of_software_bugs

Pour commencer, écrire des programmes corrects vis-à-vis de leur spécification requiert de l'expérience, de la culture informatique générale et un bagage de connaissances qui ne viennent qu'avec des années de pratique. On peut déjà considérablement augmenter la confiance dans les programmes que l'on conçoit si, avant de les coder, on prend le temps de bien comprendre le problème et de choisir des structures de données adaptées à la solution. Et lorsque l'on passe au développement, on augmente encore cette confiance si l'on commente soigneusement son code, le fait relire par des collègues, effectue des tests et, plus généralement, si l'on suit tous les bons conseils que donnent les ouvrages comme *The Practice of Programming* de Kernighan et Pike [30] ou *Programming Pearls* de Bentley [7]. Par ailleurs, le dernier point mentionné, c'est-à-dire, effectuer les tests, est particulièrement important : un jeu de tests élaboré permet de trouver et corriger des bugs dès les premières étapes du développement. En termes simples, le test [35] consiste à choisir un ensemble de valeurs que l'on passe une par une en entrée du programme, pour observer si la valeur en sortie correspond au résultat attendu. Notons toutefois que le test de programmes est un sujet de recherche à part entière, qu'il existe plusieurs approches différentes (« black box testing », « white box testing », test unitaire, etc.) et que construire un bon jeu de tests demande beaucoup d'expertise. Aussi utile soit-il, le test reste cependant limité en tant qu'outil pour montrer la correction d'un programme. La raison est que le nombre d'états possibles d'exécution pour un programme même très petit est souvent infini, alors que le programme peut être testé seulement sur un nombre *fini* de valeurs. Et même lorsque l'on a l'impression que le test permet de couvrir tous les cas (ou, du moins, tous les cas critiques), il reste toujours la question de savoir en quoi cette impression est elle-même juste. En résumé, et pour citer Edsger Dijkstra, « le test permet de montrer la présence de bugs, pas leur absence ». Nous sommes alors obligés de nous tourner vers des *methodes formelles*, un domaine très vaste qui regroupe des approches plus puissantes, mais aussi plus coûteuses que le test.

1.2 Méthodes formelles

Les approches et les techniques que l'on qualifie de méthodes formelles partagent la même idée de modéliser le comportement d'un programme à l'aide des concepts mathématiques dans le but de montrer la correction d'un programme vis-à-vis de sa spécification. Il existe de nombreuses approches formelles, basées sur des formalismes différents. On peut citer notamment la *vérification de modèles* (en anglais *model checking*) [16], l'*interprétation abstraite* [18], le *raffinement* [4], l'utilisation *des systèmes de types* [37] et *la vérification déductive des programmes* [22]. Parmi toutes les approches formelles, on peut distinguer celles qui s'appuient sur une *analyse statique* : cela veut dire que, contrairement au test, ces approches visent à établir la correction d'un programme, sans exécuter son code. L'idée de ces approches est que, lorsque l'on est capable de donner une description mathématique d'un langage de programmation

(typiquement, en termes d'une sémantique formelle, par exemple, une sémantique opérationnelle, dénotationnelle, par un ensemble d'équations, etc.), il devient possible de connaître précisément ou, du moins, d'approximer le comportement de chaque instruction sans avoir besoin de l'exécuter. Pour un aperçu détaillé du panorama des méthodes formelles, le lecteur pourra consulter par exemple les ouvrages de Monin [32] et Aleida et *al.* [2]. En ce qui concerne cette thèse, ce sont les deux dernières approches formelles mentionnées, le typage et la vérification déductive, qui nous intéressent particulièrement, la vérification déductive étant le contexte général de notre travail et le typage étant notre outil principal. Avant de passer à la problématique et aux contributions de cette thèse, nous présentons brièvement ces deux approches.

1.2.1 Systèmes de types

Un système de types est une forme d'analyse statique utilisée pour montrer, le plus souvent d'une manière décidable, l'absence d'une certaine catégorie d'erreurs bien définie, en classifiant les phrases de programme selon les types de valeurs qu'elles représentent. Explicitons certains points de cette définition un peu succincte.

Le premier aspect important du typage concerne l'utilisation des types en tant qu'objets syntaxiques. Cela signifie simplement que le typage manipule *explicitement* des classifications que le programmeur garde généralement en tête ou laisse en commentaires. Par exemple, une expression entière comme $5 + 42$ va avoir le type `int`, une expression booléenne $4^2 < 42$ va avoir le type `bool`, une fonction `fib` qui calcule les nombres de Fibonacci va avoir le type $\texttt{int} \to \texttt{int}$, etc. En utilisant les informations sur les types uniquement, le système vérifie alors la *cohérence* de l'utilisation des types dans les expressions composées, telles que les appels de procédures, les instructions de contrôle, etc., selon un *ensemble de règles de typage* où chaque règle décrit le typage d'une catégorie syntaxique donnée. Sans rentrer dans les détails techniques, insistons sur le caractère *compositionnel* des règles du typage : pour qu'une expression donnée soit bien typée, il faut que toutes ses sous-expressions soient bien typées et que leurs typages soient cohérents entre eux. Par exemple, dans une conditionnelle de la forme $\texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, il est raisonnable de la typer selon la règle que b doit avoir le type `bool` et que les expressions e_1 et e_2 de chaque branche doivent avoir le même type. De même, dans un appel de fonction comme $f\ 42\ \texttt{true}$, il est raisonnable de la typer selon la règle que le type de chaque argument factuel doit être le même que celui du paramètre formel correspondant, c'est-à-dire que le type de f doit être $\texttt{int} \to \texttt{bool} \to \tau$. L'intérêt de manipuler les types explicitement est que les expressions d'un programme qui n'ont aucun sens calculatoire comme $1 + \texttt{fib}$, $42\ \texttt{true}$, etc., vont alors être rejetées par le typage comme erronées, car elles ne respectent aucune règle du typage.

Un autre aspect du typage important est qu'il peut être considéré non seulement comme outil d'analyse statique, mais qu'il peut faire l'objet d'une étude théorique. C'est d'ailleurs l'aspect du typage qui nous importe certainement le plus ici, car le but de cette thèse est d'explorer des solutions qu'une approche à base de systèmes de types

peut apporter à la vérification des programmes. Plus spécifiquement, chaque fois que nous allons étudier un système de types, nous allons procéder en trois temps : d'abord, nous donnons la *syntaxe abstraite* et la *sémantique opérationnelle* d'un langage de programmation ; ensuite, nous équipons ce langage par un système de types caractérisé par un ensemble des règles de typage ; et enfin, nous montrons la *correction* du typage, c'est-à-dire que les programmes bien typés ont les propriétés qui nous intéressent. Remarquons que pour montrer la correction du système des types nous allons utiliser l'approche syntaxique de Wright et Felleisen [44] où une *sémantique opérationnelle à petits pas*, définie inductivement comme une relation entre des configurations de départ et une configurations d'arrivée, décrit toutes les manières possibles de réaliser *un pas* d'évaluation, ce qui permet de montrer la correction du typage par induction sur la relation d'évaluation.

Enfin, il peut paraître au lecteur comme trop restrictif qu'un système de types a pour objectif de montrer l'absence non pas de tous les bugs, mais seulement d'une certaine catégorie. Néanmoins, on comprend mieux l'importance et les enjeux pragmatiques derrière le typage, lorsque l'on considère le caractère *décidable* de la plupart des systèmes de typage utilisés en pratique. En effet, suite aux travaux de Gödel, Church et Turing, dans les années trente du dernier siècle, on sait que la décidabilité et la complétude sont deux choses irréconciliables. Cela a pour conséquence que, pour tout langage de programmation quelque peu réaliste, trouver toutes les erreurs d'exécution possibles est un problème indécidable, c'est-à-dire qu'il n'existe pas d'algorithme qui permettrait de décider en temps fini si un programme arbitraire est correct ou pas. Or, lorsque l'on renonce à la complétude, on retrouve alors la possibilité de concevoir un algorithme qui décide si un programme est exempt ou pas de certains types d'erreurs. On comprend mieux maintenant pourquoi des langages de programmation comme Java, C, C++, OCaml sont tous équipés d'un système de types : un grand nombre d'erreurs peuvent être détectées automatiquement par le typage pendant la phase de compilation, en avertissant le programmeur par un message instructif sur l'endroit et l'origine des erreurs trouvées.

Enfin, notons que nous n'abordons pas ici de nombreux aspects du typage en soit très importants mais orthogonaux à notre propos, tels que la correspondance Curry-Howard, l'inférence de types, le sous-typage, la théorie des types dépendants, etc. Le lecteur intéressé pourrait consulter l'excellent ouvrage « *Types and Programming Languages* » de Benjamin Pierce [37, 38] .

Comme nous venons de le remarquer, le typage renonce de la complétude au profit de la décidabilité. Nous présentons maintenant brièvement la *vérification déductive de programmes*, une méthode formelle qui, bien que plus coûteuse qu'un test et moins décidable que le typage, peut apporter un degré maximal de confiance dans la correction d'un programme.

1.2.2 Vérification déductive des programmes

L'idée clé de la vérification déductive est en fait très simple : puisque, intuitivement, un programme correct est un programme qui fait exactement ce qu'il est censé faire selon les intentions du programmeur, il suffit, d'une part, de rendre ces intentions explicites dans un langage mathématique et, d'autre part, d'exprimer la correction d'un programme par un ensemble d'énoncés mathématiques lesquels, lorsqu'ils sont vrais, impliquent la correction du programme vis-à-vis des intentions de l'utilisateur. La figure 1.2 représente ces trois étapes de la vérification déductive schématiquement. Ces idées, exprimées pour la première fois dans le travail de Turing, *Checking a large*

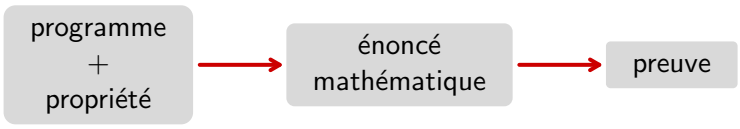


FIGURE 1.2 – Vérification déductive des programmes.

routine [43], pour montrer la correction d'un petit programme calculant la factorielle, soulèvent au moins trois questions.

La première question est de savoir quelle est la manière adéquate pour exprimer l'interaction entre la spécification et l'ensemble des instructions d'un programme. Une première tentative d'élever les idées de Turing au rang d'un cadre formel, dans lequel on pourrait raisonner sur la correction de programmes d'une manière systématique, a été faite par Floyd [25] dans son travail *Assigning meaning to programs*. L'interaction entre la spécification et le code est exprimée à l'aide des organigrammes, mais l'idée reste celle de Turing : à chaque instruction on associe une condition qui doit être vraie avant que l'exécution soit exécutée et une assertion qui doit être vraie après l'exécution si la condition initiale a été respectée. En faisant une preuve par induction sur le nombre d'instructions exécutées, il devient alors possible de démontrer des propriétés du programme de la forme : « si les valeurs initiales des variables et des arguments en entrée satisfont une propriété R_1, alors les valeurs des variables et le résultat en sortie satisfont une propriété R_2 ». Par ailleurs, la preuve de terminaison du programme devient possible : il suffit de montrer qu'une mesure décroît strictement et d'une manière finie à chaque transition possible dans le flot de contrôle du programme. Mais c'est surtout le travail de Hoare *An axiomatic basis for computer programming* [26], paru deux ans après celui de Floyd, qui a donné à la vérification déductive le cadre formel que l'on utilise encore aujourd'hui et que l'on appelle *la logique de Hoare*. Le concept de base de cette logique consiste à incarner les idées de Turing à l'aide des *triplets* de la forme

$$\{\ P\ \}\ s\ \{\ Q\ \}$$

qui relient l'instruction de programme s, la précondition P et la post-condition Q. Un tel triplet est valide si et seulement si l'exécution de l'instruction s dans un état

quelconque qui satisfait les propriétés de la précondition P résulte en un état qui satisfait les propriétés de la post-condition Q. La correction de programme peut être alors obtenue en assemblant les triplets de chaque instruction selon les *règles d'inférence* qui décrivent les formes possibles des triplets valides pour chaque construction du langage. En procédant de cette manière-là, l'énoncé mathématique qui se porte garant de la correction d'un programme devient simplement la validité du triplet

$$\{\ P\ \}\ S\ \{\ Q\ \}$$

où S correspond au programme en entier, P est l'ensemble des hypothèses sur les entrées et l'état avant son exécution et Q constitue les propriétés que l'on souhaite démontrer pour la valeur et l'état final de l'exécution.

La logique de Hoare permet de poser un cadre formel pour construire les conditions de vérifications de l'énoncé de correction, mais elle ne fournit pas de méthode efficace pour le faire. En effet, il est possible d'appliquer les règles de Hoare à chaque instruction de programme, en construisant pas à pas la dérivation du triplet pour le programme en entier, mais cela nécessite de donner toutes les pré- et des post- conditions des triplets intermédiaires. Cette manière de procéder est trop fastidieuse pour construire l'énoncé de correction de programmes même relativement petits. La deuxième question est donc de savoir comment construire l'énoncé de correction d'une manière plus efficace. La réponse à cette question a été apportée en 1975 par Dijsktra [21] qui y a introduit le *calcul de plus faibles préconditions*, une technique qui a permis d'apporter beaucoup d'automatisation dans le processus de la construction de l'énoncé de correction. L'observation de Dijsktra était que seulement *certains* endroits dans le programme nécessitent que l'utilisateur écrive des assertions logiques intermédiaires (typiquement, les boucles), les assertions de la plupart des triplets intermédiaires pouvant être inférées et combinées d'une manière automatique. Partant de cette observation, l'idée de Dijsktra est de calculer récursivement, à partir du programme S et la post-condition Q, une précondition $wp(S, Q)$ qui exprime la plus faible condition logique sur les états initiaux de l'exécution de S telle que les états finaux vérifient la post-condition Q. L'énoncé de correction pour le programme S est alors exprimé par une implication

$$P \quad \Longrightarrow \quad wp(S, Q)$$

Le calcul de plus faibles préconditions de Dijkstra permet ainsi de construire des conditions de vérifications d'une manière efficace et automatisable. Cependant, pour prouver un programme, il reste à démontrer l'énoncé mathématique de sa correction. La troisième question est donc de savoir quelles sont les approches plus efficaces que sortir une feuille de papier et un crayon pour faire la preuve. Une approche possible est d'utiliser des assistants de preuve tels que Coq [9], Isabelle [28] ou PVS [36], ce qui apporte un degré de confiance inégalé dans la preuve, mais demande toujours à l'utilisateur de mener lui-même le raisonnement. Cependant, on peut remarquer qu'une grande partie des conditions de vérification concernent les propriétés de sûreté

(l'absence de débordement des calculs arithmétiques, l'accès aux tableaux à l'intérieur des bornes, etc.) pour lesquelles la preuve automatique devient envisageable. D'autre part, les démonstrateurs automatiques ne cessent d'être perfectionnés, et permettent de résoudre des problèmes de plus en complexes, au moins pour certaines théories bien définies. On peut citer notamment les démonstrateurs automatiques de la famille des solveurs SMT (*Satisfiability Modulo Theory*) tels que Alt-Ergo [11], CVC3 [6], Z3 [19]. À l'aide de ces démonstrateurs, il devient possible de prouver automatiquement la correction, y compris la correction fonctionnelle, des programmes aussi complexes que l'algorithme de Strassen de multiplication des matrices [17] ou l'algorithme de Koda-Ruskey de génération des idéaux d'une forêt d'arbres partiellement ordonnées [24].

Bien entendu, pour que le processus de la vérification déductive devienne l'objet d'une technologie, il faut concevoir un outil de vérification qui permette à l'utilisateur d'effectuer chacune des trois étapes résumées dans la figure 1.2. C'est exactement ce qui est fait dans les plateformes de vérifications telles que Boogie [5] et Why3 [23]. Comme le travail de cette thèse concerne directement Why3, nous en donnons d'abord un bref aperçu et un exemple de son utilisation, avant de présenter les contributions de cette thèse.

1.2.3 L'outil de vérification Why3

Why3 est une plateforme de vérification qui donne accès à un ensemble d'outils permettant à l'utilisateur d'implémenter, spécifier formellement et prouver des programmes à l'aide de nombreux démonstrateurs automatiques tels que Alt-Ergo [11], CVC3 [6], z3 [19] etc., et interactifs tels que Coq [9], Isabelle [28] et PVS [36]. Pour cela, Why3 dispose d'un langage de programmation et de spécification, appelé WhyML [23], qui présente de nombreuses similarités avec des langages fonctionnels de la famille ML, telles que du filtrage par motif, des définitions de types algébriques, du polymorphisme et l'inférence de types à la Hindley-Minler, mais aussi des traits impératifs comme la modification de l'état mémoire ou des exceptions.

La correction fonctionnelle des procédures peut être spécifiée au travers de contrats sous forme de pré- et post-conditions, ainsi que des clauses `reads`/`writes` pour indiquer la portion de la mémoire accédée en lecture/écriture. Dans le but de faciliter la preuve de correction des procédures implémentées, leur corps peut être instrumenté par des invariants de boucles, des variants justifiant la terminaison ou encore par des assertions intermédiaires.

Les obligations de preuve sont générées par l'outil à partir du programme ainsi annoté et spécifié en utilisant un calcul de plus faibles pré-conditions. L'un des atouts de Why3 est que ces obligations de preuve peuvent être envoyées à de nombreux démonstrateurs automatiques ou interactifs. Le système permet également d'appliquer des transformations logiques des obligations de preuve avant de les envoyer aux démonstrateurs. Par exemple, lorsqu'une obligation de preuve correspond à une conjonction des buts, il est possible de la transformer en une liste de buts qui peuvent alors être

pris en charge par des démonstrateurs différents.

WhyML peut également être utilisé pour développer des théories logiques, modélisant divers concepts mathématiques ou informatiques, notamment des structures de données. Pour cela, le langage dispose d'une logique de premier ordre avec des types polymorphes, des définitions de types algébriques, des types abstraits, des prédicats (co-)inductifs et des définitions de fonctions récursives. Des exemples de telles théories peuvent être trouvés dans la bibliothèque standard de Why3. Par exemple, on peut trouver la théorie des entiers, des flottants, ou encore la théorie des ensembles et des séquences. Cette bibliothèque standard, ainsi qu'une présentation bien plus détaillée de Why3, est disponible en ligne sur le site du projet `http://why3.lri.fr`.

Un exemple

Pour conclure notre brève description de Why3, illustrons comment on peut y prouver un programme sur un exemple. Pour cela, intéressons-nous au problème suivant connu en littérature anglophone sous le nom de « *the maximum subarray problem* » : étant donné un tableau de n entiers signés, on peut en prendre un segment quelconque et sommer ses éléments ce qui donne un nombre qui peut être négatif ou nul. Le problème est alors de trouver quel est le nombre maximal obtenu par ce procédé. Par exemple, pour le tableau `T` ci-dessous, le programme doit renvoyer l'entier 187, qui est la somme des éléments du segment `T[2..6]`. La solution la plus élégante et la plus ef-

T	31	-41	59	26	-53	58	97	-93	-23	84
	0	1	2	3	4	5	6	7	8	9

ficace à ce problème est donnée par l'algorithme de Kadane.[2] Ce cet algorithme-là que nous allons maintenant coder et prouver en Why3. Écrivons d'abord sa spécification :

```
let maximum_subarray (a: array int): int
  ensures { ∀l h: int. 0 ≤ l ≤ h ≤ length a → sum a l h ≤ result }
  ensures { ∃l h: int. 0 ≤ l ≤ h ≤ length a ∧ sum a l h = result }
= ...
```

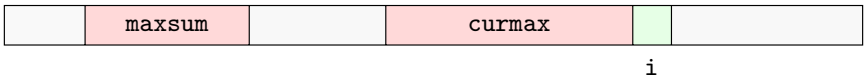
Dans les deux post-conditions ci-dessus, le mot `result` est un réservé qui désigne le résultat renvoyé par le programme et l'expression `sum a l h` donne la somme des éléments du segment `a[l, h[`. La première post-condition exprime le fait que le résultat doit être un entier supérieur ou égal à la somme de tout segment du tableau `a`. La deuxième post-condition garantit que ce résultat est bien lui-même la somme d'un de ses segments. Pour trouver la somme maximale, l'algorithme de Kadane effectue un

2. Il est curieux de noter qu'il n'existe pas de publication de cette solution par son auteur. Elle a été présentée par Jon Bentley dans [8] où le chroniqueur raconte que l'algorithme a été inventé par le statisticien Joe Kadane qui avait trouvé sa solution, à la fois plus élégante et plus efficace que les autres solutions, en une minute.

seul parcours du tableau, en utilisant deux variables mutables, appelées ici `maxsum` et `curmax` qui contiennent initialement chacune zéro.

```
= let maxsum = ref 0 in
  let curmax = ref 0 in
```

L'idée de l'algorithme est que, à l'i-ième étape du parcours, la variable `maxsum` contient la solution du problème pour le tableau `a[0, i[` tandis que la variable `curmax` contient la plus grande somme positive ou nulle d'un segment de `a` dont la borne à droite est égale à l'indice `i-1`. La façon dont les deux variables sont mises à jour dépend alors



de la valeur de la case `a[i]`. On commence par ajouter `a[i]` au contenu de la variable `curmax` et on regarde si `curmax + a[i]` reste une quantité positive ou nulle, auquel cas on ne fait rien. Sinon, lorsque la nouvelle valeur de `curmax` est devenue négative, on remet `curmax` à zéro. Enfin, on regarde si la somme courante ainsi calculée est strictement plus grande que la solution déjà trouvée, auquel cas on assigne à la variable `maxsum` la valeur de `curmax`. Ainsi, une fois le tableau parcouru, il suffit de renvoyer la valeur stockée dans `maxsum`. Ceci correspond au code suivant :

```
for i = 0 to a.length - 1 do
  curmax += a[i];
    if !curmax < 0 then curmax := 0;
    if !curmax > !maxsum then maxsum := !curmax;
done;
!maxsum
```

Si l'on ouvre la session de preuve Why3 maintenant, on obtient cinq obligations de preuve dont une demande à vérifier que l'accès au tableau `a[i]` se fait à l'intérieur des bornes (ce qui est facile à montrer), et les quatre autres correspondent aux post-conditions, faisant distinction par cas selon que le tableau `a` est vide ou pas. Si le tableau est vide, alors sa taille est nulle et donc `a.length - 1` est une quantité négative. Par conséquent, le corps de la boucle sera ignoré, ce qui rend la vérification des deux post-conditions triviale. Il reste à traiter le cas où le tableau n'est pas vide. Dans ce cas, le corps de la boucle sera exécuté au moins une fois. Il nous faut donc fournir les invariants de boucle qui permettront de montrer que les deux post-conditions restent valident lorsque l'exécution du programme sort de la boucle `for`. La difficulté principale ici est que la deuxième post-condition nous demande de montrer l'existence des indices du segment dont les éléments fournissent la somme maximale. Une manière de le montrer est simplement d'exhiber ces indices explicitement. Pour cela, nous allons modifier légèrement le code de l'algorithme en introduisant, juste avant la boucle `for`, trois variables mutables `lo, hi` et `cl` initialisées chacune à zéro :

```
let ghost lo = ref 0 in
let ghost hi = ref 0 in
let ghost cl = ref 0 in
```

L'idée est que variables `lo` et `hi` indiquent les bornes du segment qui donne la somme maximale. Quant à la variable `cl`, elle indique la borne gauche du segment qui donne la somme courante `curmax` (dont la borne droite correspond à l'indice `i` lors de l'i-ième itération de la boucle). Le mot `ghost` qui apparaît dans la déclaration de ces trois variables est un mot réservé de Why3 pour indiquer que ces variables sont introduites uniquement pour le besoin de la spécification et de la preuve. Comme on verra plus tard, cela aide le système de vérifier que leur usage n'aura pas d'influence sur le résultat de l'algorithme.

Pour mettre à jour ces trois variables, on modifie le corps de la boucle `for` :

```
for i = 0 to a.length - 1 do
  curmax += a[i];
  if !curmax < 0
    then begin curmax := 0; cl := i+1 end;
  if !curmax > !maxsum
    then begin maxsum := !curmax; lo := !cl; hi := i+1 end
done;
```

Il nous reste alors à exprimer le sens de ces variables formellement à l'aide d'invariants de boucle. On commence par deux invariants qui relient les variables `lo, hi` et `maxsum` :

```
invariant { ∀l h: int. 0 ≤ l ≤ h ≤ i → sum a l h ≤ !maxsum }
invariant { 0 ≤ !lo ≤ !hi ≤ i ∧ sum a !lo !hi = !maxsum }
```

Ces invariants permettent de montrer la validité des post-conditions, mais si l'on rouvre la session de preuve maintenant, on verra apparaître des nouvelles obligations de preuve correspondant à l'initialisation des invariants et leur préservation par une itération arbitraire de la boucle. La nouvelle difficulté est alors de montrer la préservation de ces invariants. La raison est bien sûr que la valeur de `maxsum` est calculée à partir de la valeur de la variable `curmax`, de même que l'indice `lo` est calculé à partir de la valeur de l'indice `cl`. On a donc besoin d'exprimer également le sens des variables `curmax` et `cl`, ce que l'on fait à l'aide des deux invariants suivants :

```
invariant { ∀l : int. 0 ≤ l ≤ i → sum a l i ≤ !curmax }
invariant { 0 ≤ !cl ≤ i ∧ sum a !cl i  = !curmax }
```

Maintenant, si l'on relance la session de preuve, on pourra voir que toutes les obligations de preuve sont vérifiées instantanément. La figure 1.3 montre l'intégralité du code (on peut voir également les déclarations d'utilisation des modules et des théories de la bibliothèque standard de Why3 tels que les entiers, les tableaux et les références mutables).

```
module Maximum_Subarray

  use import int.Int
  use import ref.Refint
  use export array

  use export array.ArraySum
  (* fournit l'operation sum a l h = la somme des elements a[l,h[
     ainsi que des lemmes sur ses proprietes *)

  let maximum_subarray (a: array int): int
    ensures { ∀l h: int. 0 ≤ l ≤ h ≤ length a → sum a l h ≤ result }
    ensures { ∃l h: int. 0 ≤ l ≤ h ≤ length a ∧ sum a l h = result }
  = let maxsum = ref 0 in
    let curmax = ref 0 in
    let ghost lo = ref 0 in
    let ghost hi = ref 0 in
    let ghost cl = ref 0 in
    for i = 0 to a.length - 1 do
      invariant { ∀l : int. 0 ≤ l ≤ i → sum a l i ≤ !curmax }
      invariant { 0 ≤ !cl ≤ i ∧ sum a !cl i  = !curmax }
      invariant { ∀l h: int. 0 ≤ l ≤ h ≤ i → sum a l h ≤ !maxsum }
      invariant { 0 ≤ !lo ≤ !hi ≤ i ∧ sum a !lo !hi = !maxsum }
      curmax += a[i];
      if !curmax < 0
         then begin curmax := 0; cl := i+1 end;
      if !curmax > !maxsum
         then begin maxsum := !curmax; lo := !cl; hi := i+1 end
    done;
    !maxsum

end
```

FIGURE 1.3 – Réalisation de l'algorithme de Kadane en Why3.

1.3 Contributions de cette thèse

Les contributions de cette thèse sont en rapport direct avec le système de types du langage WhyML. Ce système présente de nombreuses similarités avec celui du langage OCaml telles que le mécanisme d'inférence des types à la Hindley-Milner, le polymorphisme des types, la possibilité de définir des types algébriques récursifs, des types abstraits et des structures de données mutables. Le fait d'associer à chaque expression le type qui la caractérise permet de vérifier au typage la cohérence de l'utilisation des données, notamment lors de l'appel d'une fonction où le paramètre formel et l'argument factuel doivent avoir le même type. En particulier, WhyML offre une certaine garantie de sûreté au sens du slogan de Milner « *Well-typed programs do not go wrong* ». Mais en plus de cette propriété de sûreté standard, le système de types de WhyML garantit des propriétés supplémentaires en rapport avec la preuve des programmes. À cette fin, WhyML attribue à chaque expression, en plus d'un type, des entités caractérisant l'exécution d'une manière plus fine que son type, décrivant notamment les effets de bord liés à la modification de l'état mémoire et à la terminaison. Le premier objectif de cette thèse est d'étudier deux de tels aspects, à savoir le *code fantôme* et le *contrôle statique des alias*.

1.3.1 Code fantôme

Nous avons déjà mentionné le code fantôme dans l'exemple introductif lorsque nous avons introduit dans le programme les trois variables mutables `lo`, `hi` et `cl`. Grâce à ces variables auxiliaires, nous avons pu raisonner sur les indices du segment de la somme maximale. Mis à part le mot clé `ghost` que l'on a utilisé lors de leur déclaration, nous avons modifié ces variables dans le corps de la boucle `for` et nous les avons utilisées à l'intérieur des invariants de boucle au même titre que les variables mutables ordinaires. Faisant cela, nous avons précisé que ces variables « fantômes » étaient introduites uniquement pour les besoins de la spécification et de la preuve, en assurant le lecteur qu'elles ne modifient pas le résultat du programme initial. Ce genre de variables auxiliaires est un exemple d'une notion plus générale, appelée communément *code fantôme* (« *ghost code* » en anglais), qui consiste à introduire dans un programme des données et des calculs supplémentaires dans le but de rendre plus simple le processus de la vérification. L'idée est aussi vieille que la vérification déductive elle-même : à la fin des années soixante, les variables dites « auxiliaires », la forme la plus rudimentaire du code fantôme que l'on vient de voir, étaient déjà utilisées dans le cadre de la vérification des programmes concurrents. D'après Jones [29] et Reynolds [41], ces variables auxiliaires ont été utilisées pour la première fois par Lucas en 1968 [31]. Aujourd'hui, de nombreux outils de la vérification, dont Why3, ont adapté et considérablement élargi l'usage du code fantôme dans divers domaines de la vérification. Le code fantôme y est devenu un code arbitraire qui peut partager

avec le programme les mêmes constructions et les mêmes idiomes du langage. Au delà des variables fantômes, ces outils permettent d'ajouter des paramètres fantômes aux fonctions, d'introduire d'attributs fantômes dans des classes, etc.

Le fait que le code fantôme peut être utilisé au milieu du programme en prenant les apparences du code ordinaire a, certes, l'avantage de faciliter le raisonnement, mais son utilisation peut devenir très vite incorrecte dès lors qu'elle a un impact sur le résultat ou les effets observables de l'exécution du programme. Pour le dire d'une manière affirmative, l'utilisation du code fantôme doit toujours obéir au principe de la *non-interférence* : aussi intriquée ou complexe que soit la manière dont le code fantôme est utilisé, on doit toujours pouvoir l'effacer du programme sans que cela affecte le comportement observable de ce dernier et en particulier le résultat final de son exécution. Sur l'exemple aussi simple que celui que l'on vient de voir, il est facile de se convaincre que le code fantôme n'interfère pas avec le programme initial. Pour des programmes plus complexes, vérifier et garantir la non-interférence du code fantôme avec le reste du programme devient non-trivial. En Why3, la non-interférence du code fantôme est assurée par le typage. À partir de quelques annotations de l'utilisateur telles que l'on vient de voir dans l'exemple introductif, le système de types de WhyML infère pour chaque expression du programme son statut (fantôme ou non-fantôme) qui lui permet de vérifier si ce statut ne rentre pas en conflit avec d'autres attributs inférés statiquement tels que les effets de bord ou la terminaison.

La première contribution de cette thèse consiste à étudier rigoureusement la manière dont le code fantôme est réalisé dans l'outil Why3 : après avoir exposé d'une manière informelle les intuitions et les idées clés de cette approche, on la décrira formellement sur un fragment de WhyML et montrera sa correction.

1.3.2 Contrôle statique des alias

Le deuxième aspect de WhyML abordé dans cette thèse concerne le *contrôle statique des alias*. Pour comprendre la problématique et les motivations du sujet, expliquons d'abord brièvement la manière dont WhyML traite les données mutables. Nous avons déjà rencontré deux types de données mutables provenant de la bibliothèque standard de Why3 : les variables mutables et les tableaux. La bibliothèque standard de Why3 en propose d'autres, par exemple, les tables de hachage (définies dans le module `hashtbl`) et les tableaux en deux dimensions (définis dans le module `matrix`). Malgré les différences apparentes liées au sucre syntaxique, telles que la notation `!v` pour l'accès au contenu d'une variable mutable ou les crochets `a[i]` pour la lecture de l'i-ième case d'un tableau, tous ces types de données mutables sont en réalité définis à l'aide d'un même mécanisme de *types enregistrements*. Chaque type enregistrement décrit une structure de données contenant un nombre fini de champs dont certains peuvent être déclarés comme mutables à l'aide du mot clé `mutable`. Par exemple, les variables mutables sont définies par le type enregistrement

```
type ref 'a = { mutable contents : 'a }
```

et les tableaux par le type

```
type array 'a =
  private   { mutable ghost elts : map int 'a;
                          length : int }
  invariant { 0 ≤ self.length }
```

(ignorons pour l'instant le mot `private` et l'invariant). L'utilisateur peut introduire de nouvelles structures de données mutables à l'aide du même mécanisme. Par exemple, on peut définir la structure de pile par :

```
type stack 'a =  { mutable data: list 'a;
                   mutable size: int }
```

Il est également possible de stocker dans un champ mutable des données qui sont elles-mêmes mutables. Ainsi, on peut définir des types tels que

```
type double_ref 'a =  { mutable rdata: ref 'a   }
type array_ref  'a =  { mutable adata: array 'a  }
```

On peut alors se demander si d'autres façons de combiner des types mutables sont possibles en WhyML : peut-on définir par exemple une liste ou un ensemble dont les éléments sont des données mutables ? La réponse à cette question est négative : les composantes mutables peuvent apparaître seulement dans les données pour lesquelles le système des types de WhyML peut inférer l'identité exacte de chaque pointeur accessible depuis ces données. Pour donner à cette affirmation un sens plus précis et expliquer les motivations derrière la restriction qu'elle impose, il nous faut considérer le phénomène d'*aliasing*, c'est-à-dire la situation où plusieurs pointeurs manipulés dans le programme dénotent une même case mémoire. Le problème qui nous intéresse ici est que lorsqu'un programme comporte un alias, le cadre théorique de la logique de Hoare dans lequel nous nous sommes placés peut conduire à une incohérence. Illustrons ce propos sur l'exemple suivant. Supposons que l'on ait défini une fonction

```
let toto (x y: ref int): unit
  writes   { x, y }
  ensures  { !x = 0 ∧ !y = 1 }
= x := 0; y := 1
```

dont le corps vérifie le triple de Hoare

$$\{\texttt{true}\}\ x := 0;\ y := 1\ \{x = 0 \wedge y = 1\}.$$

Considérons alors un appel à la fonction `toto` où l'on passe deux fois la même variable `z` en argument. On est alors tenté d'en dériver, à partir du triplet valide ci-dessus, le triplet

$$\{\texttt{true}\}\ z := 0; z := 1\{z = 0 \wedge z = 1\}.$$

Or, ce triplet est évidemment invalide. Comment cela a-t-il pu se produire ? La raison est que lorsque l'on avait dérivé le triplet de Hoare pour le corps de la fonction, on

avait appliqué la règle d'affectation

$$\{\, P[x \leftarrow E] \,\}\ x := E\ \{\, P \,\}$$

qui présuppose que l'adresse mémoire à laquelle x se réfère n'a pas d'autres noms symboliques que la variable x. En particulier, on a supposé que les deux paramètres de `toto` réfèrent nécessairement à deux cases mémoires distinctes. Or, l'appel `toto z z` ne vérifie pas cette hypothèse, car les paramètres formels `x` et `y` deviennent aliasés.

De nombreuses approches ont été proposées au cours des dernières décennies pour tenter de résoudre ce problème. L'approche la plus simple consiste simplement à interdire l'aliasing ou à restreindre son utilisation à des formes syntaxiques très simples comme cela est fait dans « *Syntactic Control of Interference* » de Reynolds [40].

Une autre possibilité est de modéliser la mémoire utilisée par le programme explicitement. Par exemple, la mémoire peut être représentée par une référence vers un tableau global infini. On introduit d'abord un nouveau type pour représenter les pointeurs (en particulier, le pointeur nul comme ci-dessous) :

```
type loc
constant null: loc
```

La mémoire peut être alors encodée par une référence vers un dictionnaire des pointeurs vers les données :

```
val m: ref (map loc 'a)
```

que l'on peut manipuler à l'aide de deux fonctions `get_m` et `set_m` :

```
val function get_m (p: loc) : int
  requires { p ≠ null }
  ensures  { result = !m[p] }

val set_m (p: loc) (v: int) : unit
  requires { p ≠ null }
  writes   { m }
  ensures  { !m = set (old !m) p v }
```

Le code de la fonction `toto` devient

```
let toto (x y: loc): unit
  writes   { m }
  ensures  { x != y → get_m x = 0 ∧ get_m y = 1 }
  ensures  { x =  y → get_m y  = 1 }
= set_m x 0; set_m y 1
```

On voit qu'avec le modèle mémoire explicite, nous pouvons maintenant passer à la fonction `toto` deux fois le même argument sans que cela conduise à une incohérence. Néanmoins, le fait d'exprimer l'affectation de chaque variable par une mise à jour de la mémoire globale conduit à un raisonnement *global*, au sens que chaque nouvelle mise

à jour affecte toutes les assertions que l'on a faites sur la mémoire. Dans la logique de Hoare, la règle d'affectation, bien que restrictive, permet au contraire de raisonner sur la mémoire *localement* : seules les assertions qui mentionnent la variable modifiée sont affectées. Les obligations de preuve sont donc plus simples dans le cadre de la logique de Hoare que lorsque l'on modélise la mémoire par un dictionnaire global. Les travaux pionniers de Burstall [12] et Cartwright [13] ont visé d'une manière ou d'une autre à trouver une approche permettant de raisonner sur des programmes avec des pointeurs, sans exclure la possibilité d'aliasing, tout en essayant de rendre le raisonnement plus local que dans le modèle mémoire que l'on vient de voir. En simplifiant beaucoup, on peut dire que la recherche de telles méthodes a pris depuis trois directions différentes, chacune issue d'une communauté de chercheurs différente. Dans la communauté des langages fonctionnels, traditionnellement axée sur l'utilisation des systèmes de type et la gestion la mémoire automatique, Tofte et Talpin [42] ont proposé un partitionnement de la mémoire utilisée par un ensemble de *régions* : à partir des annotations de l'utilisateur, le système de types infère suffisamment d'information pour garantir que les parties de la mémoire représentées par des régions différentes sont indépendantes les unes des autres, permettant ainsi de raisonner sur les propriétés de l'état mémoire à l'échelle des régions.

Dans le contexte de la programmation orientée-objet, le problème a été attaqué sous un angle différent, résultant en un ensemble de techniques telles que l'*ownership* [14, 34, 20] *islands* [27], *balloon types* [3], ou encore *universe types* [33]. L'idée commune à toutes ces techniques n'est pas de modéliser la mémoire d'une manière précise, mais de permettre l'utilisation d'alias contrôlée, en mettant en place des mécanismes d'encapsulation. Basés sur des annotations de type fournies par l'utilisateur, ces mécanismes permettent de déterminer quelles sont les composantes d'un objet qui sont accessibles depuis d'autres objets.

Enfin la troisième approche de la preuve de programmes avec des pointeurs est la *logique de séparation* introduite par Reynolds [39] et activement étudiée depuis. La logique de séparation introduit, d'une part, les formules atomiques de la forme $x \mapsto v$ pour exprimer le fait que x est un pointeur vers une case mémoire contenant la valeur v ; et, d'autre part, elle introduit la conjonction de séparation $\{P * Q\}$ qui n'a pas le même sens que la conjonction$\{P \wedge Q\}$ habituelle : pour que la formule $\{x \mapsto v_1 * y \mapsto v_2\}$ soit vraie il faut non seulement que x pointe vers la valeur v_1 et y pointe vers la valeur v_2, mais que x et y soient des pointeurs disjoints. Si l'on applique l'approche à la spécification de la fonction `toto`, on peut remarquer que, quelles que soient les valeurs v_1, v_2, le triplet logique (correspondant à sa définition)

$$\{x \mapsto v_1 * y \mapsto v_2\}\ x := 0;\ y := 1\ \{x = 0 * y = 1\}.$$

reste valide même lorsque l'on substitue x et y par le même argument z : c'est simplement la précondition $\{z \mapsto v_1 * z \mapsto v_2\}$ qui devient fausse.

La deuxième contribution de cette thèse concerne la manière dont le problème d'aliasing est abordé en WhyML. L'originalité de cette approche réside dans le fait

qu'elle cherche un compromis pragmatique entre expressivité du langage et simplicité des obligations de preuve. Sans exclure totalement les alias dans les programmes, WhyML permet néanmoins de rester dans le cadre de la logique de Hoare. L'idée de base est que, contrairement aux approches dans lesquelles les conditions de séparation apparaissent explicitement dans les obligations de preuve, WhyML opère au préalable un *contrôle statique des alias*, avant même de générer les obligations de preuve : tant que tous les alias manipulés dans le programme sont connus statiquement, ils sont autorisés. Pour contrôler statiquement les alias, l'identité unique de chaque valeur mutable est encodée en WhyML non pas dans son nom symbolique mais dans un type singleton dont il est l'unique habitant. On peut maintenant expliquer pourquoi des structures de données telles que des listes ou des tableaux de références ne sont pas autorisés en Why3. Comme les identités des valeurs mutables sont inférées statiquement à travers leurs régions, le type d'une structure de données doit contenir l'information sur les régions de toutes les composantes mutables qu'elle comprend. Pour les données mutables telles que des listes chaînées ou les tableaux de références, cette analyse n'est pas possible, car on ne connaît pas statiquement le nombre exact de régions qu'il faudrait attribuer.

1.3.3 Raffinement des données

Enfin, le troisième aspect de WhyML abordé dans cette thèse concerne l'obtention des programmes corrects par *raffinement*.

Le raffinement [4] est une technique de vérification déductive qui permet de construire des programmes corrects progressivement, en plusieurs étapes. À l'étape initiale, le programme est représenté d'une manière abstraite par sa spécification. Pour obtenir un code exécutable, les parties abstraites du programme sont remplacées à chaque étape du raffinement, par des parties de plus en plus concrètes (par exemple, en rendant les calculs plus déterministes, en implémentant des structures des données, etc.). L'intérêt de procéder ainsi est que lorsqu'on arrive à montrer que chaque étape préserve toutes les propriétés postulées ou établies aux niveaux précédents, il n'est plus nécessaire de montrer la correction du programme exécutable obtenu à la fin. On dit alors que le code exécutable est *correct par construction*.

À cette dimension « verticale » du raffinement, s'ajoute une dimension « horizontale » : à chaque étape du raffinement, l'abstraction, notamment l'abstraction de structures de données, peut être utilisée en tant qu'outil de développement et de vérification modulaires. En effet, pour utiliser une structure de données, il n'est pas nécessaire de connaître la manière exacte dont elle est réalisée. Pour cela, il suffit souvent de connaître l'ensemble des opérations permettant de manipuler les objets du type de données abstrait. Ceci est également vrai du point de la vérification : il est possible de mener la preuve d'un programme en connaissant seulement le modèle logique des structures de données qu'il utilise. Ce modèle peut être donné sous forme des champs fantômes, des *invariants de liaison* et des contrats qui spécifient le comportement des

opérations permettant de manipuler les objets de la structure de données correspondante.

On peut par ailleurs distinguer deux types de raffinement : le *raffinement algorithmique*, qui consiste à dériver le code exécutable à partir de la spécification (par exemple, dériver le corps d'une procédure à partir de son contrat), et le *raffinement des structures de données* qui consiste à transformer une structure de données abstraite en une structure de donnée plus concrète. C'est ce second type de raffinement que nous allons aborder dans cette thèse. Concrètement, nous nous concentrons sur le raffinement des types enregistrements en Why3. L'intérêt de l'étudier dans cette thèse est double.

Premièrement, des structures de données abstraites peuvent aisément être modélisées par des types enregistrements. Cela est dû au fait que pour prouver le code client manipulant des données d'un type abstrait, il est souvent nécessaire de raisonner sur le modèle logique correspondant. Or, doter un type enregistrement d'un champ fantôme permet justement d'y enchâsser ce modèle logique. Ceci est d'autant plus pratique qu'un tel champ modèle peut être déclaré mutable pour refléter les changements du modèle lorsque le contenu des données change. Par exemple, la structure de données d'ensembles mutables peut être représentée par le type enregistrement suivant :

```
type t = private { ghost mutable repr: set 'a
                           mutable size: int    }
       invariant { size = S.cardinal repr     }
```

Le mot `private` joue ici un rôle important, car sa présence permet d'établir une *barrière d'abstraction* entre le code qui utilise les ensembles mutables et celui qui le raffine. En particulier, la création des objets d'un type privé, ainsi que la modification de leur contenu n'est possible qu'au travers des opérations fournies par le module où le type est défini.

Deuxièmement, les types enregistrements peuvent être vus comme des régions. Comme nous allons le voir, le problème d'aliasing va réapparaître avec l'introduction de nouvelles composantes mutables dans des types enregistrements. Il sera utile de s'appuyer sur le formalisme que l'on développe dans le chapitre sur les régions pour tenir compte du contrôle statique des alias lors du raffinement. La troisième contribution de cette thèse consiste à proposer des conditions suffisantes, mais pas trop restrictives, sur la manière dont les nouvelles régions sont introduites pendant le raffinement pour que celui-ci reste valide. Concrètement, nous allons étendre le formalisme du chapitre sur les régions avec les notions de régions privées et le raffinement des régions et des signatures de fonctions, pour ensuite montrer que le raffinement ainsi défini préserve la relation du typage. Faisant ainsi, nous étudions la question de la correction du raffinement des données uniquement du point de vue du typage, indépendamment des autres conditions de la validité du raffinement telles que la préservation des invariants de types et des contrats de fonctions, c'est-à-dire indépendamment des conditions qui sont plus en rapport avec la préservation des propriétés logiques.

1.4 Plan

Cette thèse est organisée de la façon suivante. Le chapitre 2 introduit un petit langage de programmation, que nous appelons GhostML, qui est un langage fonctionnel à la ML avec un état global, des fonctions récursives et du code fantôme. L'intérêt principal de ce formalisme sera de montrer comment un système de type permet d'utiliser, d'une manière à la fois correcte et expressive, les mêmes types de données et les mêmes fonctions dans le code fantôme et le code réel.

Le chapitre 3 introduit un autre langage, que nous appelons RegML, qui est un langage avec des structures de données ayant des composantes mutables imbriquées à une profondeur bornée. Nous montrons comment le contrôle statique des alias est possible grâce un système de types avec effets et régions.

Ensuite, dans le chapitre 4, nous présentons un certain type de raffinement des données pour lequel nous étendons le formalisme du chapitre précédent avec les notions de régions privées et le raffinement des régions et des signatures de fonctions, et enfin montrons le théorème de préservation du typage par le raffinement ainsi défini.

Enfin, dans le chapitre 5, nous discutons des choix et des limitations de notre travail et donnons quelques directions possibles pour continuer le travail entrepris dans cette thèse.

Bibliographie

[1] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, and Pierre Boullier, *Compilateurs : principes, techniques et outils : cours et exercices*, Sciences sup, Dunod, Paris, 2000, La couv. porte en plus : 2e cycle, écoles d'ingénieurs.

[2] José Bacelar Almeida, Maria João Frade, Jorge Sousa Pinto, and Simão Melo de Sousa, *Rigorous software development - an introduction to program verification*, Undergraduate Topics in Computer Science, Springer, 2011.

[3] Paulo Sérgio Almeida, *Balloon types : Controlling sharing of state in data types*, Proceedings ECOOP'97, LNCS, vol. 1241, Springer, June 1997, pp. 32–59.

[4] Ralph-Johan Back and Joakim von Wright, *Refinement calculus - a systematic introduction*, Undergraduate texts in computer science, Springer, 1999.

[5] Mike Barnett, Robert DeLine, Bart Jacobs, Bor-Yuh Evan Chang, and K. Rustan M. Leino, *Boogie : A Modular Reusable Verifier for Object-Oriented Programs*, Formal Methods for Components and Objects : 4th International Symposium (Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, eds.), Lecture Notes in Computer Science, vol. 4111, 2005, pp. 364–387.

[6] Clark Barrett and Cesare Tinelli, *CVC3*, 19th International Conference on Computer Aided Verification (Berlin, Germany) (Werner Damm and Holger Her-

manns, eds.), Lecture Notes in Computer Science, vol. 4590, Springer, July 2007, pp. 298–302.

[7] J.L. Bentley, *Programming pearls*, Addison-Wesley, 1986.

[8] Jon Louis Bentley, *Algorithm design techniques*, Commun. ACM **27** (1984), no. 9, 865–871.

[9] Yves Bertot and Pierre Castéran, *Interactive theorem proving and program development*, Springer-Verlag, 2004.

[10] Sandrine Blazy and Marsha Chechik (eds.), Lecture Notes in Computer Science, Toronto, Canada, Springer, July 2016.

[11] François Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout, *The Alt-Ergo automated theorem prover*, 2008, http://alt-ergo.lri.fr/.

[12] Rod Burstall, *Some techniques for proving correctness of programs which alter data structures*, Machine Intelligence **7** (1972), 23–50.

[13] Robert Cartwright and Derek Oppen, *The logic of aliasing*, Acta Informatica **15** (1981), no. 4, 365–384.

[14] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad, *Ownership types : A survey*, in Clarke et al. [15], pp. 15–58.

[15] David Clarke, James Noble, and Tobias Wrigstad (eds.), *Aliasing in object-oriented programming : Types, analysis and verification*, Lecture Notes in Computer Science, vol. 7850, Springer, 2013.

[16] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled, *Model checking*, MIT Press, Cambridge, MA, USA, 1999.

[17] Martin Clochard, Léon Gondelman, and Mário Pereira, *The matrix reproved*, in Blazy and Chechik [10].

[18] Patrick Cousot and Radhia Cousot, *Systematic design of program analysis frameworks*, POPL '79 : Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (New York, NY, USA), ACM, 1979, pp. 269–282.

[19] Leonardo de Moura and Nikolaj Bjørner, *Z3, an efficient SMT solver*, http://research.microsoft.com/projects/z3/.

[20] Werner Dietl and Peter Müller, *Object ownership in program verification*, in Clarke et al. [15], pp. 289–318.

[21] Edsger W. Dijkstra, *Guarded commands, nondeterminacy and formal derivation of programs*, Commun. ACM **18** (1975), 453–457.

[22] Jean-Christophe Filliâtre, *Deductive software verification*, International Journal on Software Tools for Technology Transfer (STTT) **13** (2011), no. 5, 397–403.

[23] Jean-Christophe Filliâtre and Andrei Paskevich, *Why3 — where programs meet provers*, Proceedings of the 22nd European Symposium on Programming (Matthias Felleisen and Philippa Gardner, eds.), Lecture Notes in Computer Science, vol. 7792, Springer, March 2013, pp. 125–128.

[24] Jean-Christophe Filliâtre and Mário Pereira, *Producing all ideals of a forest, formally (verification pearl)*, in Blazy and Chechik [10].

[25] Robert W. Floyd, *Assigning meanings to programs*, Mathematical Aspects of Computer Science (Providence, Rhode Island) (J. T. Schwartz, ed.), Proceedings of Symposia in Applied Mathematics, vol. 19, American Mathematical Society, 1967, pp. 19–32.

[26] C. A. R. Hoare, *An axiomatic basis for computer programming*, Communications of the ACM **12** (1969), no. 10, 576–580 and 583.

[27] John Hogg, *Islands : Aliasing protection in object-oriented languages*, SIGPLAN Not. **26** (1991), no. 11, 271–285.

[28] *The ISABELLE system*, http://isabelle.in.tum.de/.

[29] Cliff B. Jones, A.W. Roscoe, and Kenneth R. Wood, *Reflections on the work of C.A.R. Hoare*, 1st ed., Springer Publishing Company, Incorporated, 2010.

[30] Brian W. Kernighan and Rob Pike, *The practice of programming*, Addison-Wesley Longman Publishing Co., Inc., 1999.

[31] P. Lucas, *Two constructive realizations of the block concept and their equivalence*, Technical Report 25.085, IBM Laboratory, Vienna, June 1968.

[32] Jean-François Monin, *Introduction aux méthodes formelles*, Collection technique et scientifique des télécommunications, Hermès science, Paris, 1re éd. chez Masson, sous le titre : Comprendre les méthodes formelles.

[33] Peter Müller and Arsenii Rudich, *Ownership transfer in universe types*, ACM SIGPLAN conference on Object-oriented programming systems and applications (OOPSLA), ACM, 2007, pp. 461–478.

[34] Alan Mycroft and Janina Voigt, *Notions of aliasing and ownership*, in Clarke et al. [15], pp. 59–83.

[35] Glenford J. Myers, Corey Sandler, Tom Badgett, and Todd M. Thomas, *The art of software testing, second edition*, Wiley, June 2004.

[36] S. Owre, J. M. Rushby, and N. Shankar, *PVS : A prototype verification system*, 11th International Conference on Automated Deduction (Saratoga Springs, NY) (Deepak Kapur, ed.), Lecture Notes in Computer Science, vol. 607, Springer, June 1992, pp. 748–752.

[37] Benjamin C. Pierce, *Types and programming languages*, MIT Press, 2002.

[38] Benjamin C. Pierce (ed.), *Advanced topics in types and programming languages*, MIT Press, 2005.

[39] J. C. Reynolds, *Separation logic : a logic for shared mutable data structures*, 17th Annual IEEE Symposium on Logic in Computer Science, IEEE Comp. Soc. Press, 2002.

[40] John C. Reynolds, *Syntactic control of interference*, POPL '78 : Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (New York, NY, USA), ACM, 1978, pp. 39–46.

[41] John C. Reynolds, *The craft of programming*, Prentice Hall International series in computer science, Prentice Hall, 1981.

[42] Mads Tofte and Jean-Pierre Talpin, *Implementation of the typed call-by-value lambda-calculus using a stack of regions*, Symposium on Principles of Programming Languages, 1994, pp. 188–201.

[43] Alan Mathison Turing, *Checking a large routine*, Report of a Conference on High Speed Automatic Calculing Machines (Cambridge), Mathematical Laboratory, 1949, pp. 67–69.

[44] Andrew K. Wright and Matthias Felleisen, *A syntactic approach to type soundness*, Information and Computation **115** (1992), 38–94.

2 Code fantôme

Il arrive souvent, dans la pratique de la vérification déductive des programmes, d'introduire des données et des calculs auxiliaires dont le seul et unique but est de rendre le processus de la vérification plus simple. Ce genre de données, superflues de point du vue opérationnel mais utiles du point de vue de la preuve, est ce l'on a l'habitude d'appeler le « code fantôme » ("ghost code" en anglais). On parle ainsi de variables « fantômes », de paramètres de fonctions « fantômes », d'attributs « fantômes » dans des classes, etc. La possibilité de donner à des éléments de la spécification les apparences du code opérationnel facilite beaucoup le raisonnement sur les propriétés des programmes : par exemple, on peut suivre de près les modifications d'une partie de la mémoire avec une référence fantôme ou incrémenter un compteur pour montrer qu'un algorithme a une certaine complexité en temps ; de même, doter une structure de données d'un champ fantôme permet d'y enchâsser son modèle logique, ce qui facilite par exemple le raisonnement sur la préservation des invariants du type lorsque le contenu des données change.

Pour donner un exemple très simple d'utilisation du code fantôme, regardons comment un paramètre formel fantôme peut faciliter la preuve du petit programme OCaml suivant, `fibo`, qui calcule le n-ième terme de la suite de Fibonacci, à l'aide d'une fonction auxiliaire `aux` :

```
let rec aux a b n =
  if n = 0 then a else aux b (a+b) (n-1)

let fibo n = aux 0 1 n
```

Supposons que l'on veuille montrer la correction de `fibo` vis-à-vis de la spécification suivante

```
let fibo (n: int) : int
   requires { 0 ≤ n }
   ensures  { result = fib n }
 = aux 0 1 n
```

où `fib n` est une logique axiomatisée donnant le n-ième terme de la suite de Fibonacci. Pour prouver que le code du `fibo` s'accorde avec sa spécification, il nous faut, bien entendu, spécifier le comportement de la fonction auxiliaire `aux`. Par exemple, on pourrait donner à `aux` le contrat suivant :

```
let rec aux (a b n: int) : int
  requires { 0 ≤ n }
  requires { ∃k. 0 ≤ k ∧ a = fib k ∧ b = fib (k+1) }
  ensures  { ∃k. 0 ≤ k ∧ a = fib k ∧ b = fib (k+1)
                                   ∧ result = fib (k+n) }
= if n = 0 then a else aux b (a+b) (n-1)
```

Quoique acceptable, cette manière de procéder n'est pas sans inconvénients : d'un côté, on duplique une partie de la spécification (la quantification existentielle) dans la pré- et la post-condition ; de plus, l'utilisation des quantificateurs existentiels rend la spécification moins lisible pour un humain et en même temps moins maniable du point de vue de la preuve automatique. Une manière de remédier à ces inconvénients est de transformer les quantificateurs existentiels en quelque chose de plus constructible, à savoir, de matérialiser l'existence de l'indice `k` par l'ajout d'un paramètre supplémentaire de la fonction `aux` :

```
let rec aux (k: int) (a b n: int) : int
  requires { 0 ≤ k ∧ 0 ≤ m }
  requires { a = fib k ∧ b = fib (k+1) }
  ensures  { result = fib (k+n) }
  ...
```

Ceci rend la spécification plus simple et plus claire. Bien entendu, l'appel récursif dans la définition de `aux`, ainsi que l'appel depuis la fonction principale, doivent être modifiés en conséquence en fournissant un argument supplémentaire :

```
  ...
= if n = 0 then a else aux (k+1) b (a+b) (n-1)

let fibo (n: int) : int
  requires { 0 ≤ n }
  ensures  { result = fib n }
= aux 0 0 1 n
```

De tels paramètres, arguments et calculs auxiliaires sont un exemple typique de l'usage que l'on fait du code fantôme (dans le code ci-dessus, tout ce qui relève du code fantôme est colorié en bleu clair). Bien que très simple, l'exemple que nous venons de voir soulève déjà plusieurs questions. D'un point de vue purement syntaxique, les fonctions `fibo` avec et sans code fantôme sont deux programmes différents. Le code fantôme peut-t-il être effacé, une fois la preuve terminée et, si oui, comment ? Plus important encore, en supposant que l'on ait prouvé la fonction `fibo` avec code fantôme, pourquoi sa correction implique-t-elle la correction du programme initial ? Les deux programmes sont-ils donc équivalents d'un point de vue opérationnel ?

L'idée essentielle derrière l'utilisation correcte du code fantôme réside dans le *principe de non-interférence* : aussi intriquée ou complexe que soit la manière dont le code fantôme est utilisé, on doit toujours pouvoir l'effacer du programme sans que cela

affecte le comportement observable de ce dernier et en particulier le résultat final de son exécution. Sur l'exemple de `fibo`, il est facile de se convaincre que les calculs et les données fantômes n'ont aucun impact sur le résultat final du programme. Pour des programmes quelque peu plus complexes, vérifier et garantir la non-interférence du code fantôme avec le code *réel*[1] devient une affaire non-triviale, notamment en présence d'effets de bord. Ainsi, lorsque le langage permet la modification de mémoire, on doit vérifier qu'une expression fantôme n'écrit jamais dans une case mémoire accessible depuis des références réelles. De même, si l'on dispose d'un mécanisme d'exceptions, on doit s'assurer que le code fantôme ne lève jamais d'exception qui lui échapperait en se propageant jusqu'au contexte réel. Plus subtilement encore, le comportement divergeant constitue également un effet. Par conséquent, une analyse correcte de la non-interférence du code fantôme implique la vérification de sa terminaison. En effet, si une sous-expression fantôme du programme divergeait, on pourrait lui assigner une post-condition arbitraire et donc en déduire que le reste du programme est correct vis-à-vis de n'importe quelle spécification. De même, il n'est pas complètement trivial que l'on ne peut pas appliquer l'égalité structurelle à des données ayant des composantes fantômes : en effet, lorsque deux telles valeurs sont structurellement égales à des composantes fantômes près, le résultat de la comparaison ne sera pas le même avant et après l'effacement du code fantôme. On voit donc que la notion de code fantôme, bien qu'elle est aussi vieille que la vérification déductive elle-même et intégrée dans la plupart des outils de vérification modernes [9, 1, 6, 4], présente plusieurs points subtils et mérite une description formelle rigoureuse.

Aussi simple soit-il, l'exemple de `fibo` montre aussi un autre aspect important de la notion du code fantôme qui est la possibilité de réutiliser les mêmes types de données et opérations que dans le code réel. Dans notre exemple, on réutilise le même type primitif `int`, les constantes numériques et l'opération d'addition mais, d'une manière plus générale, on pourrait souhaiter utiliser dans le code fantôme des types et des opérations définis par l'utilisateur. Il n'y a donc pas seulement des questions d'utilisation correcte du code fantôme mais également des questions liées à des choix de conception qui nous importent. À quel point devons-nous annoter le code fantôme explicitement ? Une variable fantôme doit-elle être annotée en tant que telle ou bien peut-on inférer le statut fantôme des variables à partir des valeurs qu'on leur affecte ? Plus important encore : quelles constructions, fonctionnalités du langage et parties du code peut-on partager entre le code fantôme et le code réel ? Par exemple, peut-on passer une valeur fantôme en argument d'une fonction dont le paramètre formel correspondant ne porte pas d'étiquette fantôme explicite ? De même, a-t-on le droit de stocker une valeur fantôme dans une structure de données qui n'a pas été spécialement conçue pour contenir des données fantômes, par exemple un tableau ou un n-uplet ? D'une manière générale, lorsque l'on conçoit un langage avec code fantôme on doit définir où le code fantôme a le droit d'apparaître et ce qui peut apparaître dans le code fantôme.

1. dans ce chapitre, par *réel* on entendra toujours « non-fantôme »

Le but de ce chapitre est de montrer comment une approche statique basée sur un système de types avec effets peut apporter les réponses à toutes ces questions de sûreté et de conception, en permettant une utilisation du code fantôme à la fois correcte, expressive et concise. Concrètement, nous allons introduire, en tant que preuve de concept, un petit langage de programmation, que nous appelons GhostML qui est un langage fonctionnel à la ML avec un état global, des fonctions récursives et du code fantôme. L'intérêt principal de ce formalisme sera de montrer comment le système de type permet d'utiliser les mêmes types de données et les mêmes définitions de fonctions dans le code fantôme et le code réel. Cela fera l'objet des sections 2.1 et 2.2. Ensuite, dans la section 2.3 nous verrons comment définir l'opération d'effacement du code fantôme et montrerons sa correction en utilisant la technique de bisimulation ce qui aura comme conséquence la non-interférence du code fantôme avec le code réel. Le langage du formalisme est délibérément simplifié pour se concentrer sur les points et les difficultés essentiels du traitement du code fantôme, laissant les fonctionnalités plus avancées à la discussion de l'implémentation de code fantôme en Why3, ce qui fera l'objet de la section 2.4. Nous terminerons le chapitre avec la section 2.5 où nous donnerons des notes bibliographiques en rapport avec la notion du code fantôme en général et notre travail en particulier.

2.1 GhostML : un petit langage avec du code fantôme

Dans cette section, nous présentons GhostML, un langage fonctionnel à la ML avec un état global, des fonctions récursives et du code fantôme. Nous donnons d'abord sa syntaxe puis sa sémantique opérationnelle.

2.1.1 Syntaxe de GhostML

La syntaxe des types de GhostML est donnée dans la figure 2.1. Il y a deux sortes de types : des types scalaires et des types de fonctions. Ces derniers sont de la forme $\tau_2^{\beta} \xrightarrow{\epsilon} \tau_1$ où τ_2 est le type du paramètre formel et τ_1 est celui du résultat. Le symbole β placé en exposant du type τ_2 est une étiquette booléenne qui indique le statut fantôme du paramètre formel : lorsque $\beta = \top$, il s'agit d'un paramètre fantôme, et sinon, il s'agit d'un paramètre réel[2]. Le symbole ϵ est une autre valeur booléenne qui indique les *effets latents* d'une fonction : ϵ vaut $\top$ lorsque l'appel d'une telle fonction est susceptible de ne pas terminer ou de modifier l'état mémoire et vaut $\bot$ sinon. La syntaxe des

τ	$::=$	TYPES
	$\mid \nu$	*type scalaire*
	$\mid \tau^{\beta} \xrightarrow{\epsilon} \tau$	*type de fonction*
ν	$::=$	TYPES SCALAIRES
	$\mid$ `Int` $\mid$ `Bool` $\mid$ `Unit`	
β	$\in \{\bot, \top\}$	STATUT FANTÔME
ϵ	$\in \{\bot, \top\}$	EFFET LATENT

FIGURE 2.1 – Types et effets.

expressions de GhostML est donnée dans la figure 2.2. Toutes les constructions, à l'exception du mot-clef `ghost` qui permet de transformer une expression en du code fantôme, sont des constructions que l'on retrouve dans des langages de programmation de la famille ML. On distingue deux sortes d'expressions : les expressions *atomiques* et celles qui sont *composées*. Les expressions atomiques permettent de séparer en deux catégories distinctes les variables et les valeurs sémantiques, chaque valeur sémantique

2. Rappelons que par *réel* on entend « non-fantôme »

e	$::=$	EXPRESSIONS
	$\mid a$	*expression atomique*
	$\mid e\ a$	*application*
	$\mid \texttt{let}\ x^\beta = e\ \texttt{in}\ e$	*liaison locale*
	$\mid \texttt{if}\ a\ \texttt{then}\ e\ \texttt{else}\ e$	*conditionnelle*
	$\mid r^\beta := a$	*affectation d'une référence*
	$\mid\ !r^\beta$	*lecture d'une référence*
	$\mid \texttt{ghost}\ e$	*code fantôme*
a	$::=$	EXPRESSIONS ATOMIQUES
	$\mid x^\beta$	*variable*
	$\mid v$	*valeur*
v	$::=$	VALEURS
	$\mid c$	*constante scalaire*
	$\mid \lambda x^\beta : \tau.\ e$	*fonction anonyme*
	$\mid \texttt{rec}\ f^\beta : \tau^\beta \overset{\epsilon}{\Longrightarrow} \tau.\ \lambda x^\beta : \tau.\ e$	*fonction récursive*
c	$::=$	CONSTANTS
	$\mid \ldots, -1, 0, 1, \ldots$	*entiers*
	$\mid \texttt{true}, \texttt{false}$	*booléens*
	$\mid \texttt{()}$	*unit*
	$\mid +, \vee, =, \ldots$	*opérations primitives*

FIGURE 2.2 – Syntaxe abstraite des expressions de GhostML.

étant soit une constante scalaire, une fonction anonyme notée $\lambda x^\beta : \tau.\ e$, ou bien une fonction récursive que l'on note $\texttt{rec}\ f^\beta : \tau^\beta \overset{\epsilon}{\Longrightarrow} \tau.\ \lambda x^\beta : \tau.\ e$.

Les expressions composées comportent la conditionnelle, la liaison locale, l'application d'une expression à une expression atomique, l'affectation d'une référence globale, notée $r^\beta := a$, et l'accès à son contenu, noté $!r^\beta$. Nous écrivons $x^\beta, y^\beta, \ldots, f^\beta, g^\beta; \ldots$ pour dénoter les variables (y compris les noms des fonctions récursives) et $r^\beta, s^\beta, \ldots$ pour dénoter les références. Les deux forment des catégories syntaxiques distinctes. Chaque occurrence d'une variable ou d'une référence est annotée par son statut fantôme β. Par ailleurs, on adopte la convention de nommage suivante : les noms des variables et des paramètres formels fantômes sont disjoints des noms des variables et des paramètres réels. Une telle convention n'est pas contraignante en soi, car elle équivaut à une procédure d'α-renommage. Notons que, contrairement aux variables,

les références ne peuvent être utilisées qu'avec un opérateur d'affectation ou d'accès. Enfin, nous supposons dans la suite avoir fixé à l'avance un ensemble fini de références globales deux à deux distinctes. Il n'y a donc pas de possibilité d'introduire des références locales et en particulier, il n'y a pas d'aliasing possible entre les références.

Notons également que les expressions composées sont écrites dans une variante de la forme *A-normale* [5] : dans la conditionnelle, l'application et l'affectation l'une des sous-expressions est nécessairement atomique. L'intérêt d'une telle présentation est qu'elle permet de simplifier le nombre de cas à considérer lorsque l'on décrit le modèle d'exécution du langage, sans pour autant restreindre son pouvoir expressif. Sans démontrer cette dernière affirmation, notons simplement que l'on peut toujours exprimer les étapes intermédiaires du calcul à l'aide d'une liaison locale. Par exemple, la conditionnelle $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$ se réécrit en $(\texttt{let}\ x^\beta = e_1\ \texttt{in}\ (\texttt{if}\ x^\beta\ \texttt{then}\ e_2\ \texttt{else}\ e_3))$, avec x^β est variable fraîche. Pour rendre les exemples plus facile à lire, on s'autorisera systématiquement à ne pas y respecter la forme A-normale. Considérons par exemple l'expression GhostML suivante :

$$\texttt{let}\ upd^\top = \lambda x^\bot : \texttt{Int}.\ s^\top := x^\bot\ \texttt{in}\ upd^\top\ !r^\bot.$$

Ici on définit d'abord la fonction fantôme $upd^\top$ qui affecte le contenu de la référence fantôme $s^\top$ par la valeur de son paramètre réel $x^\bot$, et on applique ensuite $upd^\top$ au contenu de la référence réelle $r^\bot$. Par ailleurs, on s'est permis d'écrire simplement $upd^\top\ !r^\bot$ qui n'est pas en forme A-normale, au lieu d'introduire une liaison locale auxiliaire ce qui rendrait l'exemple moins lisible.

2.1.2 Sémantique opérationnelle de GhostML

La sémantique opérationnelle de GhostML est définie par l'ensemble des règles données dans la figure 2.3. Chaque règle est de la forme

$$\mu \cdot e \longrightarrow \mu' \cdot e'$$

où $\mu \cdot e$ et $\mu' \cdot e'$ sont respectivement la configuration avant et après un pas d'exécution. Une configuration est formée d'une expression sans variables libres et d'un état mémoire μ défini comme une fonction associant à chaque référence globale r^β une constante scalaire $\mu(r^\beta)$. Les références ne contiennent donc que des valeurs scalaires, jamais des fonctions.

Chaque règle décrit une manière possible de réaliser un pas de transition entre la configuration de départ et la configuration d'arrivée. Toutes les règles, à l'exception des règles (E-Ctx-Let) et (E-Ctx-App), correspondent à des cas de « réduction en tête » où l'expression e est un rédex, c'est-à-dire qu'il n'existe pas de sous-expression de e pour laquelle une des règles de sémantique s'appliquerait. Quant aux règles (E-Ctx-Let) et (E-Ctx-App), ce sont donc les deux seules règles « contextuelles » : elles identifient l'endroit dans e où le calcul doit se poursuivre. La règle (E-Ghost) exprime le fait que

$$\frac{}{\mu \cdot \texttt{ghost}\ e_1 \longrightarrow \mu \cdot e_1}\,(\text{E-Ghost})$$

$$\frac{1 \leq k < \text{arity}(c_0)}{\mu \cdot c_0\ c_1\ \ldots\ c_k \longrightarrow \mu \cdot \lambda x^\beta : \nu.\ c_0\ c_1\ \ldots\ c_k\ x^\beta}\,(\text{E-Op-}\lambda)$$

$$\frac{m = \text{arity}(c_0) \qquad \delta(c_0, c_1, \ldots, c_m) \text{ est défini}}{\mu \cdot c_0\ c_1\ \ldots\ c_m \longrightarrow \mu \cdot \delta(c_0, c_1, \ldots, c_m)}\,(\text{E-Op-}\delta)$$

$$\frac{}{\mu \cdot (\lambda x^\beta : \tau.\ e_1)\ v \longrightarrow \mu \cdot e_1[x^\beta/v]}\,(\text{E-App-}\lambda)$$

$$\frac{v_0 = \texttt{rec}\ f^\beta : \tau^\beta \xRightarrow{\epsilon} \tau.\ \lambda x^\beta : \tau.\ e_1}{\mu \cdot (v_0\ v) \longrightarrow \mu \cdot e_1[f^\beta/v_0][x^\beta/v]}\,(\text{E-App-Rec})$$

$$\frac{}{\mu \cdot \texttt{let}\ x^\beta = v\ \texttt{in}\ e_1 \longrightarrow \mu \cdot e_1[x^\beta/v]}\,(\text{E-}\zeta)$$

$$\frac{}{\mu \cdot \texttt{if true then}\ e_1\ \texttt{else}\ e_2 \longrightarrow \mu \cdot e_1}\,(\text{E-True})$$

$$\frac{}{\mu \cdot \texttt{if false then}\ e_1\ \texttt{else}\ e_2 \longrightarrow \mu \cdot e_2}\,(\text{E-False})$$

$$\frac{r^\beta \in \text{dom}\,\mu}{\mu \cdot !r^\beta \longrightarrow \mu \cdot \mu(!r^\beta)}\,(\text{E-Deref})$$

$$\frac{r^\beta \in \text{dom}\,\mu}{\mu \cdot (r^\beta := c) \longrightarrow \mu[r^\beta \mapsto c] \cdot ()}\,(\text{E-Assign})$$

$$\frac{\mu \cdot e_1 \longrightarrow \mu' \cdot e_1'}{\mu \cdot (e_1\ v) \longrightarrow \mu' \cdot (e_1'\ v)}\,(\text{E-Ctx-App})$$

$$\frac{\mu \cdot e_1 \longrightarrow \mu' \cdot e_1'}{\mu \cdot (\texttt{let}\ x^\beta = e_1\ \texttt{in}\ e_2) \longrightarrow \mu' \cdot (\texttt{let}\ x^\beta = e_1'\ \texttt{in}\ e_2)}\,(\text{E-Ctx-Let})$$

FIGURE 2.3 – Sémantique opérationnelle de GhostML.

du point du vue opérationnel, il n'y a pas de différence entre le code fantôme et le code réel. Cela implique en particulier qu'il n'y a pas de valeurs scalaires fantômes, car une expression telle que `ghost` 42, qui n'est pas une valeur en elle-même, se réduit toujours en 42. Les autres règles sont des règles classiques d'un langage fonctionnel avec références globales et l'évaluation en l'appel par valeur ("call-by- value" en terminologie anglaise). Ainsi, les règles (E-Op-λ), (E-Op-δ) permettent d'évaluer l'application d'une constante c_0 à des constantes $c_1, \ldots, c_k$. Lorsque l'application $c_0\, c_1 \ldots c_k$ est partielle, c'est-à-dire quand $1 \leq k < \mathrm{arity}(c_0)$, on la réduit avec la règle (E-Op-λ) en une fonction $\lambda x^{\beta} : \nu.\ c_0\ c_1\ \ldots\ c_k\ x^{\beta}$. Lorsque l'application est totale, c'est-à-dire quand $k = \mathrm{arity}(c_0)$, on la réduit par la règle (E-Op-δ) en une constante donnée par un oracle sémantique δ. Dans ce cas, on note le résultat de l'évaluation $\delta(c_0, c_1, \ldots, c_m)$. Par exemple, on a $\delta(\texttt{not}, \texttt{true}) = \texttt{false}$, $\delta(+, 47, -5) = 42$, etc.

On dénote par $\longrightarrow^{\star}$ la clôture réflexive transitive de $\longrightarrow$. Lorsque l'on veut être précis sur le nombre de pas d'exécution, on l'ajoute en exposant sur la flèche. Par exemple, $\longrightarrow^{0|1}$ dénote une évaluation en 0 ou 1 étape, $\longrightarrow^{\geq 1}$ l'évaluation en un nombre d'étapes strictement positif, etc. On dit aussi qu'une expression e s'évalue en une valeur v dans l'état-mémoire μ lorsqu'il existe un état mémoire μ' tel que

$$\mu \cdot e \ \longrightarrow^{\star}\ \mu' \cdot v.$$

Rappelons que, dans une configuration sémantique, l'expression e doit être close. Observons par ailleurs que les règles de sémantique n'introduisent pas de variables libres. Par conséquent, la valeur finale v est une expression close aussi. Pour tenir compte de la situation où l'évaluation à partir d'une configuration $\mu \cdot e$ donnée diverge, on dénote de tels comportements divergents par le prédicat $\mu \cdot e \ \longrightarrow\ \infty$ que l'on définit par coinduction de la manière suivante :

$$\frac{\mu \cdot e \longrightarrow \mu' \cdot e' \qquad \mu' \cdot e' \ \longrightarrow\ \infty}{\mu \cdot e \ \longrightarrow\ \infty}\,(\text{E-Div}).$$

Enfin, notons que quelle que soit la configuration considérée, il y aura toujours au plus une règle de sémantique s'applique. Autrement dit, la sémantique opérationnelle de notre langage est déterministe.

2.2 Typage des expressions GhostML

Lorsque l'évaluation ne peut plus avancer, l'expression qui en résulte n'est pas nécessairement une valeur. Par exemple, aucune règle de sémantique ne s'applique à des expressions comme `if` 42 `then` e_1 `else` e_2 ou (`true` 42). Pour autant, de telles expressions *irréductibles* ne sont pas des valeurs. En substance, cela signifie que le processus d'évaluation s'est arrêté dans une configuration qui n'a de sens calculatoire.

Le but d'un système de typage est précisément de garantir que les expressions bien typées « ne dévient pas de leur chemin » [3]. En clair, cela veut dire que l'évaluation des expressions bien typées doit soit diverger, soit terminer sur une valeur. Mais les objectifs du système de types que nous présentons dans cette section ne se limitent pas à la sûreté du typage. Le but du jeu sera de garantir par ailleurs que les programmes bien typés possèdent une propriété nécessaire pour démontrer la non-interférence entre le code réel et le code fantôme, ce qui dans notre formalisme s'exprime par le fait que le code fantôme ne modifie jamais le contenu des références réelles et que son exécution termine toujours. Nous allons donc présenter un tel système de types pour les expressions GhostML puis nous montrerons sa correction.

2.2.1 Système de types avec effets de GhostML

Le système de types de GhostML est défini inductivement par l'ensemble des règles qui sont données dans la figure 2.4. Chaque règle se termine par un jugement de la forme

$$\Gamma \cdot \Sigma \vdash e : \tau \cdot \beta \cdot \epsilon$$

où τ est le type de l'expression e et les valeurs booléennes β et ϵ indiquent, respectivement, son statut fantôme et ses effets de bord. La fonction Γ définie sur l'ensemble des variables (dont on a convenu qu'il correspondait à l'union disjointe des variables fantômes et des variables réelles) représente l'environnement de typage des variables. La fonction Σ définie sur l'ensemble des références globales (dont on a convenu qu'il était fini et fixé à l'avance) représente l'environnement de typage des références globales associant à chaque référence r^β un type scalaire $\Sigma(r^\beta)$.

Pour garantir la non-interférence, chaque règle comporte en plus la condition implicite suivante :

$$\beta \implies \neg\epsilon \wedge \neg\epsilon^+(\tau) \tag{2.1}$$

où l'on définit $\epsilon^+(\tau)$ récursivement sur la structure du type τ par les équations :

$$\begin{array}{lcl} \epsilon^+(\nu) & \triangleq & \bot \\ \epsilon^+({\tau_2}^\beta \xrightarrow{\epsilon} \tau_1) & \triangleq & \epsilon \vee \epsilon^+(\tau_1) \end{array}$$

3. "Well-typed programs cannot go wrong" (Milner, 1978)

L'idée est que la condition (2.1) serve d'invariant du typage : lorsque l'expression e est fantôme, son exécution ne peut ni diverger, ni modifier le contenu des références réelles. Notons qu'un invariant plus faible $\beta \implies \neg\epsilon$ serait suffisant mais l'avantage de la condition (2.1), où l'on demande en plus que $\neg\epsilon^+(\tau)$ lorsque e est fantôme, est qu'elle permet de rejeter les définitions de fonctions dont les effets latents ne respecteraient pas la non-interférence.

Expliquons les règles du typage en détails. La règle (T-Const) exprime le fait qu'une constante c correspond toujours à un code réel, pur et terminant. En particulier, si c est une opération, les effets latents doivent être tous vides ($\epsilon^+(\tau) = \bot$), et les types des arguments formels doivent être tous réels, ce que l'on note avec $\beta^+(\tau) = \bot$ où $\beta^+(\tau)$ est définie à l'instar de $\epsilon^+(\tau)$ par les équations :

$$\begin{array}{lcl} \beta^+(\nu) & \triangleq & \bot \\ \beta^+(\tau_2 \overset{\beta}{\underset{\epsilon}{\Longrightarrow}} \tau_1) & \triangleq & \beta \vee \beta^+(\tau_1) \end{array}$$

Le type des constantes est donné par un oracle $\texttt{Typeof}(c)$. Ainsi, on a par exemple $\texttt{Typeof}(+) = \texttt{Int}^\bot \overset{\bot}{\Longrightarrow} \texttt{Int}^\bot \overset{\bot}{\Longrightarrow} \texttt{Int}$. Par ailleurs, on impose que les oracles `Typeof` et δ soient toujours cohérents entre eux :

Définition 2.2.1 (Cohérence de `Typeof` et δ)**.** Soit c_0 une constante telle que

$$\texttt{Typeof}(c_0) = \nu_1^\bot \overset{\bot}{\Longrightarrow} \nu_2^\bot \overset{\bot}{\Longrightarrow} \ldots \overset{\bot}{\Longrightarrow} \nu_m^\bot \overset{\bot}{\Longrightarrow} \nu_0,$$

Alors l'arité du symbole c_0 est égale à m et quelles que soient les constantes $c_1, \ldots, c_m$ avec $\texttt{Typeof}(c_i) = \nu_i$ pour $1 \leq i \leq m$, la constante $\delta(c_0, c_1, \ldots, c_m)$ est bien définie et et on a $\texttt{Typeof}(\delta(c_0, c_1, \ldots, c_m)) = \nu_0$.

La règle (T-Ghost) permet de typer une expression e dans un contexte fantôme `ghost` e, pourvu que e soit pure et terminante. Dans les règles (T-Deref) et (T-Assign), le type de la valeur stockée dans la référence r^β est donné par l'environnement Σ. Rappelons que dans notre formalisme une référence globale ne peut contenir que des constantes scalaires. Par conséquent, la récursivité (et donc potentiellement la divergence) ne peut pas être encodée dans notre langage à l'aide d'un « nœud de Landin » (c'est-à-dire, en encodant la récursivité par une référence contenant une fonction). Par ailleurs, la règle (T-Assign) autorise de stocker une valeur réelle dans une référence fantôme mais l'inverse n'est pas possible : la prémisse $\beta' \leq \beta$ interdit de modifier le contenu d'une référence réelle par une valeur fantôme. De plus, dans la conclusion de la règle, les effets d'écriture sont présents ($\epsilon = \top$) si et seulement si la référence modifiée est réelle : l'écriture dans les références fantômes ne fait donc pas partie des effets dans notre système du typage.

La règle (T-λ) décrit le typage des fonctions anonymes. D'une part, le statut fantôme d'une fonction anonyme est celui de son corps. D'autre part, les effets ϵ calculés pour le corps deviennent latents et sont placés sur la flèche du type fonctionnel, alors que la fonction elle-même n'a pas d'effet. Le typage des fonctions récursives, donné

$$\frac{\texttt{Typeof}(c) = \tau \qquad \neg\epsilon^+(\tau) \wedge \neg\beta^+(\tau)}{\Gamma \cdot \Sigma \vdash c : \tau \cdot \bot \cdot \bot}\,(\text{T-Const}) \qquad \frac{\Gamma(x^\beta) = \tau}{\Gamma \cdot \Sigma \vdash x^\beta : \tau \cdot \beta \cdot \bot}\,(\text{T-Var})$$

$$\frac{\Gamma \cdot \Sigma \vdash e_1 : \tau \cdot \beta \cdot \bot}{\Gamma \cdot \Sigma \vdash (\texttt{ghost}\ e_1) : \tau \cdot \top \cdot \bot}\,(\text{T-Ghost}) \qquad \frac{\Sigma(r^\beta) = \nu}{\Gamma \cdot \Sigma \vdash\ !r^\beta : \nu \cdot \beta \cdot \bot}\,(\text{T-Deref})$$

$$\frac{\Sigma(r^\beta) = \nu \qquad \Gamma \cdot \Sigma \vdash a : \nu \cdot \beta' \cdot \bot \qquad \beta' \leq \beta}{\Gamma \cdot \Sigma \vdash (r^\beta := a) : \texttt{Unit} \cdot \beta \cdot \neg\beta}\,(\text{T-Assign})$$

$$\frac{[x^\beta \mapsto \tau]\Gamma \cdot \Sigma \vdash e_1 : \tau_1 \cdot \beta_1 \cdot \epsilon}{\Gamma \cdot \Sigma \vdash (\lambda x^\beta : \tau.\ e_1) : \tau^\beta \xrightarrow{\epsilon} \tau_1 \cdot \beta_1 \cdot \bot}\,(\text{T-}\lambda)$$

$$\frac{\Gamma \cdot \Sigma \vdash e_1 : {\tau_2}^\bot \xrightarrow{\epsilon_0} \tau_1 \cdot \beta_1 \cdot \epsilon_1 \qquad \Gamma \cdot \Sigma \vdash a_2 : \tau_2 \cdot \beta_2 \cdot \bot}{\Gamma \cdot \Sigma \vdash (e_1\ a_2) : \tau_1 \cdot \beta_1 \vee \beta_2 \cdot \epsilon_0 \vee \epsilon_1}\,(\text{T-App}^\bot)$$

$$\frac{\Gamma \cdot \Sigma \vdash e_1 : {\tau_2}^\top \xrightarrow{\epsilon_0} \tau_1 \cdot \beta_1 \cdot \epsilon_1 \qquad \Gamma \cdot \Sigma \vdash a_2 : \tau_2 \cdot \beta_2 \cdot \bot}{\Gamma \cdot \Sigma \vdash (e_1\ a_2) : \tau_1 \cdot \beta_1 \cdot \epsilon_0 \vee \epsilon_1}\,(\text{T-App}^\top)$$

$$\frac{\Gamma \cdot \Sigma \vdash e_1 : \tau_1 \cdot \beta_1 \cdot \epsilon_1 \qquad [x^\bot \mapsto \tau_1]\Gamma \cdot \Sigma \vdash e_2 : \tau_2 \cdot \beta_2 \cdot \epsilon_2}{\Gamma \cdot \Sigma \vdash (\texttt{let}\ x^\bot = e_1\ \texttt{in}\ e_2) : \tau_2 \cdot \beta_1 \vee \beta_2 \cdot \epsilon_1 \vee \epsilon_2}\,(\text{T-Let}^\bot)$$

$$\frac{\Gamma \cdot \Sigma \vdash e_1 : \tau_1 \cdot \beta_1 \cdot \bot \qquad [x^\top \mapsto \tau_1]\Gamma \cdot \Sigma \vdash e_2 : \tau_2 \cdot \beta_2 \cdot \epsilon_2}{\Gamma \cdot \Sigma \vdash (\texttt{let}\ x^\top = e_1\ \texttt{in}\ e_2) : \tau_2 \cdot \beta_2 \cdot \epsilon_2}\,(\text{T-Let}^\top)$$

$$\frac{\begin{array}{c}\epsilon_0 = \texttt{CheckTermination}(\texttt{rec}\ f^{\beta_1} : \tau_0.\ \lambda x^\beta : \tau_2.\ e_1) \qquad \epsilon_0 \leq \epsilon_1 \\ [f^{\beta_1} \mapsto \tau_0]\Gamma \cdot \Sigma \vdash (\lambda x^\beta : \tau_2.\ e_1) : \tau_0 \cdot \beta_1 \cdot \bot \qquad \tau_0 = (\tau_2^\beta \xrightarrow{\epsilon_1} \tau_1)\end{array}}{\Gamma \cdot \Sigma \vdash (\texttt{rec}\ f^{\beta_1} : \tau_0.\ \lambda x^\beta : \tau_2.\ e_1) : \tau_0 \cdot \beta_1 \cdot \bot}\,(\text{T-Rec})$$

$$\frac{\Gamma \cdot \Sigma \vdash a : \texttt{Bool} \cdot \beta_0 \cdot \bot \qquad \Gamma \cdot \Sigma \vdash e_1 : \tau \cdot \beta_1 \cdot \epsilon_1 \qquad \Gamma \cdot \Sigma \vdash e_2 : \tau \cdot \beta_2 \cdot \epsilon_2}{\Gamma \cdot \Sigma \vdash (\texttt{if}\ a\ \texttt{then}\ e_1\ \texttt{else}\ e_2) : \tau \cdot \beta_0 \vee \beta_1 \vee \beta_2 \cdot \epsilon_1 \vee \epsilon_2}\,(\text{T-If})$$

FIGURE 2.4 – Règles du typage de GhostML.

par la règle (T-Rec), est plus complexe. En effet, on doit vérifier que le code fantôme termine toujours. Or, en présence de fonctions récursives, la terminaison est en général un problème undécidable. La solution la plus simple serait alors d'interdire l'utilisation des fonctions récursives dans le code fantôme, ce qui reviendrait à assigner à toute définition récursive un effet latent de non-terminaison. Cependant, en pratique, les outils de vérification existants disposent souvent de techniques d'analyse statique permettant d'établir la terminaison des fonctions récursives, au quel cas on peut les utiliser dans un contexte fantôme. Pour tenir compte de cette possibilité de terminaison prouvable, sans pour autant donner une préférence à une approche particulière, notre formalisme modélise suppose l'existence d'un oracle `CheckTermination` qui, étant donnée une définition récursive, indique $\bot$ lorsque sa terminaison peut être statiquement établie (c'est-à-dire que l'on peut montrer que tout appel à une telle fonction récursive termine quel que soit l'état mémoire courant et l'argument auquel la fonction est appliquée), et $\top$ sinon. Un exemple typique d'un tel oracle serait la génération des obligations de preuve à partir des variants fournis par l'utilisateur. Notons que `CheckTermination` ne prend pas en compte la non-interférence et donc n'a pas besoin de distinguer lui-même le code fantôme et le code réel. Enfin, pour tenir compte de la situation où, d'après l'oracle, une expression termine mais possède des effets, car modifie des références réelles, la règle (T-Rec) impose que les effets latents des fonctions récursives soient au moins aussi grands que les effets donnés par l'oracle ($\epsilon_0 \leq \epsilon_1$). Notons, en revanche, que le statut fantôme β_1 d'une fonction récursive dans notre formalisme doit être le même que celui de la fonction anonyme qui la définit.

Le typage des expressions conditionnelles représente un cas de figure où le code fantôme peut se propager depuis l'une des sous-expressions de sorte que l'expression toute entière est typée comme fantôme. En effet, comme le montre la règle (T-If), il suffit qu'une seule des branches soit fantôme pour que la conditionnelle le devienne elle-même. Par conséquent, le typage rejette les conditionnelles dont l'une des deux branches est fantôme tandis que l'autre produit des effets, car cela ferait sinon défaut à la condition implicite (2.1). Ainsi, l'expression `if true then` $r^{\bot} := 42$ `else ghost ()` n'est pas bien typée : le statut fantôme de l'expression doit être $\top$, puisque sa branche droite est `ghost ()` mais, d'après la condition (2.1), les effets de la conditionnelle doivent alors être $\bot$, ce qui est faux ici, puisque la branche gauche $r^{\bot} := 42$ a un effet $\top$.

La règle (T-Let$^{\top}$) permet de typer les liaisons locales dont le lieur est fantôme. Notons que, étant donnée une telle expression `let` $x^{\top} = e_1$ `in` e_2, on peut lier $x^{\top}$ à e_1, même si e_1 n'est pas lui-même une expression fantôme. Il faut néanmoins dans ce cas-là que e_1 soit pure et terminante. Plus subtil encore, lorsque e_1 est fantôme, une expression `let` ne devient pas nécessairement fantôme : plutôt que de propager le contexte fantôme à l'intérieur de e_2, le système de types propage l'information que la valeur associée à $x^{\top}$ est fantôme, garantissant ainsi que l'utilisation de chaque occurrence de $x^{\top}$ dans e_2 ne va pas interférer avec les parties réelles.

La règle (T-App$^{\top}$) donne le typage des applications $(e_1\ a_2)$ où le type de e_1 est de la

forme ${\tau_2}^\top \xRightarrow{\epsilon_0} \tau_1$. Comme dans le cas de (T-Let$^\top$), l'expression e_1 peut être appliquée aussi bien à un argument réel qu'à un argument fantôme et le statut fantôme de l'application ne dépend pas du statut fantôme de l'argument a_2 (la situation ici est même plus simple, puisque, en forme A-normale, l'expression a_2 est atomique donc nécessairement pure et terminant). Considérons le programme suivant :

$$\begin{array}{l}\texttt{let } succ^\top = \lambda y^\top : \texttt{Int}.\ (y^\top + 1) \texttt{ in}\\ \texttt{let } x^\perp = !r^\perp \texttt{ in}\\ \texttt{let } trace^\top = succ^\top\ x^\perp \texttt{ in}\\ r^\perp := x^\perp + 1\end{array}$$

Grâce à la règle (T-App$^\top$), on passe une variable réelle $x^\perp$ en argument de la fonction fantôme $succ^\top$. Ensuite, grâce à la règle (T-Let$^\top$), le contexte fantôme correspondant à l'application fantôme $(succ^\top\ x^\perp)$ est « contenu » dans la variable $trace^\top$ sans interférer avec le code réel qui suit après.

La règle (T-Let$^\perp$) est, d'une certaine manière, le dual de la règle (T-Let$^\top$) : lorsque l'on lie une variable réelle $x^\perp$ à une expression fantôme e_1, le système de types propage le contexte fantôme jusqu'à ce que l'expression $\texttt{let } x^\perp = e_1 \texttt{ in } e_2$ toute entière devienne fantôme, « contaminée » par le statut fantôme de e_1. De même, la règle (T-App$^\perp$) permet de passer un argument fantôme à une fonction dont le paramètre formel est déclaré comme réel, auquel cas l'application toute entière devient du code fantôme.

Reprenons l'exemple précédent. Le fait que l'on ait déclaré la fonction $succ^\top$ fantôme est un peu regrettable, car cela empêche de l'appeler dans le code réel. Par exemple, on ne peut pas remplacer la dernière ligne $r^\perp := x^\perp + 1$ simplement par $r^\perp := succ^\top\ x^\perp$. Or, étant donné que la fonction successeur est pure et terminant, il serait naturel de la définir en tant que fonction réelle. C'est là que les règles (T-Let$^\perp$) et (T-App$^\perp$) prennent leur intérêt car elles permettent d'utiliser la même définition à la fois dans le code réel et le code fantôme :

$$\begin{array}{l}\texttt{let } succ^\perp = \lambda y^\perp : \texttt{Int}.\ (y^\perp + 1) \texttt{ in}\\ \texttt{let } x^\perp = !r^\perp \texttt{ in}\\ \texttt{let } trace^\top = succ^\perp\ x^\perp \texttt{ in}\\ r^\perp := succ^\perp\ x^\perp\end{array}$$

En résumé, les règles (T-Let$^\top$) et (T-App$^\top$) permettent de réguler la propagation du code fantôme dans un contexte réel, alors que les règles (T-Let$^\perp$) et (T-App$^\perp$) permettent d'utiliser le code réel dans un contexte fantôme.

Notons qu'il n'y a pas de sous-typage dans le système de types que nous venons de présenter : dans les règles (T-App$^\top$) et (T-App$^\perp$), le paramètre formel et l'argument effectif doivent avoir exactement le même type. En particulier, lorsque le paramètre formel et l'argument sont eux-mêmes des fonctions, les effets latents ainsi que les statuts fantômes des types fonctionnels correspondants doivent être les mêmes. Ainsi, une fonction d'ordre supérieur dont le paramètre formel a le type ${\texttt{Int}^\perp} \xRightarrow{\epsilon} \texttt{Int}$ ne peut pas être appliquée dans notre système à un argument factuel dont le type serait

$\texttt{Int}^{\top} \overset{\epsilon}{\Longrightarrow} \texttt{Int}$.

Pour conclure, remarquons que, lorsque l'on parle des expressions typées, la mise en forme A-normale des expressions doit tenir compte du typage et donc des statuts fantômes des expressions telles que $(e_1\ e_2)$ et $(\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2)$. Bien que nous n'avons pas présenté les règles du typage correspondantes (les expressions $(e_1\ e_2)$ et $(\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2)$ ne faisant pas partie de la syntaxe), on peut montrer néanmoins qu'elles seraient admissibles dans le système du typage que nous venons de présenter. Par conséquent, la mise en forme A-normale des expressions bien typées ne diminue pas l'expressivité de la partie typée du langage. Par exemple, la forme A-normale de la conditionnelle $\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ correspond à l'expression $\texttt{let}\ x^{\beta_0} = e_0\ \texttt{in}\ \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ où β_0 est le statut fantôme de e_0. De même, la mise en forme A-normale de $e_1\ e_2$ correspond à l'expression GhostML $\texttt{let}\ y^{\beta_2} = e_2\ \texttt{in}\ \texttt{let}\ x^{\beta_1} = e_1\ \texttt{in}\ x^{\beta_1}\ y^{\beta_2}$ où β_1 est le statut fantôme du premier paramètre formel du type (nécessairement fonctionnel) de e_1 et β_2 est le statut fantôme de e_2. Un tel procédé montre que la mise en forme A-normale ne diminue pas l'expressivité de la partie du langage typée. En particulier, si l'on ajoutait les expressions $e_1\ e_2$ et $\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ à la grammaire, les règles de typage correspondantes seraient admissibles dans le système du typage que nous venons de présenter.

2.2.2 Correction du typage

Pour montrer la correction du système des types que nous venons de présenter, nous utilisons l'approche syntaxique de Wright et Felleisen [20] où la correction est une conséquence de deux lemmes. D'une part, on montre le lemme de *progrès*, à savoir qu'une expression close bien typée est nécessairement soit une valeur, soit une expression réductible. D'autre part, on montre le lemme de *préservation*, à savoir que lorsqu'une règle de sémantique s'applique à une expression bien typée, le résultat du pas de l'évaluation correspondant reste bien typé et a le même type. Comme nous allons le voir, les effets et le statut fantôme ne sont pas nécessairement les mêmes, car ils peuvent diminuer au fil de l'évaluation (en revanche, ils ne peuvent jamais augmenter). Pour énoncer ces deux lemmes, nous avons besoin de la définition suivante :

Définition 2.2.2. Étant donné un environnement de typage Σ, un état mémoire μ est bien typé par Σ, ce que l'on note $\Sigma \models \mu$, lorsque $\operatorname{dom}\mu \subseteq \operatorname{dom}\Sigma$ et pour chaque $r^{\beta} \in \operatorname{dom}\mu$, on a $\texttt{Typeof}(\mu(r^{\beta})) = \Sigma(r^{\beta})$.

Le progrès de typage s'énonce alors comme suit :

Lemma 2.2.1 (Progrès)**.** Soit e une expression close, telle que $\Gamma \cdot \Sigma \vdash e : \tau \cdot \beta \cdot \epsilon$. Alors, ou bien e est une valeur, ou bien, quel que soit l'état mémoire μ tel que l'on ait $\Sigma \models \mu$, il existe une configuration $(\mu' \cdot e')$ telle que $\mu \cdot e \longrightarrow \mu' \cdot e'$.

Démonstration. Intuitivement, l'essentiel dans la preuve de progrès est de vérifier que lorsqu'une expression bien typée n'est pas une valeur, les prémisses des règles de la

sémantique opérationnelle qui s'appliquent (et lorsque la sémantique est déterministe, *la* règle qui s'applique) sont toutes vérifiées. Lorsque les règles en question sont contextuelles, cela se déduit directement par récurrence. Il reste alors à vérifier les prémisses des règles de réduction en tête. Dans les règles (E-Deref) et (E-Assign) pour la lecture et la modification d'une référence, la prémisse $r^\beta \in \operatorname{dom} \mu$ est vraie en vertu de l'hypothèse $\Sigma \models \mu$. Enfin, traitons le cas où e correspond à l'application d'une constante c_0 à des constantes $c_1, \ldots, c_m$. Dans ce cas-là, en vertu de la cohérence des oracles `Typeof` et δ (définition 2.2.1) on déduit, d'une part, que le type de c_0 est de la forme

$$\texttt{Typeof}(c_0) = \nu_1^\perp \xRightarrow{\perp} \nu_2^\perp \xRightarrow{\perp} \ldots \xRightarrow{\perp} \nu_m^\perp \xRightarrow{\perp} \nu_0,$$

et d'autre part, que l'entier m est inférieur à l'arité du symbole c_0. Par conséquent, lorsque l'application est partielle, elle se réduit par la règle (E-Op-λ), et lorsque l'application est totale, la constante $\delta(c_0, c_1, \ \ldots, \ c_m)$ est bien définie, ce qui permet de conclure. □

Donnons maintenant l'énoncé du lemme de préservation :

Lemma 2.2.2 (Préservation)**.** Quel que soit le pas d'évaluation $\mu \cdot e \longrightarrow \mu' \cdot e'$,

$$\begin{array}{c}\Gamma \cdot \Sigma \vdash e : \tau \cdot \beta \cdot \epsilon \\ \Sigma \models \mu\end{array} \quad \Longrightarrow \quad \exists\, \epsilon' \leq \epsilon.\ \exists\, \beta' \leq \beta. \quad \begin{array}{c}\Gamma \cdot \Sigma \vdash e' : \tau \cdot \beta' \cdot \epsilon' \\ \Sigma \models \mu'\end{array}$$

Démonstration. Le lemme de préservation demande un peu plus d'attention. En particulier, nous avons besoin d'un lemme auxiliaire pour la substitution des variables dans une expression. En effet, lorsque le pas de réduction $\mu \cdot e \longrightarrow \mu' \cdot e'$ effectue une β-réduction et donc une substitution d'une variable dans une expression, il faut montrer que l'expression qui en résulte reste bien typée. Énonçons donc le lemme de substitution des variables :

Lemma 2.2.3 (Lemme de substitution)**.** Soient deux jugements du typage valides $[x^{\beta_0} \mapsto \tau_1]\Gamma \cdot \Sigma \vdash e_2 : \tau_2 \cdot \beta_2 \cdot \epsilon_2$ et $\Gamma \cdot \Sigma \vdash a : \tau_1 \cdot \beta_1 \cdot \bot$ tels que $(\epsilon_2 \vee \epsilon^+(\tau_2) \wedge \neg\beta_0) \implies \neg\beta_1$. Alors le jugement $\Gamma \cdot \Sigma \vdash e_2[x^{\beta_0}/a] : \tau_2 \cdot \beta_3 \cdot \epsilon_2$ est dérivable et vérifie :

i) $\beta_0 = \bot \Rightarrow \beta_3 \leq (\beta_1 \vee \beta_2)$

ii) $\beta_0 = \top \Rightarrow \beta_3 \leq \beta_2$

Démonstration. Notons que l'hypothèse $(\epsilon_2 \vee \epsilon^+(\tau_2) \wedge \neg\beta_0) \implies \neg\beta_1$ est bien nécessaire. En effet, d'après l'énoncé du lemme, lorsque l'on substitue une variable réelle par une expression fantôme, le statut fantôme de $e_2[x^{\beta_0}/a]$ peut devenir strictement plus grand que le statut de e_2. Or, si e_2 possédait les effets (y compris les effets latents) non vides, l'invariant du typage (2.1) tomberait en défaut après la substitution. D'autre part, il est important de distinguer dans la conclusions les deux cas *i)* et *ii)*. En effet, lorsque la variable que l'on substitue est réelle ($\beta_0 = \bot$), le statut fantôme

de l'expression dans laquelle on la substitue peut dépendre de β (par exemple, lorsque e_2 est la variable $x^{\bot}$ elle-même). Rien n'empêche alors que l'on substitue x par une variable fantôme $y^{\top}$, de sorte que le statut fantôme de $e_2[x^{\bot}/y^{\top}]$ devient plus grand que le statut de e_2 mais pas plus grand que le statut de $\beta_1 \vee \beta_2$. Cela garantît donc que le statut fantôme du résultat de la β-réduction ne sera pas plus grand que le statut du rédex. Montrons donc le lemme de substitution par récurrence sur la dérivation de $[x^{\beta_0} \mapsto \tau_1]\Gamma \cdot \Sigma \vdash e_2 : \tau_2 \cdot \beta_2 \cdot \epsilon_2$.

Démonstration du lemme de substitution. Le résultat est immédiat dans les cas où e est une constante c ou une expression de la forme $!r^{\beta}$. Les cas où e est une conditionnelle, une application ou est de la forme `ghost` e_2' se déduisent en appliquant directement les hypothèses de récurrence aux prémisses des règles de typage respectives. Lorsque e est une fonction anonyme, une définition récursive, ou une liaison locale, le résultat du lemme se déduit également par récurrence, en utilisant par ailleurs les lemmes standard d'affaiblissement et de permutation (cf. ([13, p.106]) qui ne présentent aucune difficulté particulière ici et que nous ne détaillons pas. En revanche, traitons en détails les cas restants où e est une variable ou une affectation.

Cas de la règle (T-Var). Lorsque e_2 est une variable y^{β_2} différente de x^{β}, la substitution $e_2[x^{\beta_0}/a]$ n'a aucun effet et il suffit donc de prendre $\beta_3 = \beta_2$. Sinon, $e_2[x^{\beta_0}/a] = a$, au quel cas on peut prendre $\beta_3 = \beta_1$. En effet, si $\beta_0 = \bot$, le résultat est vérifié, car $\beta_1 \leq \beta_1 \vee \beta_2$. Sinon, lorsque $\beta_0 = \top$, le résultat est encore vérifié car $\beta_2 = \beta_0$ et donc $\beta_1 \leq \top$.

Cas de la règle (T-Assign). D'après les hypothèses, la dérivation du typage de e se termine par

$$\frac{\Sigma(r^{\beta_2}) = \nu \qquad \Gamma' \cdot \Sigma \vdash a_1 : \nu \cdot \beta_2' \cdot \bot \qquad \beta_2' \leq \beta_2}{\Gamma' \cdot \Sigma \vdash (r^{\beta_2} := a_1) : \texttt{Unit} \cdot \beta_2 \cdot \neg\beta_2}\,(\text{T-Assign})$$

où $\Gamma' = [x^{\beta_0} \mapsto \tau_1]\Gamma$. Par récurrence, on obtient le jugement $\Gamma \cdot \Sigma \vdash a_1[x^{\beta_0}/a] : \nu \cdot \beta_3 \cdot \bot$ avec β_3 vérifiant $\beta_0 = \bot \Rightarrow \beta_3 \leq (\beta_1 \vee \beta_2')$ et $\beta_0 = \top \Rightarrow \beta_3 \leq \beta_2'$. Lorsque $\beta_0 = \top$, on a par transitivité $\beta_3 \leq \beta_2$ et on conclut en appliquant la règle (T-Assign). Sinon, $\beta_0 = \bot$ et alors on a deux cas à considérer. Si la référence r^{β_2} est fantôme, on peut conclure directement en appliquant la règle (T-Assign). Sinon, lorsque la référence est réelle, ses effets sont non-vides ($\neg\beta_2 = \top$). D'après l'hypothèse du lemme, on a alors nécessairement $\beta_1 = \bot$. Par conséquent, on a de nouveau $\beta_3 \leq \beta_2'$, ce qui permet de conclure. □

Maintenant prouvons le lemme de préservation lui-même par récurrence sur la dérivation de $\mu \cdot e \longrightarrow \mu' \cdot e'$ avec analyse par cas sur la dernière règle appliquée.

Cas de la règle (E-Ghost). D'après les hypothèses, e qui est de la forme `ghost` e' se réduit vers e' ayant le même type et les mêmes effets vides que e. De plus, le statut fantôme de e' est plus petit que le statut de e qui est égal à $\top$. Enfin, le pas de réduction de e ne modifie pas l'état mémoire, lequel reste donc bien typé.

Cas de la règle (E-Op-λ). D'après la règle, l'expression e correspond à l'application d'une constante c_0 aux constantes $c_1 \ldots c_k$ avec $1 \leq k < \text{arity}(c_0)$. Puisqu'il s'agit d'une application partielle, e étant est bien typée, le type de c_0, donné par l'oracle `Typeof`, est un type fonctionnel de la forme

$$\nu_1^\perp \xRightarrow{\perp} \nu_2^\perp \xRightarrow{\perp} \ldots \xRightarrow{\perp} \nu_m^\perp \xRightarrow{\perp} \nu_0.$$

D'autre part, e se réduit en une fonction anonyme $\lambda x^\beta : \nu.\ c_0\, c_1 \ldots c_k\ x^\beta$. Le type ν et le statut fantôme du paramètre formel x peuvent être choisi arbitrairement, posons donc $\nu = \nu_{k+1}$ et $\beta = \perp$. Il suffit alors de remarquer que e' est une η-expansion de e et donc que leurs types coïncident, ce qui permet de conclure.

Cas de la règle (E-Op-δ). On a toujours $e = c_0\ c_1 \ldots c_m$ mais cette fois $m = \text{arity}(c_0)$, c'est-à-dire que l'application de c_0 à $c_1 \ldots c_m$ est totale. On en déduit que le type de c_0 est nécessairement de la forme

$$\nu_1^\perp \xRightarrow{\perp} \nu_2^\perp \xRightarrow{\perp} \ldots \xRightarrow{\perp} \nu_m^\perp \xRightarrow{\perp} \nu_0$$

où $\texttt{Typeof}(c_i) = \nu_i$ pour $1 \leq i \leq m$ et ν_0 est le type de e. Or, puisque les oracles `Typeof` et δ sont cohérents d'après la définition 2.2.1, on a $\texttt{Typeof}(\delta(c_0, c_1,\ \ldots,\ c_m)) = \nu_0$ ce qui permet de conclure.

Cas des règles (E-True) *et* (E-False). Nous avons $e = \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ qui s'évalue en e_1 lorsque v est égale à `true` et en e_2 sinon. On peut alors conclure directement, en remarquant que le statut fantôme et les effets dans une conditionnelle sont toujours plus grands que le statut fantôme et les effets de chacune de ses branches.

Cas de la règle (E-Deref). Le pas de la réduction est de la forme $\mu \cdot !r^\beta \longrightarrow \mu \cdot \mu(r^\beta)$. L'état mémoire μ étant par hypothèse bien typé par Σ, on a $\Sigma(r^\beta) = \texttt{Typeof}(\mu(r^\beta))$, et on conclut en appliquant la règle (T-Const) à la constante $\mu(r^\beta)$.

Cas de la règle (E-Assign). Le pas de la réduction est de la forme

$$\mu \cdot r^\beta := c \longrightarrow \mu[r^\beta \mapsto c] \cdot ().$$

La préservation du typage de l'expression est donc trivialement vraie. Vérifions par ailleurs que l'on a bien $\Sigma \models \mu[r^\beta \mapsto c]$. En effet, le typage de e se termine par la règle (T-Assign) dont les prémisses contiennent l'égalité $\Sigma(r^\beta) = \texttt{Typeof}(c)$. De plus, toutes les références sont distinctes, donc le reste du contenu de l'état mémoire n'a pas été modifié. Ainsi, $dom(\mu') = dom(\mu) \subseteq dom(\Sigma)$ et pour chaque $r^\beta \in dom(\mu')$, on a $\texttt{Typeof}(\mu'(r^\beta)) = \Sigma(r^\beta)$, ce qui permet de conclure.

Cas de la règle (E-Ctx-Let). D'après les hypothèses, le pas de la réduction est de la forme

$$\mu \cdot (\texttt{let}\ x^\beta = e_1\ \texttt{in}\ e_2) \longrightarrow \mu' \cdot (\texttt{let}\ x^\beta = e_1'\ \texttt{in}\ e_2)$$

où $\mu' \cdot e_1'$ provient de la prémisse $\mu \cdot e_1 \longrightarrow \mu' \cdot e_1'$ de la règle (E-Ctx-Let). On a deux cas à considérer selon que x^β est une variable fantôme ou pas. Lorsque $\beta = \perp$, la dérivation du typage de e se termine par :

$$\frac{\Gamma \cdot \Sigma \vdash e_1 : \tau_1 \cdot \beta_1 \cdot \epsilon_1 \qquad [x^{\perp} \mapsto \tau_1]\Gamma \cdot \Sigma \vdash e_2 : \tau_2 \cdot \beta_2 \cdot \epsilon_2}{\Gamma \cdot \Sigma \vdash (\texttt{let}\ x^{\perp} = e_1\ \texttt{in}\ e_2) : \tau_2 \cdot \beta_1 \vee \beta_2 \cdot \epsilon_1 \vee \epsilon_2}(\text{T-Let}^{\perp})$$

En appliquant l'hypothèse de récurrence à la réduction $\mu \cdot e_1 \longrightarrow \mu' \cdot e_1'$, on en déduit qu'il existe $\beta_1' \leq \beta_1$ et $\epsilon_1' \leq \epsilon_1$ tels que $\Gamma \cdot \Sigma \vdash e_1' : \tau_1 \cdot \beta_1' \cdot \epsilon_1'$ et que $\Sigma \models \mu'$. Il suffit donc de refaire le typage de $\texttt{let}\ x^{\perp} = e_1'\ \texttt{in}\ e_2$ par la règle (T-Let$^{\perp}$) ce qui permet de conclure, car $(\beta_1' \vee \beta_2) \leq (\beta_1 \vee \beta_2)$ et $(\epsilon_1' \vee \epsilon_2) \leq (\epsilon_1 \vee \epsilon_2)$. Sinon, lorsque $\beta = \top$, la dérivation du typage de e se termine par l'application de la règle (T-Let$^{\top}$). L'hypothèse de récurrence donne donc de nouveau $\Sigma \models \mu'$ et le fait que e_1' est bien typée. La règle (T-Let$^{\top}$) permet alors dériver le typage de $\texttt{let}\ x^{\top} = e_1'\ \texttt{in}\ e_2$. De plus, seuls les effets et le statut fantôme de e_2 sont pris en compte, donc les conditions $\epsilon' \leq \epsilon$ et $\beta' \leq \beta$ sont trivialement vraies, ce qui permet de conclure. Le cas de la règle (E-Ctx-App) se démontre de la même manière que le cas précédent.

Cas des règles (E-App-λ), (E-App-Rec), (E-ζ). Dans chacun des trois cas, la préservation du typage et le fait que les effets et les statuts fantômes décroissent s'obtiennent en appliquant le lemme de substitution des variables. En effet, notons β_0 le statut fantôme du paramètre formel (resp. de lieur dans le cas (E-ζ)) et β_1 le statut fantôme de l'argument actuel (resp. de la valeur liée dans le cas (E-ζ)). Lorsque $\beta_0 = \perp$, la dérivation du typage de e se termine par la règle (T-App$^{\perp}$) si e est une application et par la règle (T-Let$^{\perp}$) dans le cas de la règle (E-ζ). Or, dans les deux cas, en vertu de la condition implicite de l'invariant du typage, on a bien $(\beta \vee \beta_1) = \perp$ lorsque $\epsilon \vee \epsilon^{+}(\tau) = \top$. En particulier, $\beta_1 = \perp$ et donc l'hypothèse du lemme de substitution

$$(\epsilon \vee \epsilon^{+}(\tau) \wedge \neg\beta_0) \implies \neg\beta_1$$

est vérifiée, ce qui permet d'utiliser le lemme pour les trois cas mentionnés. De plus, aucune de ces règles ne modifie l'état mémoire, ce qui permet de conclure. □

2.2.3 Cohérence des effets statiques avec les effets observables

Le progrès et la préservation du typage constituent ensemble l'argument de la sûreté. Pour garantir la non-interférence, il nous faut aussi montrer que les effets donnés statiquement par le typage reflètent correctement les effets observés pendant l'évaluation. Concrètement, nous montrons qu'un programme dont les effets sont vides ($\epsilon = \perp$), s'évalue en une valeur en nombre fini de pas de réductions sans modifier le contenu des références réelles.

Lemma 2.2.4 (Préservation de la mémoire réelle)**.** Soit une configuration $\mu \cdot e$ telle que l'on ait $\Gamma \cdot \Sigma \vdash e : \tau \cdot \beta \cdot \perp$ et $\Sigma \models \mu$. Notons $\mu_{\perp}$ la partie réelle de μ, c'est-à-dire la restriction de μ à des références réelles. Quel que soit le pas de réduction $\mu \cdot e \longrightarrow \mu' \cdot e'$, on a $\mu_{\perp} = \mu'_{\perp}$.

Démonstration. Par récurrence sur la relation de l'évaluation. Dans toutes les règles de sémantique, sauf les règles (E-Assign), (E-Ctx-Let) et (E-Ctx-App), l'état mémoire reste inchangé et donc le lemme est trivialement vrai. Traitons les trois cas restants.

Cas de la règle (E-Assign). D'après les hypothèses, le pas de réduction est de la forme

$$\mu \cdot (r^\beta := c) \longrightarrow \mu[r^\beta \mapsto c] \cdot ()$$

Par hypothèse, $\epsilon = \bot$, donc d'après la règle (T-Assign), la référence modifiée est nécessairement fantôme et, par conséquent, on a bien $\mu_\bot = \mu'_\bot$.

Cas de la règle (E-Ctx-Let). D'après les hypothèses, le pas de réduction est de la forme

$$\mu \cdot \texttt{let } x^\beta = e_1 \texttt{ in } e_2 \longrightarrow \mu' \cdot \texttt{let } x^\beta = e'_1 \texttt{ in } e_2$$

où μ' et e'_1 sont donnés par la prémisse de la règle $\mu \cdot e_1 \longrightarrow \mu' \cdot e'_1$. Lorsque la dérivation du typage de e se termine par la règle (T-Let$^\top$), d'après la prémisse de la règle, les effets de e_1 sont vides. Sinon, dans le cas de la règle (T-Let$^\bot$), les effets de e, qui sont vides par hypothèse, sont égaux à l'union des effets de e_1 et de e_2. On en déduit que les effets de e_1 sont vides dans ce cas-là aussi. Dans les deux cas, on peut donc appliquer l'hypothèse de récurrence à la réduction $\mu \cdot e_1 \longrightarrow \mu' \cdot e'_1$, ce qui permet de conclure. Le cas de la règle (E-Ctx-App) se fait exactement de la même manière. □

Lemma 2.2.5 (Terminaison des programmes sans effets)**.** Soit une configuration $\mu \cdot e$ telle que l'on ait $\Gamma \cdot \Sigma \vdash e : \tau \cdot \beta \cdot \bot$ et $\Sigma \models \mu$. Alors, l'évaluation à partir de la configuration $\mu \cdot e$ termine, c'est-à-dire qu'il existe une valeur v et un état mémoire μ' tels que $\mu \cdot e \longrightarrow^\star \mu' \cdot v$.

Démonstration. Supposons d'abord que l'évaluation de e ne fait pas intervenir d'appels de fonctions récursives. Remarquons que, d'une part, l'ensemble des références globales est fini et, d'autre part, que chaque référence (quel que soit son statut fantôme) contient nécessairement une valeur d'un type scalaire. Par conséquent, en utilisant la monade d'état $M\ \tau \triangleq \nu_1 \times \cdots \times \nu_n \to \tau \times \nu_1 \times \cdots \times \nu_n$, le programme e peut être traduit dans un terme du lambda calcul simplement typé dont la normalisation est bien établie dans la littérature [19] ([13, ch.12]).

Maintenant, supposons par l'absurde que lorsqu'elle fait intervenir un appel de fonction récursive, l'évaluation de e diverge. Puisque e a les effets $\bot$, les seuls appels de fonctions récursives que l'évaluation de e rencontre utilisent les fonctions récursives pour lesquelles l'oracle (dont on a admis la correction) dit qu'elles sont convergentes. Substituons alors dans $\mu \cdot e \longrightarrow_m \infty$ chaque sous-suite finie de pas de réductions correspondant à un appel récursif par $(\lambda x.x)\ v$ où v est la valeur de résultat de l'appel. Après les substitutions, l'évaluation de e ne contient donc plus aucun appel récursif. Or, l'évaluation partant de e est toujours infinie, ce qui contredit le paragraphe précédent. □

2.3 Effacement du code fantôme

Dans cette section, nous définissons l'opération d'effacement du code fantôme qui transforme une expression bien typée de GhostML en une expression bien typée dans MiniML, un langage avec les mêmes constructions de syntaxe, la même sémantique et le même typage que ceux de GhostML mais sans code fantôme, c'est-à-dire dans lequel tous les statuts fantômes β sont omis et la construction `ghost` n'existe plus. L'objectif est de montrer que le code fantôme peut être effacé des programmes réels sans que cela affecte leur comportement. Dans la suite, on notera l'opération de l'effacement soit par $\epsilon_\beta(.)$, lorsqu'elle est paramétrée par un statut fantôme β, ou bien $\mathcal{E}(.)$ sinon. Par ailleurs, on note les pas de réduction dans MiniML sous la forme $\mu \cdot e \longrightarrow_m \mu' \cdot e'$. De même, on utilise la lettre m pour désigner les jugements du typage en MiniML, en écrivant $\Gamma \cdot \Sigma \vdash_m e : \tau \cdot \epsilon$.

2.3.1 Opération d'effacement du code fantôme

Définissons d'abord ce qu'est l'effacement des types et des expressions de GhostML. L'idée est que l'effacement préserve la structure des types et des expressions réels, en remplaçant le code fantôme par une valeur du type `Unit`.

Définition 2.3.1 (Effacement des types)**.** Soit τ un type de GhostML. On définit l'opération effacement de τ, paramétrée par un statut fantôme β, par induction sur la structure de τ :

$$\begin{array}{lcl} \mathcal{E}_\top(\tau) & \triangleq & \texttt{Unit} \\ \mathcal{E}_\bot({\tau_2}^{\beta_2} \xrightarrow{\epsilon} \tau_1) & \triangleq & \mathcal{E}_{\beta_2}(\tau_2) \xrightarrow{\epsilon} \mathcal{E}_\bot(\tau_1) \\ \mathcal{E}_\bot(\nu) & \triangleq & \nu \end{array}$$

Définissons ensuite l'opération d'effacement des expressions :

Définition 2.3.2 (Effacement des expressions)**.** Soit e une expression GhostML telle que $\Gamma \cdot \Sigma \vdash e : \tau \cdot \beta \cdot \epsilon$. On définit l'opération d'effacement de e, paramétrée par son statut fantôme β, par induction sur la structure de e. Lorsque e est une expression fantôme ($\beta = \top$), on pose

$$\mathcal{E}_\top(e) \quad \triangleq \quad ().$$

Sinon, e est une expression réelle ($\beta = \bot$) et alors on pose :

$$
\begin{array}{lcl}
\mathcal{E}_\bot(c) & \triangleq & c \\
\mathcal{E}_\bot(x^\bot) & \triangleq & x \\
\mathcal{E}_\bot(\lambda x^\beta : \tau.\, e_1) & \triangleq & \lambda x : \mathcal{E}_\beta(\tau).\, \mathcal{E}_\bot(e_1) \\
\mathcal{E}_\bot(\texttt{rec}\ f^\bot : {\tau_2}^{\beta_2} \xRightarrow{\epsilon} \tau_1.\, e_1) & \triangleq & \texttt{rec}\ f : \mathcal{E}_\bot({\tau_2}^{\beta_2} \xRightarrow{\epsilon} \tau_1).\, \mathcal{E}_\bot(e_1) \\
\mathcal{E}_\bot(r^\bot := a) & \triangleq & r := \mathcal{E}_\bot(a) \\
\mathcal{E}_\bot(!r^\bot) & \triangleq & !r \\
\mathcal{E}_\bot(\texttt{if}\ a\ \texttt{then}\ e_1\ \texttt{else}\ e_2) & \triangleq & \texttt{if}\ \mathcal{E}_\bot(a)\ \texttt{then}\ \mathcal{E}_\bot(e_1)\ \texttt{else}\ \mathcal{E}_\bot(e_2) \\
\mathcal{E}_\bot(e_1\ a_2) & \triangleq & \mathcal{E}_\bot(e_1)\ \mathcal{E}_{\beta'}(a_2) \quad \text{où le type de } e_1 \text{ est } {\tau_2}^{\beta'} \xRightarrow{\epsilon_1} \tau_1 \\
\mathcal{E}_\bot(\texttt{let}\ x^{\beta'} = e_1\ \texttt{in}\ e_2) & \triangleq & \texttt{let}\ x = \mathcal{E}_{\beta'}(e_1)\ \texttt{in}\ \mathcal{E}_\bot(e_2)
\end{array}
$$

Notons qu'une fonction réelle (qu'elle soit récursive ou non) dont le paramètre formel est fantôme reste une fonction mais le type de son paramètre formel devient `Unit`, quel qu'eût été son type avant l'effacement. De la même manière, lorsque e est une liaison locale dont le lieur $x^\top$ est fantôme, $\mathcal{E}_\bot(e)$ reste une expression `let` cependant elle lie désormais la variable x à `()`. En particulier, on peut remarquer que les occurrences des variables et des références fantômes n'apparaissent plus dans $\mathcal{E}_\bot(e)$, car elles sont toutes remplacées par une valeur du type `()`. D'autre part, on montre par récurrence immédiate sur la structure de e que $\mathcal{E}_\bot(e)$ est une valeur si et seulement si e est une valeur (ce qui n'est bien entendu, pas vrai pour $\mathcal{E}_\top(e)$).

Rappelons que les variables et les lieurs fantômes ont tous des noms distincts des variables et lieurs réels. En effet, on aurait sinon une traduction incorrecte :

$$\mathcal{E}_\bot(\lambda x^\bot : \texttt{Int}.\, \lambda x^\top : \mathit{Bool}.\, x^\bot) = \lambda x : \texttt{Int}.\, \lambda x : \texttt{Unit}.\, x$$

Il peut sembler inutile de laisser dans le programme des « traces » du code fantôme sous forme d'arguments et de valeurs de type `Unit` après son effacement. Cependant, il n'est pas possible de s'en débarrasser complètement. Considérons par exemple la fonction f définie par

$$\lambda x^\bot : \texttt{Int}.\, \lambda y^\top : \texttt{Int}.\, r^\bot := x.$$

Clairement, l'application de f à un argument quelconque est partielle en GhostML et donc les effets d'écriture vis-à-vis de $r^\bot$ restent latents. Or, si l'on effaçait le code fantôme complètement, en transformant la fonction f en $\lambda x^\bot : \texttt{Int}.\, r^\bot := x$, la même application à f dans MiniML deviendrait alors totale, réalisant ainsi les effets d'écriture pour $r^\bot$. Le programme réel GhostML et son effacement n'auraient donc pas le même comportement observable. D'une part, il est intéressant de noter que cela correspond à ce qui se passe en Coq lorsque l'on souhaite extraire un programme à partir du λ-terme

$$\lambda x.\, \lambda \pi : \neg(x = 0).\, 1/x$$

où il serait incorrect d'en extraire le terme $\lambda x.\, 1/x$, car l'application partielle à 0, correcte avant l'extraction, produirait une erreur de division par zéro après l'extraction.

D'autre part, il s'avère que présenter l'effacement comme un morphisme, en gardant toutes les « traces » du code fantôme, est utile pour faciliter la preuve de sa correction. Remarquons, en passant, que ce genre de traduction qui rajoute des pas de réductions « factices » est une technique connue dans la littérature sous le nom des « réductions administratives » ("administrative reduction steps" en anglais [14]) dont on peut montrer qu'elles n'ont pas d'impact sur la complexité du programme et qu'elles peuvent être éliminées par le compilateur pendant les phases d'optimisations ultérieures.

Enfin, étendons l'opération effacement sur d'autres objets que nous avons introduits : les environnements du typage et les états mémoire. En ce qui concerne Γ, pour toute variable $x^\top$, on pose $\mathcal{E}(\Gamma)(x) = \texttt{Unit}$ et, pour toute variable $x^\bot$, on pose $\mathcal{E}(\Gamma)(x) = \mathcal{E}_\bot(\tau)$ où $\Gamma(x^\bot) = \tau$. Comme les noms des variables fantômes et des variables réelles sont distincts, il n'y a pas d'ambiguïté dans la définition. On a en particulier

$$\mathcal{E}([x^\beta \mapsto \tau]\Gamma) = [x \mapsto \mathcal{E}_\beta(\tau)]\mathcal{E}(\Gamma).$$

Quant à l'environnement Σ, on restreint le domaine de $\mathcal{E}(\Sigma)$ au sous-ensemble des références globales réelles et, pour chaque $r^\bot \in \operatorname{dom}\Sigma$, on pose $\mathcal{E}(\Sigma)(r) = \Sigma(r^\bot)$.

D'une manière similaire, pour définir l'effacement d'un état mémoire, que l'on note $\mathcal{E}(\mu)$, on restreint $\operatorname{dom}\mathcal{E}(\mu)$ au sous-ensemble des références globales réelles et pour chaque $r^\bot \in \operatorname{dom}\mu$, on pose $\mathcal{E}(\mu)(r) = \mu(r^\bot)$. Notons que $\mathcal{E}(\mu)$ efface les statuts fantômes. Il s'agit donc d'un objet syntaxiquement différent de $\mu_\bot$ (la restriction de μ à des références réelles) mais, bien entendu, on a $\mu_\bot = \mu'_\bot$ si et seulement si $\mathcal{E}(\mu) = \mathcal{E}(\mu')$. Notons également que $\Sigma \models \mu$ implique bien $\mathcal{E}(\Sigma) \models_m \mathcal{E}(\mu)$. Enfin, lorsque e est bien typée et son statut fantôme est β, on note $\mathcal{E}_\beta(\mu \cdot e)$ pour désigner la configuration $\mathcal{E}(\mu) \cdot \mathcal{E}_\beta(e)$ dans MiniML.

2.3.2 Correction de l'effacement

Le but de cette section est de montrer que l'évaluation du code réel est préservée par l'opération d'effacement. Qu'entendons-nous par cela ? Lorsque l'on part d'une configuration $\mu \cdot e$ où e est un programme réel bien typé, nous savons déjà que soit il existe un état mémoire μ' et une valeur v telle que $\mu \cdot e \longrightarrow^\star \mu' \cdot v$, soit $\mu \cdot e \longrightarrow \infty$. La préservation de l'évaluation par l'effacement signifie simplement que, dans MiniML, on a respectivement $\mathcal{E}_\bot(\mu \cdot e) \longrightarrow_m^\star \mathcal{E}_\bot(\mu' \cdot v)$ dans le premier cas et $\mathcal{E}_\bot(\mu \cdot e) \longrightarrow_m \infty$ dans le second.

Pour démonter la préservation de l'évaluation par l'effacement, nous avons besoin de montrer d'abord que l'opération de l'effacement préserve la relation du typage du code réel :

Théorème 2.3.1 (préservation du typage par l'effacement)**.** Soit e un programme réel tel que $\Gamma \cdot \Sigma \vdash e : \tau \cdot \bot \cdot \epsilon$ soit dérivable. Alors le jugement $\mathcal{E}(\Gamma) \cdot \mathcal{E}(\Sigma) \vdash_m \mathcal{E}_\bot(e) : \mathcal{E}_\bot(\tau) \cdot \epsilon$ est dérivable.

Démonstration. Par récurrence sur la dérivation du jugement de typage. Les cas de la règle (T-Const) est trivial et le cas de la règle (T-Ghost) est impossible. Le cas des règles (T-Var) et (T-Deref) se déduisent directement à partir des définitions de l'effacement pour les environnements du typage.

Cas de la règle (T-Assign). L'expression e étant réelle, on déduit que la dérivation du typage de e se termine par :

$$\frac{\Gamma \cdot \Sigma \vdash a : \nu \cdot \beta' \cdot \bot \quad \Sigma(r^\beta) = \nu \quad \beta' \leq \bot}{\Gamma \cdot \Sigma \vdash r^\beta := a : \texttt{Unit} \cdot \bot \cdot \top}\,(\text{T-Assign})$$

où l'on a donc $\beta' = \bot$. L'hypothèse de récurrence s'applique alors au jugement du typage de a et on conclut par :

$$\frac{\mathcal{E}(\Gamma) \cdot \mathcal{E}(\Sigma) \vdash_m a : \nu \cdot \bot \quad \mathcal{E}(\Sigma)(r) = \nu}{\mathcal{E}(\Gamma) \cdot \mathcal{E}(\Sigma) \vdash_m r := a : \texttt{Unit} \cdot \top}$$

Le raisonnement pour la règle (T-Rec) suit le même schéma. La fonction récursive est réelle par hypothèse, on déduit donc que la fonction anonyme de sa définition est réelle également ce qui permet d'appliquer l'hypothèse de récurrence et conclure directement en reconstruisant la dérivation du typage en MiniML.

Cas de la règle (T-App$^\top$). On déduit que la dérivation du typage de e se termine par :

$$\frac{\Gamma \cdot \Sigma \vdash e_1 : {\tau_2}^\top \overset{\epsilon_0}{\Longrightarrow} \tau_1 \cdot \bot \cdot \epsilon_1 \qquad \Gamma \cdot \Sigma \vdash a_2 : \tau_2 \cdot \beta_2 \cdot \bot}{\Gamma \cdot \Sigma \vdash (e_1\ a_2) : \tau_1 \cdot \bot \cdot \epsilon_0 \vee \epsilon_1}\,(\text{T-App}^\top)$$

Par définition de $\mathcal{E}_\bot(\cdot)$, on a $\mathcal{E}_\bot(e_1\ a_2) = \mathcal{E}_\bot(e_1)\ ()$. Il suffit alors d'appliquer l'hypothèse de récurrence au jugement du typage de e_1 et reconstruire ensuite le typage de l'application en MiniML :

$$\frac{\mathcal{E}(\Gamma) \cdot \mathcal{E}(\Sigma) \vdash_m \mathcal{E}_\bot(e_1) : \texttt{Unit} \overset{\epsilon_0}{\Longrightarrow} \mathcal{E}_\bot(\tau_1) \cdot \epsilon_1 \qquad \mathcal{E}(\Gamma) \cdot \mathcal{E}(\Sigma) \vdash_m () : \texttt{Unit} \cdot \bot}{\mathcal{E}(\Gamma) \cdot \mathcal{E}(\Sigma) \vdash_m (\mathcal{E}_\bot(e_1)\ ()) : \mathcal{E}_\bot(\tau_1) \cdot \epsilon_0 \vee \epsilon_1}$$

Le cas de la règle (T-Let$^\top$) pour $\texttt{let}\ x^\top = e_1\ \texttt{in}\ e_2$ est identique au cas précédent. Enfin, dans les cas des règles (T-If), (T-λ), (T-App$^\bot$) et (T-Let$^\bot$), l'expression e étant du code réel, on en déduit que les sous-expressions sont toutes réelles, ce qui permet d'appliquer l'hypothèse de récurrence et de reconstruire la dérivation de typage correspondante en MiniML. □

Pour montrer que l'effacement préserve la relation d'évaluation, nous allons utiliser la technique de bisimulation que les deux diagrammes de la figure 2.5 résument (les flèches en noir correspondent aux hypothèses et celles en rouge aux conclusions) :

Pour montrer les résultats de ces deux diagrammes, on a besoin du lemme suivant de commutativité de l'opération de l'effacement avec la substitution syntaxique des variables :

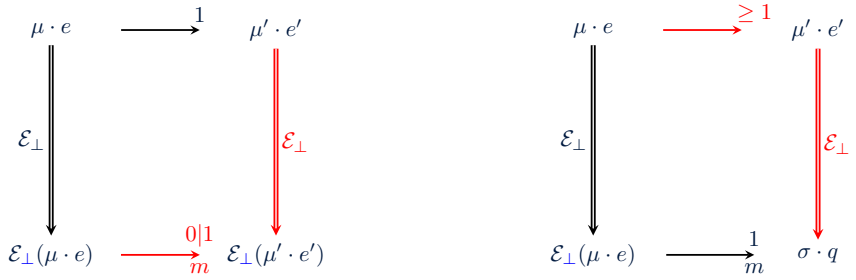


FIGURE 2.5 – Simulation en avant d'un pas d'évaluation de GhostML dans MiniML (à gauche) et d'un pas d'évaluation de MiniML dans GhostML (à droite).

Lemma 2.3.1 (Substitution sous l'effacement)**.** Soit e et a deux expressions telles que $[x^\beta \mapsto \tau]\Gamma \cdot \Sigma \vdash e : \tau_0 \cdot \perp \cdot \epsilon$ et $\Gamma \cdot \Sigma \vdash a : \tau \cdot \beta' \cdot \perp$ sont dérivables avec $\beta' \leq \beta$. Alors $\mathcal{E}_\perp(e)[x/\mathcal{E}_\beta(a)] = \mathcal{E}_\perp(e[x^\beta/a])$.

Démonstration. Par récurrence sur la structure de e. □

Montrons maintenant le lemme de simulation de GhostML dans MiniML.

Lemma 2.3.2 (Simulation de GhostML dans MiniML)**.** Soit une configuration $\mu \cdot e$ telle que $\Gamma \cdot \Sigma \vdash e : \tau \cdot \perp \cdot \epsilon$ et $\Sigma \models \mu$. Alors quel que soit le pas de réduction $\mu \cdot e \longrightarrow \mu' \cdot e'$, on a dans MiniML : $\mathcal{E}_\perp(\mu \cdot e) \longrightarrow_m^{0|1} \mathcal{E}_\perp(\mu' \cdot e')$.

Démonstration. Par récurrence sur la relation de l'évaluation.

Cas de la règle (E-APP-λ). D'après les hypothèses, le pas de la réduction a la forme $(\lambda x^\beta : \tau_2.\, e_1)\ v \longrightarrow e_1[x^\beta \leftarrow v]$. Si le paramètre formel est fantôme ($\beta = \top$), on déduit que la dérivation du typage de e se termine par

$$\frac{\Gamma \cdot \Sigma \vdash \lambda x^\top : \tau_2.\, e_1 : {\tau_2}^\top \overset{\epsilon_1}{\Longrightarrow} \tau_1 \cdot \perp \cdot \perp \quad \Gamma \cdot \Sigma \vdash v : \tau_2 \cdot \beta_2 \cdot \perp}{\Gamma \cdot \Sigma \vdash (\lambda x^\top : \tau_2.\, e_1\ v) : \tau_1 \cdot \perp \cdot \epsilon_1}(\text{T-APP}^\top)$$

Par définition, $\mathcal{E}_\perp((\lambda x^\top : \tau_2.e_1)\ v) = (\lambda x : \texttt{Unit}.\mathcal{E}_\perp(e_1))\ ()$. D'après le lemme 2.3.1 on a $\mathcal{E}_\perp(e_1)[x/()] = \mathcal{E}_\perp(e_1[x^\top/v])$, ce qui permet de conclure, puisque le pas de réduction (E-APP-λ) ne modifie pas l'état mémoire. Lorsque le paramètre formel est réel ($\beta = \perp$), on déduit que la dérivation du typage de e se termine par

$$\frac{\Gamma \cdot \Sigma \vdash \lambda x^\perp : \tau_2.\, e_1 : {\tau_2}^\top \overset{\epsilon_1}{\Longrightarrow} \tau_1 \cdot \perp \cdot \perp \quad \Gamma \cdot \Sigma \vdash v : \tau_2 \cdot \perp \cdot \perp}{\Gamma \cdot \Sigma \vdash (\lambda x^\perp : \tau_2.\, e_1\ v) : \tau_1 \cdot \perp \cdot \epsilon_1}(\text{T-APP}^\perp)$$

et on procède de la même manière que dans le cas précédent.

Cas de la règle (E-APP-REC). D'après les hypothèses, le pas de la réduction a la forme

$$\mu \cdot v_0\ v \longrightarrow \mu \cdot e_1[f^\perp/v_0][x^\beta/v]$$

où $v_0 = \texttt{rec}\ f^\perp : {\tau_2}^\beta \overset{\epsilon_0}{\Longrightarrow} \tau_1.\, \lambda x^\beta : \tau_2.\, e_1$ Supposons que le paramètre formel est réel ($\beta = \perp$). Sachant que le statut fantôme de e est $\perp$, on déduit que la dérivation du typage de e a la forme suivante :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dots}{[x^{\perp} \mapsto \tau_2][f^{\perp} \mapsto \tau_2{}^{\perp} \xRightarrow{\epsilon_1} \tau_1]\Gamma \cdot \Sigma \vdash e_1 : \tau_1 \cdot \perp \cdot \epsilon_1}}{[f^{\perp} \mapsto \tau_2{}^{\perp} \xRightarrow{\epsilon_1} \tau_1]\Gamma \cdot \Sigma \vdash (\lambda x^{\perp} : \tau_2.\, e_1) : \tau_2{}^{\perp} \xRightarrow{\epsilon_1} \tau_1 \cdot \perp \cdot \perp} \quad \dfrac{\dots}{\Gamma \cdot \Sigma \vdash v : \tau_2 \cdot \perp \cdot \perp}}{\Gamma \cdot \Sigma \vdash v_0 : \tau_2{}^{\perp} \xRightarrow{\epsilon_1} \tau_1 \cdot \perp \cdot \perp}}{\Gamma \cdot \Sigma \vdash v_0\ v : \tau_1 \cdot \perp \cdot \epsilon_1}$$

où $\texttt{CheckTermination}(v_0) = \epsilon_0 \leq \epsilon_1$. Par définition, on a $\mathcal{E}_{\perp}(v_0\ v) = \mathcal{E}_{\perp}(v_0)\ \mathcal{E}_{\perp}(v)$ qui se réduit en MiniML en $\mathcal{E}_{\perp}(e_1)[f/\mathcal{E}_{\perp}(v_0)][x/\mathcal{E}_{\perp}(v)]$. Finalement, d'après le lemme 2.3.1, on a $(\mathcal{E}_{\perp}(e_1))[f/\mathcal{E}_{\perp}(v_0)][x/\mathcal{E}_{\perp}(v)] = \mathcal{E}_{\perp}(e_1[f^{\perp}/v_0][x^{\perp}/v])$ ce qui permet de conclure, puisque le pas de réduction (E-App-Rec) ne modifie pas l'état mémoire. Le cas où le paramètre formel est fantôme $\beta = \top$ se traite de la même manière.

Dans le cas des autres règles de réduction en tête, le raisonnement suit le même schéma.

Cas de la règle (E-Ctx-Let). D'après les hypothèses, le pas de la réduction a la forme suivante :

$$\dfrac{\mu \cdot e_1 \longrightarrow \mu' \cdot e_1'}{\mu \cdot (\texttt{let}\ x^{\beta} = e_1\ \texttt{in}\ e_2) \longrightarrow \mu' \cdot (\texttt{let}\ x^{\beta} = e_1'\ \texttt{in}\ e_2)}\text{(E-Ctx-Let)}$$

Lorsque $\beta = \top$, on déduit que la dérivation du typage de e se termine par

$$\dfrac{\Gamma \cdot \Sigma \vdash e_1 : \tau_1 \cdot \beta_1 \cdot \perp \quad [x^{\top} \mapsto \tau_1]\Gamma \cdot \Sigma \vdash e_2 : \tau_2 \cdot \perp \cdot \epsilon_2}{\Gamma \cdot \Sigma \vdash \texttt{let}\ x^{\top} = e_1\ \texttt{in}\ e_2 : \tau_2 \cdot \perp \cdot \epsilon_2}(\text{T-Let}^{\top})$$

Or, e_1 étant sans effets, d'après le lemme 2.2.4, on a $\mu_{\perp} = \mu'_{\perp}$, donc $\mathcal{E}(\mu) = \mathcal{E}(\mu')$. De plus $\mathcal{E}_{\perp}(\texttt{let}\ x^{\top} = e_1\ \texttt{in}\ e_2) = (\texttt{let}\ x = ()\ \texttt{in}\ \mathcal{E}_{\perp}(e_2)) = \mathcal{E}_{\perp}(\texttt{let}\ x^{\top} = e_1'\ \texttt{in}\ e_2)$ ce qui permet de conclure que

$$\mathcal{E}_{\perp}(\mu \cdot \texttt{let}\ x^{\top} = e_1\ \texttt{in}\ e_2) \longrightarrow_m^0 \mathcal{E}_{\perp}(\mu' \cdot \texttt{let}\ x^{\top} = e_1'\ \texttt{in}\ e_2)$$

Sinon, lorsque $\beta = \perp$, la dérivation du typage de e se termine par

$$\dfrac{\Gamma \cdot \Sigma \vdash e_1 : \tau_1 \cdot \perp \cdot \epsilon_1 \quad [x^{\perp} \mapsto \tau_1]\Gamma \cdot \Sigma \vdash e_2 : \tau_2 \cdot \perp \cdot \epsilon_2}{\Gamma \cdot \Sigma \vdash \texttt{let}\ x^{\perp} = e_1\ \texttt{in}\ e_2 : \tau_2 \cdot \perp \cdot \epsilon_1 \vee \epsilon_2}(\text{T-Let}^{\perp})$$

On applique alors l'hypothèse de récurrence à la réduction de e_1, ce qui donne :

$$\mathcal{E}_{\perp}(\mu \cdot e_1') \longrightarrow_m^{0|1} \mathcal{E}_{\perp}(\mu' \cdot e_1').$$

Le résultat est immédiat lorsque ce pas de la réduction est réalisé en 0 étapes. Sinon, on conclut, en appliquant la règle (E-Ctx-Let) en MiniML :

$$\dfrac{\mathcal{E}(\mu) \cdot \mathcal{E}_{\perp}(e_1) \longrightarrow_m \mathcal{E}(\mu') \cdot \mathcal{E}_{\perp}(e_1')}{\mathcal{E}(\mu) \cdot \texttt{let}\ x = \mathcal{E}_{\perp}(e_1)\ \texttt{in}\ \mathcal{E}_{\perp}(e_2) \longrightarrow_m \mathcal{E}(\mu') \cdot \texttt{let}\ x = \mathcal{E}_{\perp}(e_1')\ \texttt{in}\ \mathcal{E}_{\perp}(e_2)}\text{(E-Ctx-Let)}$$

□

Théorème 2.3.2 (Préservation de l'évaluation convergente)**.** Soit une configuration $\mu \cdot e$ telle que $\Gamma \cdot \Sigma \vdash e : \tau \cdot \perp \cdot \epsilon$ et $\Sigma \models \mu$. Alors quels que soient la valeur v et l'état mémoire μ' tels que $\mu \cdot e \longrightarrow^{\star} \mu' \cdot v$, on a $\mathcal{E}_{\perp}(\mu \cdot e) \longrightarrow_m^{\star} \mathcal{E}_{\perp}(\mu' \cdot v)$.

Démonstration. Par récurrence sur le nombre de pas de réductions de $\mu \cdot e \longrightarrow^\star \mu' \cdot v$. Lorsque $\mu \cdot e \longrightarrow^0 \mu' \cdot v$, le résultat est immédiat. Supposons donc que l'on ait $\mu \cdot e \longrightarrow^1 \mu'' \cdot e'' \longrightarrow^n \mu' \cdot v$ pour une certaine configuration intermédiaire $\mu'' \cdot e''$ avec $n \geq 0$. D'après le lemme 2.2.2, on a $\Sigma \models \mu''$ et $\Gamma \cdot \Sigma \vdash e'' : \tau \cdot \bot \cdot \epsilon''$ avec $\epsilon'' \leq \epsilon'$. Par hypothèse de récurrence sur la dérivation $\mu'' \cdot e'' \longrightarrow^n \mu' \cdot v$ on obtient $\mathcal{E}_\bot(\mu'' \cdot e'') \longrightarrow_m^\star \mathcal{E}_\bot(\mu' \cdot v)$. D'autre part, le lemme 2.3.2 permet d'obtenir le pas de réduction $\mathcal{E}_\bot(\mu \cdot e) \longrightarrow_m^{0|1} \mathcal{E}_\bot(\mu'' \cdot e'')$. En mettant les deux réductions ensemble, on a finalement la réduction $\mathcal{E}_\bot(\mu \cdot e) \longrightarrow_m^{0|1} \mathcal{E}_\bot(\mu'' \cdot e'') \longrightarrow_m^\star \mathcal{E}_\bot(\mu' \cdot v)$. □

Montrons maintenant le lemme de simulation de MiniML dans GhostML.

Lemma 2.3.3 (Simulation de MiniML dans GhostML)**.** Soit une configuration $\mu \cdot e$ telle que $\Gamma \cdot \Sigma \vdash e : \tau \cdot \bot \cdot \epsilon$ et $\Sigma \models \mu$. Quelle que soit la configuration $(\sigma \cdot q)$ de MiniML telle que $\mathcal{E}_\bot(\mu \cdot e) \longrightarrow_m \sigma \cdot q$, on a la réduction $\mu \cdot e \longrightarrow^{\geq 1} \mu' \cdot e'$ en GhostML avec $\mathcal{E}_\bot(\mu' \cdot e') = \sigma \cdot q$.

Démonstration. Par récurrence sur la dérivation de $\mathcal{E}_\bot(\mu \cdot e) \longrightarrow_m \sigma \cdot q$. Rappelons que $\mathcal{E}_\bot(e)$ est une valeur si et seulement si e l'est. Ainsi, puisque $\mathcal{E}_\bot(e)$ se réduit, e n'est pas une valeur. Or, on sait déjà que le système de types de GhostML est correct. Par conséquent, l'expression e est nécessairement réductible. Il nous reste donc à vérifier qu'à partir de la configuration $\mu \cdot e$ il existe un nombre fini de pas de réductions tels que l'effacement de la configuration $\mu' \cdot e'$ résultante correspond à la configuration MiniML $\sigma \cdot q$.

Cas de (E-Ctx-Let) *en MiniML.* La dérivation du pas de la réduction a la forme :

$$\frac{\mathcal{E}(\mu) \cdot \mathcal{E}_\bot(e_1) \longrightarrow_m \mathcal{E}(\mu') \cdot \mathcal{E}_\bot(e_1')}{\mathcal{E}(\mu) \cdot \texttt{let}\ x = \mathcal{E}_\bot(e_1)\ \texttt{in}\ \mathcal{E}_\bot(e_2) \longrightarrow_m \mathcal{E}(\mu') \cdot \texttt{let}\ x = \mathcal{E}_\bot(e_1')\ \texttt{in}\ \mathcal{E}_\bot(e_2)}$$

On en déduit que $\mathcal{E}_\bot(e_1)$ n'est pas une valeur, donc e_1 non plus. De nouveau, par le lemme du progrès, on déduit que e_1 est réductible. On a alors deux cas à considérer, selon que le lieur x est l'effacement d'un lieur fantôme ou d'un lieur réel.

Lorsque x est l'effacement d'un lieur fantôme, on déduit que la dernière règle dans la dérivation du typage de e est (T-Let$^\top$). Par conséquent, le typage de e_1 est nécessairement de la forme $\Gamma \cdot \Sigma \vdash e_1 : \tau_1 \cdot \beta_1 \cdot \bot$. Puisque l'expression e_1 n'a pas d'effets, d'après le lemme 2.2.5, on obtient une séquence finie de réductions $\mu \cdot e_1 \longrightarrow_m^\star \mu' \cdot v$. Cela permet de déduire que l'on a la séquence de pas de réductions suivante :

$$\mu \cdot \texttt{let}\ x^\top = e_1\ \texttt{in}\ e_2 \longrightarrow^\star \mu' \cdot \texttt{let}\ x^\top = v\ \texttt{in}\ e_2 \longrightarrow \mu' \cdot e_2[x^\top/v].$$

Par le lemme de simulation de GhostML dans MiniML, on a alors :

$$\mathcal{E}_\bot(\mu \cdot \texttt{let}\ x^\top = e_1\ \texttt{in}\ e_2) \longrightarrow_m^0 \mathcal{E}_\bot(\mu' \cdot \texttt{let}\ x^\top = v\ \texttt{in}\ e_2) \longrightarrow_m \mathcal{E}_\bot(\mu' \cdot e_2[x^\top/v]).$$

D'une part, d'après le lemme 2.3.1 de substitution sous l'effacement, on a

$$\mathcal{E}_\bot(e_2[x^\top/v]) = \mathcal{E}_\bot(e_2)[x/()].$$

D'autre part, l'expression e_1 étant sans effets, d'après le lemme 2.2.4 de préservation de l'état mémoire, on a $\mu_\bot = \mu'_\bot$ et donc $\mathcal{E}(\mu) = \mathcal{E}(\mu')$, ce qui permet de conclure.

Sinon, e est une expression de la forme $\texttt{let}\ x^\bot = e_1\ \texttt{in}\ e_2$. L'expression $\mathcal{E}_\bot(e_1)$ n'étant pas une valeur, il existe une configuration MiniML $\sigma \cdot q$ telle que $\mathcal{E}_\bot(\mu \cdot e_1) \longrightarrow_m \sigma \cdot q$. Or, puisque la dernière règle dans la dérivation du typage de e est (T-Let$^\bot$), et que e est réelle par hypothèse, on en déduit que le typage de e_1 est nécessairement de la forme $\Gamma \cdot \Sigma \vdash e_1 : \tau_1 \cdot \bot \cdot \epsilon_1$, c'est-à-dire que e_1 est aussi du code réel. On peut donc appliquer l'hypothèse de récurrence à e_1, ce qui donne une séquence finie de pas de réductions en GhostML :

$$\mu \cdot e_1 \longrightarrow^{\geq 1} \mu' \cdot e'_1 \ \wedge\ \mathcal{E}_\bot(\mu' \cdot e'_1) = \sigma \cdot q.$$

En appliquant la règle (E-Ctx-Let), on obtient donc

$$\mu \cdot \texttt{let}\ x^\bot = e_1\ \texttt{in}\ e_2 \longrightarrow \mu' \cdot \texttt{let}\ x^\bot = e'_1\ \texttt{in}\ e_2,$$

ce qui permet de conclure, car

$$\mathcal{E}_\bot(\mu \cdot \texttt{let}\ x^\bot = e'_1\ \texttt{in}\ e_2) = \sigma \cdot (\texttt{let}\ x = \mathcal{E}_\bot(e'_1)\ \texttt{in}\ \mathcal{E}_\bot(e_2)).$$

Le cas (E-ζ) où $\mathcal{E}_\bot(e)$ est de la forme $\texttt{let}\ x = ()\ \texttt{in}\ \mathcal{E}_\bot(e_2)$ se traite de la même manière, en distinguant selon les sous-cas où x est l'effacement d'un lieur fantôme ou d'un lieur réel. Le cas (E-Ctx-App) où $\mathcal{E}_\bot(e)$ est une application MiniML de la forme $\mathcal{E}_\bot(e_1)\ \mathcal{E}_{\beta_1}(a_2)$ avec β_1 le statut fantôme du premier paramètre formel de e_1 se fait en appliquant directement l'hypothèse de récurrence à la réduction de $\mathcal{E}_\bot(e_1)$. Dans tous les autres cas, la réduction $\mathcal{E}_\bot(\mu \cdot e) \longrightarrow_m \sigma \cdot q$ en MiniML est une réduction en tête et par les raisonnements similaires aux cas précédents on montre qu'elle correspond exactement à la réduction en tête de la même forme en GhostML. □

Enfin, montrons la correction de l'effacement lorsque l'évaluation de l'expression GhostML diverge :

Théorème 2.3.3 (Préservation de l'évaluation divergente)**.** Soit une configuration $\mu \cdot e$ telle que $\Gamma \cdot \Sigma \vdash e : \tau \cdot \bot \cdot \epsilon$ et $\Sigma \models \mu$. Si $\mu \cdot e \longrightarrow \infty$, alors $\mathcal{E}_\bot(\mu \cdot e)$ diverge, c'est-à-dire que $\mathcal{E}_\bot(\mu \cdot e) \longrightarrow_m \infty$.

Démonstration. On prouve le résultat par co-induction : il suffit de montrer qu'il existe une configuration $(\mu' \cdot e')$ telle que l'on ait d'une part $\Gamma \cdot \Sigma \vdash e' : \tau \cdot \bot \cdot \epsilon'$ et $\Sigma \models \mu'$ et, d'autre part,

$$\mathcal{E}_\bot(\mu \cdot e) \to^1_m \mathcal{E}_\bot(\mu' \cdot e') \ \wedge\ \mu' \cdot e' \longrightarrow \infty.$$

Puisque l'expression e diverge, elle n'est donc pas une valeur. Par conséquent, $\mathcal{E}_\bot(e)$ n'est pas une valeur non plus. D'après le théorème 2.3.1 de préservation du typage par l'effacement, on a alors $\mathcal{E}(\Gamma) \cdot \mathcal{E}(\Sigma) \vdash_m \mathcal{E}_\bot(e) : \mathcal{E}_\bot(\tau) \cdot \epsilon$. Par ailleurs, $\Sigma \models \mu$ entraîne

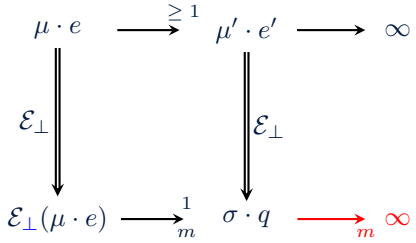
$\mathcal{E}(\Sigma) \models_m \mathcal{E}(\mu)$. En vertu de la propriété du progrès dans MiniML on sait qu'il existe une configuration $(\sigma \cdot q)$ telle que

$$\mathcal{E}_\bot(\mu \cdot e) \longrightarrow_m \sigma \cdot q.$$

D'après le lemme de simulation d'un pas de réduction de MiniML en GhostML, on obtient alors

$$\mu \cdot e \longrightarrow^{\geq 1} \mu' \cdot e' \ \wedge \ \mathcal{E}_\bot(\mu' \cdot e') = \sigma \cdot q.$$

Puisque la relation d'évaluation est déterministe, e' diverge. Cela permet d'obtenir le résultat illustré par le diagramme suivant :



En effet, par le lemme de préservation du typage on a $\Gamma \cdot \Sigma \vdash e' : \tau \cdot \bot \cdot \epsilon'$ (avec $\epsilon' \leq \epsilon$) et $\Sigma \models \mu'$. Cela permet de conclure, puisque par hypothèse de co-induction, on a $\sigma \cdot q \longrightarrow_m \infty$ et donc $\mathcal{E}_\bot(\mu \cdot e) \longrightarrow_m \infty$. □

2.4 Implémentation

L'approche que nous avons présentée décrit d'une manière rigoureuse comment, à l'aide d'un système de types, le code fantôme peut être utilisé d'une manière sûre, concise et en même temps expressive. Bien que le langage de notre formalisme n'est qu'une preuve de concept, les idées qu'il véhicule et son système de types sont à la base de ce qui est réalisé dans l'outil Why3. Bien entendu, Why3 présente de nombreuses fonctionnalités à l'égard du code fantôme que nous n'avons pas abordées dans GhostML. Le but de cette section est de passer en revue certaines de ces fonctionnalités.

Polymorphisme de types. En Why3 le typage se fait grâce à un système de types polymorphes (avec une inférence de types similaire à Damas-Milner [2]). Tant qu'une variable de type ne peut être instanciée que par un type scalaire, étendre notre formalisme avec le polymorphisme de types ne présente pas de difficultés majeures, puisque le statut fantôme y est une caractéristique propre aux variables et aux expressions, et non pas aux types eux-mêmes. Ainsi, on peut écrire `ghost 42` mais on ne peut pas écrire « `ghost Int` ».

Structures de données avec des champs fantômes. En plus du polymorphisme de types, Why3 offre la possibilité de définir des types algébriques et des enregistrements dans lesquels certains champs, qu'ils soient mutables ou pas, peuvent être fantômes. Par ailleurs, les types de données définis par l'utilisateur peuvent apparaître dans le code fantôme et dans le code réel. Ainsi, Why3 a un seul et unique type des tableaux et l'on peut manipuler aussi bien des tableaux fantômes que des tableaux réels. Notons que lorsque l'on effectue le filtrage sur les valeurs de types algébriques, cela requiert quelques précautions : lorsque l'expression analysée contient une composante fantôme, on doit vérifier que dans chacun des motifs, toutes les variables introduites à gauche qui sont en rapport avec la composante fantôme doivent être traitées à droite comme des variables fantômes. De plus, à l'instar de la manière dont le code fantôme est propagé dans les conditionnelles où la valeur de test est fantôme, si l'expression filtrée est fantôme, alors que le filtrage contient au moins deux motifs différents, on doit propager le code fantôme jusqu'à ce que la construction du filtrage devienne fantôme dans sa globalité.

Exceptions. En Why3, les exceptions sont toujours introduites globalement. Il serait donc possible d'étendre GhostML avec un mécanisme d'exceptions similaire. En effet, il suffirait pour cela d'ajouter un nouvel indicateur fantôme qui tiendrait compte de l'ensemble des exceptions qu'une expression est susceptible de lever. On pourrait alors réutiliser les mêmes exceptions dans le code fantôme et le code réel, à condition que les exceptions levées par une expression fantôme n'échappent pas au contexte fantôme *in fine*, rattrapées quelque part « plus haut » par une construction `try-with` fantôme.

Une terminaison prouvable. À l'instar de GhostML, Why3 autorise l'utilisation des fonctions récursives dans le code fantôme. Par ailleurs, les boucles fantômes sont également autorisées. Néanmoins, pour garantir la non-interférence, Why3 exige dans ces cas-là que l'on accompagne les définitions récursives et les boucles avec une clause de « variant » explicite à partir de laquelle Why3 engendre les conditions nécessaires à la preuve de terminaison. Notons que la preuve de ces conditions (par des solveurs SMT par exemple) correspond exactement à ce que nous avons modélisé dans notre formalisme par l'oracle $\texttt{CheckTermination}(\cdot)$ qui renvoie $\bot$ lorsque la terminaison peut être statiquement établie.

Références locales. En Why3 les références mutables peuvent être créées localement, passées en argument à une fonction ou en être le résultat. L'absence des références locales limite sans doute l'expressivité de notre formalisme. Néanmoins, tant que l'on exclut la possibilité d'aliasing entre deux telles références, il serait possible d'étendre GhostML avec des références locales. Dans ce cas-là, plutôt que d'être une valeur booléenne, ϵ correspondrait alors à l'ensemble des références réelles modifiées qui sont libres dans e. En particulier, une expression qui ne modifie que des références réelles définies localement serait considérée comme pure (pourvu qu'elle termine)[4]. Néanmoins, même lorsque les références mutables peuvent être locales, l'absence d'aliasing reste une restriction assez forte : rien que passer une référence en argument d'une fonction peut éventuellement créer un alias. En Why3, où l'aliasing entre les références est possible sous certaines conditions, les alias sont détectés statiquement par un système de types appelées les *régions* qui sera le sujet central du chapitre suivant,

2.4.1 Structure des files mutables avec un champs fantôme

Nous avons déjà vu dans l'introduction un petit exemple de l'utilisation du code fantôme en Why3 (le programme calculant les nombres Fibonacci). En guise de conclusion, donnons-en un autre, en illustrant quelques-unes des fonctionnalités de Why3 plus évoluées dont on vient de discuter. Considérons l'implémentation des *files mutables* suivante dont le code complet est donné dans la figure 2.6. Ici, les files mutables sont réalisées par le type d'enregistrement `queue` suivant

```
type queue = {
          mutable front: list;
          mutable rear:  list;
    ghost mutable view:  list; }
```

Les champs `front` et `rear`, contenant chacun une liste simplement chaînée immuable, sont destinés à implémenter les opérations de base `push` et `pop` d'une manière amortie (à la Baker). Le troisième champs `view` a pour but de donner le modèle logique (sous forme d'une liste immuable) de la file en question, ce qui explique son statut fantôme.

4. Nous avons représenté par ϵ à la fois les effets de terminaison et d'écriture mais on aurait pu séparer les deux. Avec les références locales, cela serait même inévitable.

Premièrement, en ce qui concerne le partage, on voit que, d'une part, le même type algébrique `list` défini par l'utilisateur est utilisé à la fois dans le code fantôme et le code réel et, d'autre part, dans la fonction `pop` (lines 34–52), la fonction réelle `rev_append` est appelée à la fois dans le code réel code (ligne 42) et le code fantôme (line 39).

Deuxièmement, sur l'exemple de la fonction `push` (lignes 27–30), où une variable locale `v` sert à recalculer le nouveau modèle logique pour ensuite mettre à jour le contenu du champ `q.view`, on peut voir comment fonctionne la propagation du code fantôme : malgré le fait que la variable `v` n'est pas déclarée comme fantôme, et le fait que la fonction `append` est aussi une fonction réelle, Why3 infère que `v` est une variable fantôme. En effet, la valeur `q.view`, qui est nécessairement fantôme, contamine le résultat du calcul de `append`, ce qui à son tour contamine l'expression `let-in` toute entière. La variable `v` devient donc fantôme. En particulier, Why3 renverrait une erreur si l'on essayait par exemple de lier `v` à un champ réel de la file `q` (ou d'une autre file). D'autre part, puisque l'expression `append q.view (Cons x Nil)` est fantôme, elle ne doit donc pas diverger en vertu du principe de la non-interférence. C'est pourquoi Why3 exige la preuve de la terminaison de la fonction `append`, ce qui est garanti ici par la présence de `variant` (ligne 8).

La galerie en ligne de programmes vérifiés avec Why3 contient plusieurs autres exemples d'utilisation de code fantôme[5].

5. http://toccata.lri.fr/gallery/ghost.en.html

```
module Queue

  type elt

  type list = Nil | Cons elt list

  let rec append (l1 l2: list) : list
    variant { l1 }
  = match l1 with
    | Nil → l2
    | Cons x r1 → Cons x (append r1 l2)
    end

  let rec rev_append (l1 l2: list) : list
    variant { l1 }
  = match l1 with
    | Nil → l2
    | Cons x r1 → rev_append r1 (Cons x l2)
    end

  type queue = {
          mutable front: list;
          mutable rear:  list;
    ghost mutable view:  list; }

  let push (x: elt) (q: queue) : unit
  = q.rear ← Cons x q.rear;
    let v = append q.view (Cons x Nil) in
    q.view ← v

  exception Empty

  let pop (q: queue): elt
    raises { Empty }
  = match q.front with
    | Cons x f →
        q.front ← f;
        q.view  ← append f (rev_append q.rear Nil);
        x
    | Nil →
        match rev_append q.rear Nil with
        | Nil      →  raise Empty
        | Cons x f →
            q.front ← f;
            q.rear  ← Nil;
            q.view  ← f;
            x
        end
    end
end
```

FIGURE 2.6 – Réalisation des files mutables en Why3.

2.5 Travaux connexes

La notion du code fantôme est une idée aussi vieille que la vérification déductive elle-même (fin des années soixante) : à l'origine, les variables dites « auxiliaires », la forme la plus rudimentaire du code fantôme, ont été utilisées dans le cadre de la vérification des programmes concurrents. D'après Jones [7] et Reynolds [17], les variables auxiliaires ont été utilisées pour la première fois par Lucas en 1968 [11]. Depuis lors, de nombreux auteurs ont adapté et considérablement élargi l'usage de code fantôme dans divers domaines de la vérification. Il est intéressant de noter que certains auteurs, notamment Reynolds [17] et Kleymann [8], font une distinction entre les variables auxiliaires « non-opérationnelles » qui sont utilisées uniquement au sein des annotations (typiquement dans les invariants de boucle) et celles qui peuvent apparaître aussi bien dans les annotations que dans le programme lui-même (comme dans notre exemple introductif du programme `fibo`). C'est donc à partir de ce dernier usage que le code fantôme a progressivement évolué vers sa forme moderne d'un code arbitraire qui, à condition qu'il n'interfère pas avec le code réel, peut partager avec lui les mêmes constructions et les mêmes idiomes du langage.

Ainsi, Zhang *et al.* [21] étudient l'utilisation du code fantôme dans le contexte de la vérification de programmes concurrents, en formalisant leurs idées sur un langage impératif simple WHILE étendu avec le parallélisme et le code fantôme. Dans leur formalisation, la non-interférence est assurée syntaxiquement, par une syntaxe stratifiée du langage : le code fantôme peut par exemple se trouver à l'intérieur d'une boucle, mais une boucle ne peut pas apparaître à l'intérieur du code fantôme ce qui, entre autres, garantit purement syntaxiquement la terminaison du code fantôme mais limite son utilisation). De façon similaire à notre approche, les auteurs définissent l'opération d'effacement et expriment la non-interférence comme la conséquence de leur preuve que les comportements de la partie réelle d'un programme (et donc de son effacement) sont inclus dans l'ensemble de ces comportements.

Schmaltz [18] propose une description rigoureuse du code fantôme pour un fragment assez conséquent du langage C avec du parallélisme, dans le cadre de la vérification des programmes `C` avec l'outil VCC [1]. Dans cet outil, l'utilisateur peut introduire des variables fantômes, définir des types de données fantômes, déclarer des champs fantômes dans les structures réelles et inclure des paramètres formels fantômes dans les fonctions réelles. En particulier, le code fantôme joue un rôle important dans la gestion du mécanisme d'*ownership*. Une différence importante par rapport à notre travail est que VCC n'effectue, à notre connaissance, aucune sorte d'inférence du code fantôme. Une autre différence est que VCC *admet* que le code fantôme termine toujours. L'utilisation des constructions du langage telles que `ghost(goto l)` rend pourtant la vérification de la terminaison du code fantôme non triviale.

Un autre exemple d'outil de vérification utilisant le code fantôme est le vérificateur de programmes `Dafny` [9]. Dans `Dafny`, comme on peut le lire dans un article de R.

Leino [10], « le concept des déclarations fantômes versus non-fantômes fait partie des bases mêmes du langage : chaque fonction, méthode, variable et paramètre peuvent être déclarés soit fantôme, soit non-fantôme » [6]. Par ailleurs, une classe peut contenir à la fois des attributs fantômes et réels mais, alors qu'il peut modifier le contenu des champs fantômes, le code fantôme ne peut pas allouer de la mémoire ou modifier le contenu des champs réels. Par conséquent, on ne peut pas utiliser dans le code fantôme le code provenant des bibliothèques où l'on fait des allocations, modifie des classes ou des tableaux. Notons, néanmoins, que sur le fragment du langage de `Dafny` correspondant à notre formalisme, la sémantique du code fantôme semble être très similaire à la sémantique que nous avons présentée dans ce travail et, à l'instar de Why3, `Dafny` exige et effectue la vérification de la terminaison du code fantôme.

La non-interférence du code fantôme peut être considérée comme un cas particulier d'analyse du flot d'information [3]. En effet, il est possible de voir le code fantôme comme un canal d'information à sécurité élevée et le code réel en tant qu'un canal d'information à sécurité basse : la non-interférence s'interprète alors comme l'absence de fuites d'information depuis le premier canal vers le second pendant l'exécution. Il est intéressant de noter, d'une part, que diverses propriétés sur les flots d'information peuvent être vérifiées à l'aide d'un système de types [15] et d'autre part, que les méthodes de preuve dans ce domaine font systématiquement l'usage de la technique de bisimulation (mais pas forcément via l'effacement). Notons cependant que cela ne veut pas dire nécessairement que l'on puisse utiliser directement les systèmes de types existants dans la littérature de flots d'information pour garantir la non-interférence du code fantôme. En effet, la terminaison du code fantôme est une condition *sine qua non* pour que la vérification des programmes avec du code fantôme ait un sens, alors que, pour citer un exemple parmi d'autres, le système de types présenté par Simonet et Pottier [16] présuppose la terminaison du « code secret ».

En ce qui concerne l'opération d'effacement, elle est très proche du mécanisme d'extraction que l'on trouve dans l'assistant de preuve Coq [12]. En effet, le but de l'extraction de Coq est d'effacer du programme tout ce qui n'est pas pertinent du point de vue opérationnel. Par exemple, dans une preuve de la forme $\exists x.\, P(x)$, lorsqu'elle est constructive, c'est-à-dire lorsque l'on a exhibé un programme t en tant que témoin de la preuve de $P(t)$, le terme t sera le seul à être préservé par l'extraction. Tout ce qui relève de la preuve elle-même étant effacé. Remarquons qu'en Coq ce mécanisme est basé sur la dualité entre la sorte `Set` pour typer les objets de nature *calculatoire* et la sorte `Prop` pour typer les preuves. En Coq la non-interférence est donc également garantie aussi par un système de typage. Néanmoins, lorsque l'utilisateur introduit un nouveau type de données, il doit choisir une fois pour toute entre `Set` et `Prop`. Ainsi, le type `nat` des entiers naturels possède la sorte `Set` et donc les entiers naturels ne sont jamais effacés pendant l'extraction, même lorsqu'ils sont utilisés uniquement pour les besoins de la spécification ou de la preuve. Contrairement à Coq, l'approche que nous

6. "The concept of ghost versus non-ghost declarations is an integral part of the Dafny language : each function, method, variable, and parameter can be declared as either ghost or non-ghost."

avons présentée permet d'utiliser le même type d'entiers naturels aussi bien dans le code fantôme que dans le code réel, et seuls les entiers utilisés dans la partie fantôme du programme y seront effacés.

Bibliographie

[1] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies, *VCC : A practical system for verifying concurrent C*, Theorem Proving in Higher Order Logics (TPHOLs), Lecture Notes in Computer Science, vol. 5674, Springer, 2009.

[2] Luis Damas and Robin Milner, *Principal type-schemes for functional programs*, POPL '82 : Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), ACM Press, 1982, pp. 207–212.

[3] D. E. Denning and P. J. Denning, *Certification of programs for secure information flow*, Communications of the ACM **20** (1977), no. 2, 504–513.

[4] Jean-Christophe Filliâtre and Andrei Paskevich, *Why3 — where programs meet provers*, Proceedings of the 22nd European Symposium on Programming (Matthias Felleisen and Philippa Gardner, eds.), Lecture Notes in Computer Science, vol. 7792, Springer, March 2013, pp. 125–128.

[5] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen, *The essence of compiling with continuations*, SIGPLAN Not. **28** (1993), no. 6, 237–247.

[6] Bart Jacobs and Frank Piessens, *The VeriFast program verifier*, CW Reports CW520, Department of Computer Science, K.U.Leuven, August 2008.

[7] Cliff B. Jones, A.W. Roscoe, and Kenneth R. Wood, *Reflections on the work of C.A.R. Hoare*, 1st ed., Springer Publishing Company, Incorporated, 2010.

[8] Thomas Kleymann, *Hoare logic and auxiliary variables.*, Formal Asp. Comput. **11** (1999), no. 5, 541–566.

[9] K. Rustan M. Leino, *Dafny : An automatic program verifier for functional correctness*, LPAR-16, Lecture Notes in Computer Science, vol. 6355, Springer, 2010, pp. 348–370.

[10] K. Rustan M. Leino and Michał Moskal, *Co-induction simply : Automatic coinductive proofs in a program verifier*, FM 2014 : Formal Methods (Cliff Jones, Pekka Pihlajasaari, and Jun Sun, eds.), Lecture Notes in Computer Science, vol. 8442, Springer, May 2014, pp. 382–398.

[11] P. Lucas, *Two constructive realizations of the block concept and their equivalence*, Technical Report 25.085, IBM Laboratory, Vienna, June 1968.

[12] Christine Paulin-Mohring, *Extracting F_ω's programs from proofs in the Calculus of Constructions*, Sixteenth Annual ACM Symposium on Principles of Programming Languages (Austin), ACM Press, January 1989.

[13] Benjamin C. Pierce, *Types and programming languages*, MIT Press, 2002.

[14] Gordon D. Plotkin, *Call-by-name, call-by-value and the lambda-calculus.*, Theor. Comput. Sci. **1** (1975), no. 2, 125–159.

[15] François Pottier and Sylvain Conchon, *Information flow inference for free*, Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00) (Montréal, Canada), September 2000, pp. 46–57.

[16] François Pottier and Vincent Simonet, *Information flow inference for ML*, ACM Transactions on Programming Languages and Systems **25** (2003), no. 1, 117–158, ©ACM.

[17] John C. Reynolds, *The craft of programming*, Prentice Hall International series in computer science, Prentice Hall, 1981.

[18] Sabine Schmaltz, *Towards the pervasive formal verification of multi-core operating systems and hypervisors implemented in c*, Ph.D. thesis, Saarland University, Saarbrücken, 2013.

[19] William W. Tait, *Intensional interpretations of functionals of finite type I*, J. Symb. Log. **32** (1967), no. 2, 198–212.

[20] Andrew K. Wright and Matthias Felleisen, *A syntactic approach to type soundness*, Information and Computation **115** (1992), 38–94.

[21] Zipeng Zhang, Xinyu Feng, Ming Fu, Zhong Shao, and Yong Li, *A structural approach to prophecy variables*, 9th annual conference on Theory and Applications of Models of Computation (TAMC) (Manindra Agrawal, S.Barry Cooper, and Angsheng Li, eds.), Lecture Notes in Computer Science, vol. 7287, Springer Berlin Heidelberg, 2012, pp. 61–71.

3 Régions

Dans ce chapitre, nous nous intéressons à la vérification des programmes manipulant des pointeurs. En présence d'*aliasing*, c'est-à-dire lorsque plusieurs pointeurs manipulés dans un programme dénotent une même case mémoire, la spécification et la vérification deviennent non triviales.

De nombreuses approches formelles, comme le *modèle mémoire explicite* [6], la *logique de séparation* [26] ou encore les *dynamic frames* [19] ont été proposées au cours des dernières décennies pour affronter la complexité du raisonnement sur les programmes manipulant des pointeurs. Si toutes ces approches ont vu le jour, c'est que même pour les cas de figure plus élémentaires (sans parler des cas aussi complexes que les listes chaînées, les arbres mutables, etc.), lorsque le programme comporte un alias, on sort du cadre théorique de la logique de Hoare [17]. En effet, la validité et, pour ainsi dire, la simplicité naturelle de la logique de Hoare gît dans le fait que la règle d'affectation

$$\{\, P[x \leftarrow E] \,\}\ x := E\ \{\, P \,\}$$

présuppose que l'adresse mémoire à laquelle x se réfère n'a pas d'autres noms symboliques que la variable x. Or, en présence d'alias, on doit abandonner cette hypothèse. Lorsque les alias qu'un programme manipule sont arbitraires, cela ne fait pas de doute que l'on doit chercher un cadre logique plus complexe.

Cependant, on peut observer qu'en pratique la grande majorité du code que l'on souhaite vérifier, bien qu'il puisse présenter certains cas de figure d'aliasing élémentaires, rentre néanmoins dans le cadre de la logique de Hoare. Pourquoi en est- il ainsi ? Le secret est la modularité et l'abstraction. Prenons l'exemple fréquent d'une structure de données permettant de manipuler un ensemble mutable. La réalisation d'une telle structure peut bien entendu faire usage de pointeurs arbitraires (typiquement, des arbres mutables ou des tables de hachage avec chaînage). Néanmoins, le code client qui utilise la structure en question peut faire abstraction de la manière dont elle est réalisée. En effet, sans connaître la complexité de la réalisation, le code client peut manipuler l'ensemble mutable à travers des fonctions abstraites comme s'il s'agissait d'une simple variable mutable, dans le sens de la logique de Hoare. Par conséquent, on peut espérer vérifier au moins certains fragments conséquents du code en utilisant des méthodes de preuve simples *à la* logique de Hoare. Mais quelle est la complexité de la relation d'aliasing dans de tels fragments ? Il n'est certainement pas réaliste de nous limiter à des programmes dans lesquels il n'y a jamais d'alias. Le but de ce chapitre

```
1  hashTable = struct {
2    data : array of elt;
3    size : int;
4    ghost model : set of elt;
5  }
6
7  function createTable()
8    var h = new hashTable
9    h.data ← createArray(16)
10   h.size ← 0
11   h.model ← ∅
12   return h
13
14 function clear(h)
15   ... remove all elements ...
16   h.size ← 0
17   h.model ← ∅
18 function contains(h , x)
19   ... search for x in h.data ...
20
21 function resize(h)
22   var n = 2 × length(h.data)
23   var b = createArray(n)
24   ... move elements from h.data to a ...
25   h.data ← b
26
27 function add(h , x)
28   if contains(h , x) then
29     return
30   if isFull(h) then begin
31     resize(h)
32   ... insert x in h.data ...
33   h.size ← h.size + 1
34   h.model ← h.model ∪ {x}
```

FIGURE 3.1 – A hash table implementation.

est de montrer qu'il existe un juste milieu où les alias, tant qu'ils sont connus statiquement, sont autorisés. L'idée clé est que, contrairement aux approches mentionnées ci-dessus dans lesquelles les conditions de séparation apparaissent explicitement dans les obligations de preuve, notre approche est d'opérer un contrôle statique des alias au préalable, avant même de générer les obligations de preuve. En effet, si tous les alias de la variable x sont connus statiquement, nous pouvons adapter la règle de Hoare pour l'affectation, réconciliant une partie de la complexité rencontrée en pratique avec la simplicité de la logique de Hoare. Concrètement, nous présentons dans ce chapitre une méthode de contrôle statique des alias pour un petit langage de programmation que nous appelons RegML et qui permet de manipuler des structures de données avec des composantes mutables imbriquées à une profondeur bornée. Notre approche s'appuie sur un système de types avec effets et des types singletons que nous appelons *régions*, l'idée étant d'encoder l'identité unique de chaque valeur mutable non pas dans son nom symbolique mais dans sa région dont elle est l'unique habitant. Notons que, dans la pratique, les effets et les régions peuvent être inférés automatiquement, en cachant ainsi une partie de la complexité des règles de typage à l'utilisateur. C'est d'ailleurs ce qui se passe dans la réalisation du système de typage dans l'outil Why3, qui se base sur les mêmes idées que le système de types que nous présentons dans ce chapitre, où les annotations de types écrites par l'utilisateur ne mentionnent jamais les régions.

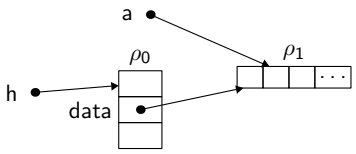
Avant de présenter le formalisme lui-même, illustrons notre approche d'une manière informelle sur l'exemple d'une implémentation des tables de hachage donnée dans la

figure 3.1. Ici, une table de hachage est représentée (lignes 1–5) par une structure de données avec trois champs mutables : un tableau *data* qui détient les entrées de la table de hachage (on suppose une implémentation à base d'un adressage ouvert), un entier *size* représente le nombre total des éléments dans la table et un champ fantôme *model* qui représente un modèle logique de la table de hachage en tant qu'un ensemble d'éléments. Expliquons brièvement le code de chaque fonction. La fonction CREATETABLE (lignes 7–12) renvoie une nouvelle table fraîche qui est vide. La fonction CLEAR (lignes 14–17) vide le contenu de la table h. La fonction CONTAINS (lignes 18–19) renvoie le booléen indiquant que la table h contient l'élément x. La fonction RESIZE (lignes 21–25) multiplie par deux la capacité de stockage de la table h, en remplaçant le tableau $h.data$ par un nouveau tableau plus grand où tous les éléments de la table h sont recopiés hachés vis-à-vis de ce nouveau tableau. La fonction ADD (lignes 27–34) ajoute l'élément x à la table h lorsqu'il n'y est pas encore présent. Enfin, notons que le type exact des éléments qui n'est pas pertinent ici est laissé non spécifié. De même, nous avons remplacé certaines parties de code par des commentaires lorsqu'elles ne sont pas pertinentes pour notre propos.

Commençons par voir ce qui se passe lorsque l'on assigne le tableau $h.data$ à une nouvelle variable a :

var $a = h.data$.

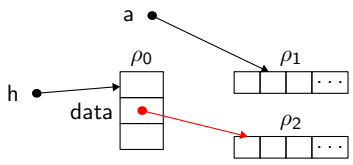
Dans ce cas-là, notre système de types va lui attribuer la même région ρ_1 que celle du champ $h.data$ de la table h, reflétant le fait que les deux noms symboliques représentent le même tableau : De cette manière-là, un générateur d'obligations de preuve basé sur



la logique de Hoare aura l'information nécessaire pour mettre à jour les valeurs de h et de a correctement lorsque l'un des deux noms symboliques sera modifié, à condition que la modification préserve la relation d'aliasing entre la table h et la variable a. Voyons donc ce qui se passe lorsqu'une modification *rompt* l'aliasing entre h et a. Affectons par exemple au champ $h.data$ un nouveau tableau :

$h.data \leftarrow$ CREATEARRAY(10)

Dans ce cas, une solution possible serait de changer le type de $h.data$ et *a fortiori* le type de h :



Une telle approche est connue dans la littérature sous le nom de *strong update* [4]. Or, cette approche requiert l'utilisation de types dépendants, dès lors que la relation d'aliasing est sujette à des modifications sous une conditionnelle. Pour s'en persuader, considérons ce qui se passe lorsque la déclaration

var $a = h.data$

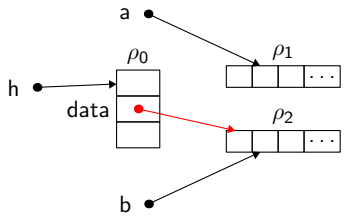
est suivie par un appel à la fonction ADD(h,x) sur un élément x que la table h ne contient pas. On arrive dans ce cas-là au code suivant (lignes 30–31 dans la figure 3.1) :

if ISFULL(h) **then**
  RESIZE(h)

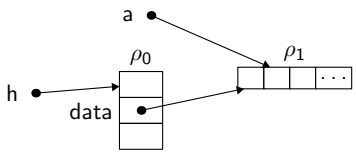
Si le booléen ISFULL(h) est vrai, alors la fonction RESIZE va être appelée sur h et on aura alors le fragment du code suivant (lignes 21–25) :

var $n = 2 \times$ LENGTH($h.data$)
var $b =$ CREATEARRAY(n)
  ...on transfère le contenu de la table vers le tableau b...
  $h.data \leftarrow b$
end

Comme on peut le voir, le tableau $h.data$ cesse d'être l'alias de la variable a si et seulement si la condition ISFULL(h) est vraie, ce que l'on ne peut savoir statiquement,



ISFULL(h) = `true`.



ISFULL(h) = `false`.

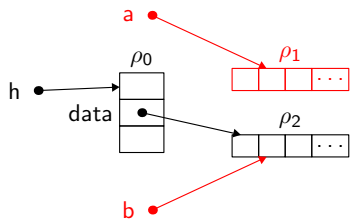
et comme le montre l'exemple ci-dessus, de telles conditionnelles peuvent même être cachées dans un appel de fonction.

Effectuer les *strong updates* nous amènerait vers l'utilisation de types dépendants. Nous optons donc pour une approche différente qui consiste à détecter, entre deux noms symboliques, les « *conflits d'aliasing* » potentiels qui surviennent lorsque les deux noms aliasés cessent de l'être ou, au contraire, lorsque les deux noms qui n'étaient pas aliasés le deviennent. En présence d'un conflit d'aliasing, notre solution est alors de bannir l'un des deux noms de la suite du programme. Par exemple, dans le fragment de code ci-dessus, l'affectation

$h.data \leftarrow b$

engendre deux conflits d'aliasing. D'une part, la variable a et le tableau $h.data$, qui étaient aliasés ne le sont plus, alors qu'ils partageaient la même région ρ_1. D'autre part,

la variable b et $h.data$ qui n'étaient pas aliasés, le deviennent alors qu'ils habitent deux régions distinctes. Pour préserver la cohérence du typage, notre approche rend donc illicite l'utilisation des variables a et b dans la suite du programme après l'affection conflictuelle :



Cependant, on a toujours le droit de mentionner le champ $h.data$. De plus, puisque l'on a éliminé tous les autres prétendants à la région ρ_1, le type de $h.data$ et *a fortiori* de la table h n'a plus besoin d'être changé.

Notons que l'on aurait pu inversement invalider le tableau $h.data$ (et *a fortiori* la table h), en préservant les variables a et b, ce qui aurait également résolu les conflits d'aliasing survenus. Cependant, nous observons que ces confits ont été engendrés par la modification de la table h ce qui nous incite à supposer que l'intention de l'utilisateur était avant tout de garder la table h. C'est donc ce que l'on choisit de faire systématiquement dans notre approche.

Techniquement, la résolution des conflits d'aliasing et l'élimination des noms conflictuels est réalisée avec un effet *d'écriture* et un *effet d'invalidation*. Ainsi, à l'affectation $h.data \leftarrow b$ ci-dessus, le système de types va associer les effets

$$(writes\ \{\rho\} \cdot reset\ \{\rho_1, \rho_2\})$$

Grâce à ces effets, il devient possible de vérifier que les variables a et b n'apparaissent plus dans la suite du code. En effet, il suffit de vérifier statiquement que n'y apparaissent plus tous noms symboliques différents de h et $h.data$ et ayant soit le type ρ_1, soit le type ρ_2 et, plus généralement, n'importe quel nom symbolique depuis lesquels les régions ρ_1 ou ρ_2 sont accessibles directement sans passer par le type ρ.

Il est intéressant de remarquer que dans notre système de types la fraîcheur d'une nouvelle région ρ' peut être exprimée uniquement en termes d'effets. Cela est dû au fait qu'associer à une expression d'allocation les effets

$$(writes\ \varnothing \cdot reset\ \{\rho'\})$$

invalide tous les noms symboliques éventuels qui habitaient antérieurement la région ρ', de sorte qu'ils ne rentrent pas en conflit d'aliasing avec le nouvel habitant de ρ'. Il devient donc sans importance si la région ρ' a été mentionnée avant ou pas et, dans ce sens, c'est une région fraîche.

Notons que notre approche ne s'applique pas à des structures de données manipulant des alias arbitraires. Comme nous traçons les identités des valeurs mutables

statiquement à travers leurs régions, un type d'une structure de données doit contenir l'information sur les régions de toutes les composantes mutables qu'elle comprend. En particulier, nous ne traitons pas les données mutables récursives telles que des listes chaînées ou des arbres. Comme nous l'avons argumenté au début de l'introduction, nous faisons l'hypothèse que de telles structures représentent une minorité du code à vérifier en pratique et que, lorsque c'est nécessaire, le développeur pourra effectuer leur vérification séparément grâce à des barrières d'abstraction, typiquement grâce à un système de modules ou d'interfaces.

Le reste du chapitre est organisé de la manière suivante. La section 3.1 introduit un petit langage RegML avec des structures de données ayant des composantes mutables imbriquées à une profondeur bornée. Nous donnons sa syntaxe et sa sémantique opérationnelle. Ensuite, la section 3.2 présente le système de types avec effets et régions dans lequel nous formalisons les idées de notre approche de contrôle statique des alias. Après avoir présenté le système du typage, nous montrons sa correction dans la section 3.3. Comme dans le chapitre précédent sur le code fantôme, le langage du formalisme est délibérément simplifié pour ne traiter que les points et les difficultés essentiels, laissant à la section 3.4 le soin de présenter les extensions et les fonctionnalités plus avancées de l'implémentation réalisée dans l'outil Why3. Nous terminons le chapitre avec la section 3.5 où nous donnons des notes bibliographiques en rapport avec notre travail.

3.1 RegML : un petit langage avec des régions

Dans cette section, nous présentons RegML, un petit langage d'expressions, paramétré par un ensemble de fonctions globales, qui permet de manipuler des structures mutables imbriquées que l'on appelle des *enregistrements*. Nous donnons d'abord la syntaxe puis la sémantique opérationnelle de RegML.

3.1.1 Syntaxe

La syntaxe abstraite des expressions de RegML est donnée dans la figure 3.2. On distingue deux sortes d'expressions : les expressions *atomiques* et les expressions *composées*. Les expressions atomiques comprennent les *variables* et les *valeurs*. Une valeur peut être soit une *constante* c (un entier, un booléen, ou une valeur unitaire), soit une *adresse mémoire* ℓ. Notons que l'on ne précise pas la représentation concrète d'une adresse mémoire ; il s'agit seulement de distinguer les adresses mémoire des autres constantes.

Les expressions composées comportent la liaison locale, la conditionnelle, l'appel d'une fonction ainsi que les opérations d'allocation, d'accès à un champ et d'affectation parallèle. Pour allouer une nouvelle adresse mémoire, on dispose d'une construction d'*enregistrement* $\{f_1 = a_1, \ldots, f_n = a_n\}$ dans laquelle on se sert des expressions $a_1, \ldots, a_n$ pour initialiser le contenu des champs $f_1, \ldots, f_n$ respectivement. L'ordre dans lequel apparaissent les initialisations des champs ne joue aucun rôle. Tous les champs sont mutables, c'est-à-dire que l'on peut modifier leur contenu. La modification est réalisée grâce à l'opération d'*affectation parallèle*

$$a_1.\{f_{1,1} \leftarrow a_{1,1}, \ldots, f_{1,k_1} \leftarrow a_{1,k_1}\}, \ldots, a_n.\{f_{n,1} \leftarrow a_{n,1}, \ldots, f_{n,k_n} \leftarrow a_{n,k_n}\}$$

qui permet de modifier simultanément le contenu de plusieurs champs de plusieurs enregistrements. Là encore, l'ordre dans lequel on écrit les affectations n'a pas d'importance. Pour alléger la notation, lorsque l'on ne modifie qu'un seul champ dans un enregistrement, on omettra les accolades, en écrivant simplement $a.f \leftarrow a'$. Par exemple, l'expression ci-dessous alloue deux nouveaux enregistrements, liés respectivement aux variables x et y, puis on échange le contenu de $x.f$ et $y.g$.

```
let x = {f = 1} in
let y = {g = 2} in
x.f ← y.g, y.g ← x.f
```

L'appel d'une fonction se note $p(a_1, \ldots, a_n)$ où p est le nom de la fonction et les atomes $a_1, \ldots, a_n$ sont les arguments. On distingue deux sortes de fonctions : les opérations *primitives* prédéfinies qui comprennent des opérations arithmétiques et des opérateurs de comparaison, et les fonctions *définies par l'utilisateur*. On note la définition d'une telle fonction par $p(x_1, \ldots, x_n) \mapsto e$ avec $n \geq 0$ et où $x_1, \ldots, x_n$ sont

e	$::=$	EXPRESSIONS
	$\mid a$	*expression atomique*
	$\mid \texttt{let}\ x = e\ \texttt{in}\ e$	*liaison locale*
	$\mid \texttt{if}\ a\ \texttt{then}\ e\ \texttt{else}\ e$	*conditionnelle*
	$\mid p(a, \ldots, a)$	*appel de fonction*
	$\mid \{f = a, \ldots, f = a\}$	*allocation*
	$\mid a.f$	*accès au champs*
	$\mid a.\{f \leftarrow a, \ldots, f \leftarrow a\}, \ldots, a.\{f \leftarrow a, \ldots, f \leftarrow a\}$	*affectation parallèle*
a	$::=$	EXPRESSIONS ATOMIQUES
	$\mid x$	*variable*
	$\mid v$	*valeur*
v	$::=$	VALEURS
	$\mid \ell$	*adresse mémoire*
	$\mid c$	*constante scalaire*
c	$::=$	CONSTANTES SCALAIRES
	$\mid \mathbb{Z}$	*entiers*
	$\mid \texttt{True}, \texttt{False}$	*booléens*
	$\mid ()$	*unit*

FIGURE 3.2 – Syntaxe abstraite des expressions de RegML.

les arguments formels et l'expression e est le corps de la fonction p. Nous supposons que toutes les fonctions de l'utilisateur sont définies dans un environnement global P fixé. On peut donc voir un programme comme une liste de définitions globales de fonctions suivie d'une expression e. Néanmoins, pour ne pas alourdir la présentation, on garde l'environnement P comme un paramètre implicite que l'on n'évoquera que lorsque cela est nécessaire, sans pour autant l'écrire systématiquement dans le reste du formalisme. Par ailleurs, pour les opérations primitives, la notation infixe $(1 + 2)$ est privilégiée dans les exemples à la notation préfixée $+(1, 2)$.

Notons que la syntaxe du langage ne permet pas d'écrire des expressions telles que $p(e_1, \ldots, e_n)$ ou $e.f$. En effet, dans les expressions composées, sauf la liaison locale et les expressions de branchement dans une conditionnelle, toutes les sous-expressions sont nécessairement atomiques. Autrement dit, nous avons mis la syntaxe des expressions en *forme A-normale* [14]. La vertu d'une telle présentation est qu'elle permet de

simplifier le nombre de cas à considérer lorsque l'on décrit le modèle d'exécution du langage sans pour autant restreindre son pouvoir expressif. Sans démontrer cette dernière affirmation, notons simplement qu'une expression telle que $p(42).f$ peut être réécrite en une expression A-normale $\texttt{let}\ x = p(42)\ \texttt{in}\ x.f$ équivalente du point de vue sémantique. Néanmoins, pour rendre la lecture des exemples plus facile, on s'autorisera à ne pas respecter la forme A-normale. Ainsi, on écrira $\{f_1 \leftarrow 42 + 13, f_2 \leftarrow \{g = \texttt{True}\}\}$ au lieu de

$$\begin{array}{l}\texttt{let}\ x_1\ \texttt{=}\ \texttt{+(42,13)}\ \texttt{in}\\ \texttt{let}\ x_2\ \texttt{=}\ \texttt{\{}\, g\ \texttt{=}\ \texttt{True}\,\texttt{\}}\ \texttt{in}\\ \texttt{\{}\, f_1\ \texttt{=}\ x_1\texttt{,}\ \ f_2\ \texttt{=}\ x_2\,\texttt{\}.}\end{array}$$

Étant donnée une expression e, nous désignons par $\mathcal{F}_v(e)$, $\mathcal{F}_p(e)$ et $\mathcal{F}_\ell(e)$ respectivement l'ensemble des variables libres, l'ensemble des noms de fonctions et l'ensemble des adresses mémoire qui apparaissent dans e. Notons bien que seules les variables peuvent être liées dans les expressions. Lorsque $\mathcal{F}_v(e)$ est vide, nous dirons que e est une *expression close*. Enfin, on utilisera parfois la notation de séquence $e_1\,;e_2$ au lieu de $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ lorsque $x \notin \mathcal{F}_v(e_2)$.

3.1.2 Sémantique

Le modèle d'exécution de RegML est donné par une sémantique opérationnelle à petits pas qui décrit comment une expression peut se réduire lors d'un pas d'évaluation. Les effets de bord correspondent soit à l'allocation d'une nouvelle adresse mémoire, soit à la modification d'une partie de la mémoire déjà allouée dans un *état mémoire* courant.

Définition 3.1.1 (État mémoire)**.** Un état mémoire, que l'on note μ, est défini comme fonction partielle qui à une adresse mémoire ℓ et un nom de champ f donnés associe une valeur $\mu(\ell.f)$. On note $\mu[\ell.f \mapsto v]$ l'état mémoire μ' tel que le domaine de $\mu' \setminus \{\ell.f\}$ soit égal au domaine de μ, $\mu'(\ell.f)$ soit égal à v et pour tout couple (ℓ', f') tel que $\ell.f \neq \ell'.f'$ on ait $\mu'(\ell.f) = \mu(\ell.f)$. Par abus de notation, on écrira $\ell \in \operatorname{dom}\mu$ au lieu de $\exists f,\ \ell.f \in \operatorname{dom}\mu$. De même, on écrira $\ell \notin \operatorname{dom}\mu$ au lieu de $\forall f, \ell.f \notin \operatorname{dom}\mu$.

On dénote un pas de réduction par $\mu \cdot e \longrightarrow \mu' \cdot e'$ où $(\mu \cdot e)$ est la configuration initiale et $(\mu' \cdot e')$ la configuration d'arrivée avec e et e' des expressions closes. La sémantique opérationnelle est définie alors par l'ensemble des règles données dans la figure 3.3 qui décrivent toutes les formes possibles qu'un pas de réduction peut avoir. Comme d'habitude, on dénote par $\longrightarrow^\star$ la clôture réflexive transitive de $\longrightarrow$. Expliquons en détails chacune des règles de la sémantique opérationnelle. Les règles E-Op et E-δ décrivent l'évaluation d'un appel de fonction qui peut être respectivement soit une opération *primitive*, soit une fonction *définie par l'utilisateur*. Tous les appels de fonctions sont totaux. Rappelons que les fonctions primitives consistent en des opérations arithmétiques et des opérateurs de comparaison. Elles ne s'appliquent qu'à

$$\frac{\text{la constante } \delta(p, c_1, \ldots, c_n) \text{ est définie}}{\mu \cdot p(c_1, \ldots, c_n) \longrightarrow \mu \cdot \delta(p, c_1, \ldots, c_n)} \text{ (E-Op)}$$

$$\frac{p(x_1, \ldots, x_n) \mapsto e \ \in P}{\mu \cdot p(v_1, \ldots, v_n) \longrightarrow \mu \cdot e[x_i/v_i]} \text{ (E-}\delta\text{)}$$

$$\frac{\ell \notin \operatorname{dom} \mu \qquad f_i \text{ deux à deux distincts}}{\mu \cdot \{f_i = v_i \ ^{i \in [1,\ldots,n]}\} \longrightarrow \mu[\ell.f_i \mapsto v_i] \cdot \ell} \text{ (E-Alloc)}$$

$$\frac{\mu(\ell.f) \text{ est définie}}{\mu \cdot \ell.f \longrightarrow \mu \cdot \mu(\ell.f)} \text{ (E-Field)}$$

$$\frac{\ell_i.f_{i,j} \in \operatorname{dom} \mu \quad \ell_i \text{ deux à deux distincts} \quad \forall i.\ f_{i,j} \text{ deux à deux distincts}}{\mu \cdot \ell_i.\{f_{i,j} \leftarrow v_{i,j} \ ^{j \in [1,\ldots,k_i]}\}^{i \in [1,\ldots,n]} \longrightarrow \mu[\ell_i.f_{i,j} \mapsto v_{i,j}] \cdot ()} \text{ (E-Assign)}$$

$$\frac{}{\mu \cdot \texttt{if True then } e_1 \texttt{ else } e_2 \longrightarrow \mu \cdot e_1} \text{ (E-True)}$$

$$\frac{}{\mu \cdot \texttt{if False then } e_1 \texttt{ else } e_2 \longrightarrow \mu \cdot e_2} \text{ (E-False)}$$

$$\frac{}{\mu \cdot \texttt{let } x = v \texttt{ in } e \longrightarrow \mu \cdot e[x/v]} \text{ (E-}\zeta\text{)}$$

$$\frac{\mu \cdot e_1 \longrightarrow \mu' \cdot e_1'}{\mu \cdot \texttt{let } x = e_1 \texttt{ in } e_2 \longrightarrow \mu' \cdot \texttt{let } x = e_1' \texttt{ in } e_2} \text{ (E-Ctx)}$$

FIGURE 3.3 – Sémantique opérationnelle de RegML.

des valeurs scalaires, renvoient toujours une constante et ne modifient pas l'état mémoire. L'appel à une fonction primitive p est évalué en utilisant un oracle sémantique δ. Dans ce cas, on note le résultat de l'évaluation $\delta(p, c_1, \ldots, c_m)$. Par exemple, on a $\delta(\leq, 0, 42) = \texttt{True}$, $\delta(+, 21, 21) = 42$, etc.

En ce qui concerne les fonctions définies par l'utilisateur, nous imposons que dans une définition de fonction $p(x_1, \ldots, x_n) \mapsto e$ on a $\mathcal{F}_v(e) \subseteq \{x_1, \ldots, x_n\}$, $\mathcal{F}_\ell(e) = \varnothing$ et $\mathcal{F}_p(e) \subseteq P$. Nous supposons de plus que toutes les fonctions dans P sont donc mutuellement récursives. Les deux premières conditions garantissent que la sémantique opérationnelle décrit toujours l'évaluation d'expressions closes. La troisième condition assure que les fonctions dans P n'utilisent que des fonctions définies dans P. La règle E-δ permet de remplacer un appel de fonction $p(v_1, \ldots, v_n)$ par sa définition $e[x_i/v_i]$ où

chaque paramètre formel est substitué par la valeur respective passée en argument. Par ailleurs, les fonctions définies dans P étant mutuellement récursives, l'évaluation d'une expression ne termine pas nécessairement. Ainsi, on peut définir p par $p(x) \mapsto p(x)$ et alors, quel que soit l'état mémoire μ, l'évaluation de la configuration $\mu \cdot p(41)$ ne terminera pas :

$$\mu \cdot p(41) \longrightarrow \mu \cdot p(41) \longrightarrow \dots$$

Les règles E-Alloc et E-Assign décrivent respectivement l'allocation d'une nouvelle adresse mémoire et l'affectation parallèle. Afin que l'évaluation ne soit pas ambiguë, nous imposons dans ces deux règles que les noms des champs qu'elles mentionnent soient deux à deux distincts. Notons cependant que rien n'interdit de partager les mêmes noms de champs dans des enregistrements différents. Par exemple, il est acceptable d'avoir $\{f = 1; g = 2\}$ et $\{f = 1; h = 3\}$. Par ailleurs, afin d'éviter d'écrire $(\mu(\dots(\mu(\mu(\ell.f_1).f_2)\dots).f_n)$, pour dénoter l'accès à une valeur depuis une adresse mémoire ℓ dans un état mémoire μ en plusieurs étapes, nous introduisons la définition d'un *chemin d'accessibilité* :

Définition 3.1.2 (Chemin d'accessibilité)**.** Un chemin d'accessibilité, ou simplement un chemin, est une séquence (éventuellement vide) de noms de champs. Les chemins sont dénotés par la lettre π. Le chemin vide est noté ϵ. Nous écrivons $\pi_1 \preceq \pi_2$ pour dire que π_1 est un préfixe (pas nécessairement propre) de π_2.

Nous pouvons alors généraliser l'accès à l'état mémoire à des chemins d'une profondeur arbitraire bornée :

$$\mu(\ell.\pi) \triangleq \begin{cases} \ell & \text{si } \pi = \epsilon \text{ et } \ell \in \operatorname{dom}\mu \\ \mu(\ell.f) & \text{si } \pi = f \text{ et } \ell \in \operatorname{dom}\mu \\ \mu(\ell'.\pi') & \text{si } \pi = f\pi' \text{ et } \mu(\ell.f) = \ell' \end{cases}$$

Nous disons alors qu'une adresse mémoire ℓ' est *accessible* depuis une adresse ℓ dans l'état mémoire μ lorsqu'il existe un chemin π tel que $\mu(\ell.\pi) = \ell'$. Étant donné un ensemble d'adresses L, nous désignons par $\mathcal{A}_\ell(\mu \cdot L)$ l'ensemble des adresses accessibles dans μ depuis les adresses de L. Par abus de notation, on écrira $\mathcal{A}_\ell(\mu \cdot e)$ au lieu de $\mathcal{A}_\ell(\mu \cdot \mathcal{F}_\ell(e))$.

Considérons maintenant le cas où l'expression e est de la forme $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$. Si le sous-terme e_1 est une valeur v, alors l'évaluation consiste simplement à substituer dans e_2 toutes les occurrences libres de x par v (la règle E-ζ). Sinon, lorsque e_1 est une expression composée, d'après la règle E-Ctx, on a récursivement le pas de réduction $\mu \cdot e_1 \longrightarrow \mu' \cdot e_1'$, ce qui permet de passer de la configuration $\mu \cdot \texttt{let}\ x = e_1\ \texttt{in}\ e_2$ à la configuration $\mu' \cdot \texttt{let}\ x = e_1'\ \texttt{in}\ e_2$. Notons que les expressions de notre langage étant en forme A-normale, la règle E-Ctx est la seule règle contextuelle. Enfin, notons que nous avons écrit « une *fonction* de transition » et non pas une « *relation* de transition » en soulignant que quelle que soit la configuration considérée, il y aura toujours au plus une règle de sémantique s'applique. Autrement dit, la sémantique opérationnelle de notre langage est déterministe.

3.2 Système de types avec effets de RegML

Le but d'un système de typage est de garantir que l'évaluation des expressions bien typées soit se termine par une valeur, soit diverge. Mais les objectifs du système de types que nous présentons dans ce chapitre ne se limitent pas à la sûreté. Notre but est également de concevoir un système de types suffisamment expressif pour que, dans les expressions bien typées, on soit capable de traquer les alias statiquement, c'est-à-dire capable de déterminer l'ensemble des noms symboliques (variables, adresses mémoire, accès via un champs) qui dénotent la même adresse mémoire. Notons dès à présent qu'il y aura des expressions qui, bien que sûres dans le sens de Milner, seront rejetées par notre système de types. Prenons l'exemple suivant

```
let x = {f = {g = 42}} in
let y = {g = 0} in
if p(43) then x.f ← y else ();
x.f + y.g
```

où l'on ne connaît pas statiquement laquelle des deux branches de la conditionnelle sera évaluée. Par conséquent, à moins que l'on introduise un système du typage avec des types dépendants, on ne saurait pas déterminer statiquement si les noms symboliques $x.f$ et y sont aliasés ou pas dans l'expression $x.f + y.g$. Bien que certains programmes de ce genre ne seront pas typables dans notre système, cela ne signifie pour autant que tout programme modifiant la structure de l'état mémoire sera rejeté. Ainsi, si l'on change la dernière ligne dans l'exemple précédent en interdisant l'utilisation de la variable y, le programme sera accepté dans notre système des types. En effet, bien que l'on ait toujours un conflit d'aliasing entre $x.f$ et y, ce conflit n'est plus observable. D'une certaine manière, on a résolu le conflit en ne gardant que x dans la suite du programme après l'endroit où le conflit a eu lieu.

Pour détecter et résoudre les conflits d'aliasing, nous allons caractériser une expression non seulement par un type mais également par un *effet*, une sorte de « service de renseignements », qui donne les informations sur les possibles modifications de tout état mémoire dans lequel l'expression sera évaluée. Concrètement, les *jugements de typage* sont de la forme

$$\Gamma \cdot \Sigma \vdash e : \tau \cdot \varepsilon$$

où Γ et Σ sont respectivement des environnements de typage des variables et des valeurs, e est une expression, τ est un type et ε est un effet. Avant d'expliquer les règles qui définissent les jugements dérivables notre système de types, commençons par définir formellement ce que sont les types, les environnements du typage et les effets dans notre système de types.

τ	::=	TYPES
	$\mid \nu$	*type scalaire*
	$\mid \rho$	*région*
ν	::=	TYPES SCALAIRES
	$\mid$ `Int` $\mid$ `Bool` $\mid$ `Unit`	*types scalaires*
ρ	::=	RÉGIONS
	$\mid \{f : \tau, \ldots, f : \tau\}_r$	*type enregistrement*

FIGURE 3.4 – Types et régions.

3.2.1 Types

La grammaire des types est donnée dans la figure 3.4. Les entiers, les booléens et la valeur unitaire (unit) sont caractérisés par des *types scalaires*. Ces types-là sont *primitifs*, c'est-à-dire qu'ils n'ont pas de structure. Les adresses mémoire, quant à elles, sont caractérisées par des types de données structurés que nous appelons *régions*. Une région $\{f_1 : \tau_1, \ldots, f_n : \tau_n\}_r$ est constituée d'un ensemble de champs $f_1, \ldots, f_n$, chaque champs f_i ayant respectivement le type τ_i. Par ailleurs, chaque région est dotée d'un identificateur r qui sert à distinguer des régions caractérisant des adresses mémoire différentes. Ainsi, lorsque ℓ_1 et ℓ_2 sont deux adresses distinctes, en notant $\rho_1 = \{\ldots, f_i : \tau_i, \ldots\}_{r_1}$ et $\rho_2 = \{\ldots, f_j : \tau_j, \ldots\}_{r_2}$ les régions respectivement de ℓ_1 et de ℓ_2, on a alors $r_1 \neq r_2$ et donc $\rho_1 \neq \rho_2$. En résumé, une région correspond à un type singleton habité par une seule adresse mémoire. Aussi pouvons-nous établir la correspondance exacte entre les régions (entités statiques) et les adresses mémoire qu'un programme manipule (entités dynamiques) et *a fortiori* nous sommes en mesure de déterminer si deux noms symboliques sont ou non des alias. Par exemple, si l'on prend deux régions ρ_1 et ρ_2 définies par

$$\rho_1 = \{f : \texttt{Int}; g : \{h : \texttt{Bool}\}_{r_0}\}_{r_1} \qquad \rho_2 = \{f : \texttt{Int}; g : \{h : \texttt{Bool}\}_{r_0}\}_{r_2},$$

nous savons qu'elles partagent la même région $\{h : \texttt{Bool}\}_{r_0}$. Par conséquent, si deux variables x_1 et x_2 sont respectivement typées par ρ_1 et ρ_2, nous savons alors que les expressions $x_1.g$ et $x_2.g$ sont nécessairement aliasées.

Étant donnée une région $\rho = \{\ldots, f : \tau, \ldots\}_r$, on introduit la notation $\rho.f$ pour désigner le type τ du champ f. Si la région ρ ne contient pas de champ f, alors on dit que l'expression $\rho.f$ n'est pas définie. De même, l'expression $\nu.f$ n'est pas définie pour un type scalaire ν. Par ailleurs, on étend cette notation pour les chemins, en posant $\tau.\epsilon \triangleq \tau$ et $\tau.f\pi \triangleq (\tau.f).\pi$. Enfin, on note $\mathcal{R}(\tau)$ pour désigner l'ensemble de toutes les

régions apparaissant dans un type τ.

Nous disons que deux types τ_1 et τ_2 sont *structurellement égaux*, ce que l'on note par $\tau_1 \simeq \tau_2$, lorsqu'ils sont égaux à des identificateurs près. De manière équivalente, on a $\tau_1 \simeq \tau_2$ lorsque, pour tout chemin π, $\tau_1.\pi$ est défini si et seulement si $\tau_2.\pi$ est défini et si, lorsque $\tau_1.\pi$ ou $\tau_2.\pi$ est un type scalaire, alors $\tau_1.\pi = \tau_2.\pi$.

Nous aurons également besoin par la suite de substituer des régions par des régions. Une *substitution de régions* θ est une fonction injective à domaine fini associant des types à des types structurellement égaux et telle que, pour toute région $\rho \in \operatorname{dom}\theta$ et tout chemin π, ou bien $\theta(\rho.\pi) = \theta(\rho).\pi$, ou bien $\rho.\pi$ et $\theta(\rho).\pi$ sont tous les deux indéfinis. Par exemple, en reprenant les régions ρ_1, ρ_2 de ci-dessus, si l'on pose

$$\theta = \{\rho_1 \mapsto \rho_2, \rho_0 \mapsto \rho_0, \texttt{Int} \mapsto \texttt{Int}, \texttt{Bool} \mapsto \texttt{Bool}\},$$

alors on obtient l'égalité $\theta(\rho_1) = \rho_2$. En revanche, si l'on prend deux régions ρ_3 et ρ_4

$$\rho_3 = \{f : \rho_1; g : \rho_1\}_{r_3} \qquad \rho_4 = \{f : \rho_1; g : \rho_2\}_{r_4},$$

alors il n'existe pas de substitution θ telle que $\theta(\rho_3) = \rho_4$ ou $\theta(\rho_4) = \rho_3$. En effet, une telle substitution ne sera pas injective. Intuitivement, cela reflète le fait qu'avec une substitution de régions on ne peut unifier que des types structurellement égaux qui ont la même structure d'alias.

Enfin, pour tout type τ_1, on a clairement $\tau_1 = \theta(\tau_1)$ où θ est l'identité. De plus, lorsqu'il existe une substitution de régions θ telle que $\tau_1 = \theta(\tau_2)$, on a clairement $\tau_2 = \theta^{-1}(\tau_1)$, et lorsqu'il existe θ_1, θ_2 telles que $\tau_1 = \theta_1(\tau_2)$ et $\tau_2 = \theta_2(\tau_3)$, alors $\tau_1 = \theta_1 \circ \theta_2(\tau_3)$. L'existence d'une substitution de régions induit donc une relation d'équivalence entre les types. On notera dans ce cas pour deux types équivalents τ_1 et τ_2, leur équivalence par $\tau_1 \equiv \tau_2$.

Environnements de typage. Dans notre langage, ni les variables, ni les adresses mémoires ne sont annotées par leurs types. Un tel style de présentation, où l'on définit d'abord la syntaxe et la sémantique du langage, puis seulement après un système de types, s'appelle une présentation « *à la Curry* » (par opposition à une présentation « *à la Church* »). Par conséquent, le typage d'une expression atomique se fait sous une hypothèse que le type de l'atome est cohérent avec le type de l'expression où il apparaît. Pour cela, nous utilisons un *environnement de typage des variables*, noté Γ, défini comme fonction totale des variables vers des types. L'écriture $\Gamma[x \mapsto \tau]$ représente l'environnement dans lequel on associe à x le type τ et à toute variable y différente de x le type $\Gamma(y)$. On introduit également un *environnement de typage des valeurs*, noté Σ, défini comme une fonction totale qui associe des types scalaires aux constantes scalaires et des régions aux adresses mémoires.

3.2.2 Effets

Pour détecter et résoudre les conflits d'aliasing, nous associons à chaque expression bien typée un effet ε défini comme une paire $(\omega \cdot \varphi)$ constituée de deux ensembles *disjoints* ω et φ. L'ensemble ω représente les régions potentiellement modifiées alors que l'ensemble φ représente des régions « invalidées » dont l'utilisation ultérieure est interdite ou restreinte. Lorsqu'une expression est *pure*, c'est-à-dire qu'elle n'a pas d'effets observables, les deux ensembles sont vides. Dans ce cas là, on note $\varepsilon = \bot$ et on dit que l'effet est vide.

Commençons par expliquer d'une manière informelle l'intuition derrière les ensembles $(\omega \cdot \varphi)$. Pour cela, considérons une séquence $e_1; e_2$. Notons ρ la région d'une adresse mémoire (ou d'une variable libre) de e_2 et $(\omega_1 \cdot \varphi_1)$ l'effet de e_1. Ce que nous souhaitons garantir, c'est que l'utilisation de la région ρ dans e_2 soit *valide* vis-à-vis de l'effet $(\omega_1 \cdot \varphi_1)$. D'une part, cela signifie que ρ ne doit pas être une région invalidée c'est-à-dire $\rho \notin \varphi_1$. D'autre part, cela signifie que l'ensemble ω_1 représente non seulement les régions qui sont modifiées dans e_1 mais aussi les seules régions à travers lesquelles on puisse accéder depuis ρ à des régions sinon inaccessibles dans φ_1. Pour rendre formelle cette interaction entre les types et les effets, nous introduisons la définition suivante :

Définition 3.2.1 (Prédicat de validité)**.** On dit qu'un type τ est valide vis-à-vis de l'effet $(\omega \cdot \varphi)$, ce que l'on note $(\omega \cdot \varphi) \triangleright \tau$, si et seulement si tout chemin partant de τ vers une région de φ rencontre au moins une région de ω. Formellement, on définit le prédicat $(\omega \cdot \varphi) \triangleright \tau$ inductivement par les trois règles d'inférence suivantes :

$$\frac{}{(\omega \cdot \varphi) \triangleright \nu}(\text{VP}_\tau) \qquad \frac{\rho \in \omega}{(\omega \cdot \varphi) \triangleright \rho}(\text{VP}_\omega) \qquad \frac{\rho \notin \omega \quad \rho \notin \varphi \quad \forall i.\, (\omega \cdot \varphi) \triangleright \rho.f_i}{(\omega \cdot \varphi) \triangleright \rho}(\text{VP}_{\text{IND}})$$

Vérifions d'abord que cette définition reflète bien le fait que les régions invalidées ne sont pas valides :

Lemma 3.2.1. Quels que soient l'effet $(\omega \cdot \varphi)$ et la région ρ, $(\omega \cdot \varphi) \triangleright \rho \implies \rho \notin \varphi$.

Démonstration. Soit ρ une région et $(\omega \cdot \varphi)$ un effet tels que ρ est valide vis-à-vis de $(\omega \cdot \varphi)$. Si la dérivation de $(\omega \cdot \varphi) \triangleright \rho$ se termine par la règle VP_ω, alors $\rho \in \omega$. Dans ce cas-ci, on a nécessairement $\rho \notin \varphi$, car, par définition, ω et φ sont disjoints. Sinon, la dérivation de $(\omega \cdot \varphi) \triangleright \rho$ se termine par la règle VP_{IND}. Dans ce cas-là, $\rho \notin \varphi$ d'après la prémisse de la règle, ce qui permet de conclure. □

Notons que la réciproque du lemme ci-dessus est fausse : il suffit qu'il y ait une région ρ' avec $\rho' \in \mathcal{R}(\rho) \cap \varphi$ et qu'il existe un chemin de ρ à ρ' qui ne rencontre pas de régions de ω, pour que l'on ait $\neg(\omega \cdot \varphi) \triangleright \rho$.

Comme nous le verrons par la suite, lorsque l'expression que l'on souhaite typer est une conditionnelle ou une liaison locale, on est amené à exprimer son effet en fonction des effets de ses sous-expressions. Pour cela, nous introduisons la définition suivante de l'*effet union* de deux effets :

Définition 3.2.2 (Effet union)**.** On appelle l'effet union de deux effets $\varepsilon_1 = (\omega_1 \cdot \varphi_1)$ et $\varepsilon_2 = (\omega_2 \cdot \varphi_2)$, dénoté par $\varepsilon_1 \sqcup \varepsilon_2$, l'effet $(\{\rho \in \omega_1 \mid \varepsilon_2 \triangleright \rho\} \cup \{\rho \in \omega_2 \mid \varepsilon_1 \triangleright \rho\} \cdot \varphi_1 \cup \varphi_2)$.

Avant d'expliquer l'intuition derrière cette définition, vérifions d'abord que l'effet union ainsi défini est bien un effet, c'est-à-dire que l'ensemble des régions modifiées et des régions invalidées dans $\varepsilon_1 \sqcup \varepsilon_2$ sont disjoints. Pour cela, notons $(\omega \cdot \varphi) = \varepsilon_1 \sqcup \varepsilon_2$ et considérons une région $\rho \in \omega$. Si ρ appartient à ω_1 avec $\varepsilon_2 \triangleright \rho$, alors comme $(\omega_1 \cdot \varphi_1)$ est un effet, $\rho \notin \varphi_1$ et comme $\varepsilon_2 \triangleright \rho$, $\rho \notin \varphi_2$ d'après le lemme 3.2.1. Donc $\rho \notin \varphi$. Le raisonnement est exactement le même lorsque ρ appartient ω_2 avec $\varepsilon_1 \triangleright \rho$. L'union $\varepsilon_1 \sqcup \varepsilon_2$ de deux effets est donc bien un effet. Vérifions également que l'union de deux effets vérifie la propriété suivante :

Lemma 3.2.2. Quels que soient les effets ε_1, ε_2 et le type τ, le jugement $\varepsilon_1 \sqcup \varepsilon_2 \triangleright \tau$ est valide si et seulement si les jugements $\varepsilon_1 \triangleright \rho$ et $\varepsilon_2 \triangleright \tau$ sont valides.

Démonstration. Soit un type τ et deux effets $\varepsilon_1 = (\omega_1 \cdot \varphi_1)$, $\varepsilon_2 = (\omega_2 \cdot \varphi_2)$. Notons $(\omega \cdot \varphi) = \varepsilon_1 \sqcup \varepsilon_2$. Montrons d'abord que $\varepsilon_1 \sqcup \varepsilon_2 \triangleright \tau$ implique $\varepsilon_1 \triangleright \rho$ et $\varepsilon_2 \triangleright \tau$. La preuve se fait par récurrence sur la dérivation de $\varepsilon_1 \sqcup \varepsilon_2 \triangleright \tau$. Dans le cas de base $\varepsilon_1 \sqcup \varepsilon_2 \triangleright \nu$ le résultat est trivial. Dans le cas VP_ω, on a $\rho \in \omega$ et il y a deux cas à considérer. Si $\rho \in \omega_1$ avec $\varepsilon_2 \triangleright \rho$, alors on dérive $\varepsilon_1 \triangleright \rho$ par la règle VP_ω. Sinon, $\rho \in \omega_2$ avec $\varepsilon_1 \triangleright \rho$ et, de même, on dérive $\varepsilon_2 \triangleright \rho$ par la règle VP_ω.

Considérons le cas où $\varepsilon_1 \sqcup \varepsilon_2 \triangleright \rho$ est dérivé par la règle VP_{IND}. Par hypothèse, on a $\rho \notin \omega$, $\rho \notin \varphi$ et $\forall i.\, \varepsilon_1 \sqcup \varepsilon_2 \triangleright \rho.f_i$. Comme ω_1 et φ_1 sont respectivement des sous-ensembles de ω et de φ, on a $\rho \notin \omega_1$ et $\rho \notin \varphi_1$. Par ailleurs, par hypothèse de récurrence on a $\varepsilon_1 \triangleright \rho.f_i$ pour tout i. Ainsi, $\varepsilon_1 \triangleright \rho$ se dérive par la règle VP_{IND}. Par le même raisonnement on vérifie que $\varepsilon_2 \triangleright \rho$, ce qui permet de conclure pour le sens direct de l'implication.

Prouvons maintenant la réciproque. Supposons que $(\omega_1 \cdot \varphi_1) \triangleright \tau$ et $(\omega_2 \cdot \varphi_2) \triangleright \tau$. Montrons par induction sur la dérivation de $(\omega_1 \cdot \varphi_1) \triangleright \tau$ que l'on a bien $\varepsilon_1 \sqcup \varepsilon_2 \triangleright \tau$.

Le cas de base $\tau = \nu$ est trivial. Supposons alors que $(\omega_1 \cdot \varphi_1) \triangleright \rho$ est dérivé par la règle VP_ω. On a donc $\rho \in \omega_1$ et par hypothèse $(\omega_2 \cdot \varphi_2) \triangleright \rho$. Ainsi $\rho \in \{\hat{\rho} \in \omega_1 \mid \varepsilon_2 \triangleright \hat{\rho}\} \subseteq \omega$ et on dérive le jugement $(\omega \cdot \varphi) \triangleright \rho$ par la règle VP_ω.

Enfin, traitons le cas où la dérivation de $(\omega_1 \cdot \varphi_1) \triangleright \rho$ se termine par la règle VP_{IND}. D'après les prémisses de la règle, on a $\rho \notin \omega_1$, $\rho \notin \varphi_1$ et $\forall i.\, (\omega_1 \cdot \varphi_1) \triangleright \rho.f_i$. On a deux cas à considérer. Si la dérivation de $(\omega_2 \cdot \varphi_2) \triangleright \rho$ se termine par la règle VP_ω, alors sachant que $\rho \in \omega_2$ et que $\rho \in \{\hat{\rho} \in \omega_2 \mid \varepsilon_1 \triangleright \hat{\rho}\} \subseteq \omega$, on obtient $(\omega \cdot \varphi) \triangleright \rho$ par la règle VP_ω. Sinon, la dérivation de $(\omega_2 \cdot \varphi_2) \triangleright \rho$ se termine par la règle VP_{IND} avec $\rho \notin \omega_2$, $\rho \notin \varphi_2$, et $\forall i.\, (\omega_2 \cdot \varphi_2) \triangleright \rho.f_i$. On applique alors l'hypothèse de récurrence à chacun des jugements $(\omega_1 \cdot \varphi_1) \triangleright \rho.f_i$, ce qui donne $\forall i.\, (\omega \cdot \varphi) \triangleright \rho.f_i$. Par ailleurs, on sait que $\rho \notin \omega$ et $\rho \notin \varphi$. On dérive alors $(\omega \cdot \varphi) \triangleright \rho$ par la règle VP_{IND} ce qui permet de conclure. □

Expliquons maintenant l'intuition derrière cette définition. Tout d'abord, l'ensemble des régions invalidées de $\varepsilon_1 \sqcup \varepsilon_2$ doit nécessairement inclure l'union $\varphi_1 \cup \varphi_2$ puisque l'on ne connaît pas statiquement laquelle des deux branches sera évaluée. Le lecteur peut se demander néanmoins pourquoi notre définition fait intervenir le prédicat de validité dans la définition de ω. Premièrement, remarquons qu'il serait incorrect de définir ω par une simple union $\omega_1 \cup \omega_2$, car les ensembles $\omega_1 \cup \omega_2$ et $\varphi_1 \cup \varphi_2$ ne sont pas nécessairement disjoints. Deuxièmement, supposons que l'on définisse ω d'une manière plus simple par co-restriction $(\omega_1 \setminus \varphi_2) \cup (\omega_2 \setminus \varphi_1)$. On vérifie cette fois que l'union $\varepsilon_1 \sqcup \varepsilon_2$ est un effet. Par ailleurs, on peut vérifier l'implication

$$\varepsilon_1 \triangleright \rho \ \wedge \ \varepsilon_2 \triangleright \rho \implies ((\omega_1 \setminus \varphi_2) \cup (\omega_2 \setminus \varphi_1) \cdot \varphi_1 \cup \varphi_2) \triangleright \rho.$$

Cependant, l'ennui avec une telle définition de l'union, c'est que la réciproque de l'implication (et donc le résultat du lemme 3.2.2) est fausse : la validité du jugement $((\omega_1 \setminus \varphi_2) \cup (\omega_2 \setminus \varphi_1) \cdot \varphi_1 \cup \varphi_2) \triangleright \rho$ n'implique pas simultanément la validité de $\varepsilon_1 \triangleright \rho$ et $\varepsilon_2 \triangleright \rho$. Pour s'en persuader, considérons une région ρ telle que ρ appartient à ω_1 mais n'appartient ni à φ_2, ni à ω_2. Alors que l'on a $(\omega_1 \cdot \varphi_1) \triangleright \rho$, il suffit de poser $\rho = \{f : \rho'\}_r$ avec $\rho' \in \varphi_2$ (ce qui ne contredit pas les hypothèses faites) pour que l'on ne puisse pas avoir $(\omega_2 \cdot \varphi_2) \triangleright \rho$. Revenons donc à l'effet union tel qu'il est donné dans la définition 3.2.2. Remarquons que pour cette définition-là, les effets possèdent des propriétés algébriques suivantes.

Lemma 3.2.3. L'ensemble des effets forment un semi-treillis borné sur $\sqcup$ et $\bot$.

Démonstration. Vérifions que $\bot$ est un élément neutre. En effet, pour tout effet ε, on a $\varepsilon \sqcup \bot = \bot \sqcup \varepsilon = \varepsilon$. D'autre part, $\sqcup$ est idempotente. En effet, $\omega = \{\rho \in \omega \mid (\omega \cdot \varphi) \triangleright \rho\}$, donc $(\omega \cdot \varphi) \sqcup (\omega \cdot \varphi) = (\omega \cdot \varphi)$. L'opération $\sqcup$ est également commutative, puisque l'union de $(\omega_1 \cdot \varphi_1)$ et $(\omega_2 \cdot \varphi_2)$ définie comme $(\{\rho \in \omega_1 \mid \varepsilon_2 \triangleright \rho\} \cup \{\rho \in \omega_2 \mid \varepsilon_1 \triangleright \rho\} \cdot \varphi_1 \cup \varphi_2)$ est clairement égale à $(\{\rho \in \omega_2 \mid \varepsilon_1 \triangleright \rho\} \cup \{\rho \in \omega_1 \mid \varepsilon_2 \triangleright \rho\} \cdot \varphi_2 \cup \varphi_1)$. Enfin, vérifions que l'opération $\sqcup$ est associative. Notons $\omega \mid \varepsilon$ pour $\{\rho \in \omega \mid \varepsilon \triangleright \rho\}$. Remarquons que, quels que soient ω, ε et ε', d'après le lemme 3.2.2, on a bien $(\omega \mid \varepsilon) \mid \varepsilon' = \omega \mid \varepsilon \sqcup \varepsilon' = (\omega \mid \varepsilon') \mid \varepsilon$. Par conséquent, quels que soient ε_1, ε_2, ε_3, on a

$$\begin{aligned}
&(\varepsilon_1 \sqcup \varepsilon_2) \sqcup \varepsilon_3 \\
&= (\omega_1 \mid \varepsilon_2 \cup \omega_2 \mid \varepsilon_1 \cdot \varphi_1 \cup \varphi_2) \sqcup \varepsilon_3 && \text{(def)} \\
&= ((\omega_1 \mid \varepsilon_2 \cup \omega_2 \mid \varepsilon_1) \mid \varepsilon_3 \cup \omega_3 \mid \varepsilon_1 \sqcup \varepsilon_2 \cdot \varphi_1 \cup \varphi_2 \cup \varphi_3) && \text{(def)} \\
&= ((\omega_1 \mid \varepsilon_2) \mid \varepsilon_3 \cup (\omega_2 \mid \varepsilon_1) \mid \varepsilon_3 \cup \omega_3 \mid \varepsilon_1 \sqcup \varepsilon_2 \cdot \varphi_1 \cup \varphi_2 \cup \varphi_3) \\
&= (\omega_1 \mid \varepsilon_2 \sqcup \varepsilon_3 \cup (\omega_2 \mid \varepsilon_3) \mid \varepsilon_1 \cup (\omega_3 \mid \varepsilon_2) \mid \varepsilon_1 \cdot \varphi_1 \cup \varphi_2 \cup \varphi_3) \\
&= (\omega_1 \mid \varepsilon_2 \sqcup \varepsilon_3 \cup (\omega_2 \mid \varepsilon_3 \cup \omega_3 \mid \varepsilon_2) \mid \varepsilon_1 \cdot \varphi_1 \cup \varphi_2 \cup \varphi_3) \\
&= \varepsilon_1 \sqcup (\omega_2 \mid \varepsilon_3 \cup \omega_3 \mid \varepsilon_2 \cdot \varphi_2 \cup \varphi_3) && \text{(def)} \\
&= \varepsilon_1 \sqcup (\varepsilon_2 \sqcup \varepsilon_3) && \text{(def)}
\end{aligned}$$

□

La conséquence directe de ce lemme est qu'il induit la relation d'ordre partiel suivante.

Définition 3.2.3. Les effets induisent une relation d'ordre partiel $\sqsubseteq$ définie par $\varepsilon_1 \sqsubseteq \varepsilon_2$ si est seulement $\varepsilon_2 = \varepsilon_1 \sqcup \varepsilon_2$. On dit alors que l'effet ε_1 est un *sous-effet* de ε_2.

On a immédiatement le résultat suivant :

Lemma 3.2.4. Si $\varepsilon_1 \sqsubseteq \varepsilon_2$, alors pour tout τ, la validité du jugement $\varepsilon_2 \triangleright \tau$ implique celle du jugement $\varepsilon_1 \triangleright \tau$.

3.2.3 Typage des fonctions

Rappelons que les fonctions qui sont appelées dans une expression sont définies dans l'environnement global P. Avant de présenter les règles du typage, nous avons donc besoin de préciser quelles sont les hypothèses de typage que nous faisons pour les définitions de fonctions dans P.

Premièrement, nous supposons qu'à chaque définition $p(x_1, \ldots, x_n) \mapsto e$ est associée une *signature de type* que l'on dénote par $\tau_1 \times \ldots \times \tau_n \to \tau \cdot (\omega \cdot \varphi)$ et où $\tau_1, \ldots, \tau_n$ sont les types des paramètres formels, τ est le type du résultat et $(\omega \cdot \varphi)$ est l'effet latent de la fonction. Nous supposons que toutes les signatures vérifient les trois conditions suivantes : $\omega \subseteq \mathcal{R}(\tau_1, \ldots, \tau_n)$, $\varphi \subseteq \mathcal{R}(\tau_1, \ldots, \tau_n, \tau)$ et $\mathcal{R}(\tau) \setminus \mathcal{R}(\tau_1, \ldots, \tau_n) \subseteq \varphi$. Les deux premières conditions limitent la portée de l'effet latent aux régions exposées par la signature dans les types des paramètres formels et le type de retour. Remarquons que ces restrictions vont de pair avec les restrictions $\mathcal{F}_\ell(e) = \varnothing$ et $\mathcal{F}_v(e) \subseteq \{x_1, \ldots, x_n\}$ que nous avons imposées dans la sémantique opérationnelle. Remarquons également que l'on aurait pu poser la première condition à l'instar de la seconde, en ajoutant le type de retour à droite de l'inclusion mais cela est inutile : dans notre système de type, toutes les régions fraîches d'une expression appartiennent nécessairement à l'ensemble des régions invalidées de l'effet correspondant. Or, la troisième condition garantit que toute nouvelle région qui fait partie de type du résultat fait partie des régions invalidées. Par conséquent, si l'on prend par exemple deux fonctions p_0 et p_1 :

$$p_0(x) \mapsto \texttt{let } r = \{f = x\} \texttt{ in } r.f \leftarrow 1; r \qquad p_1(x) \mapsto \{f = x\}$$

on constate qu'elles ont chacune un effet de la même forme $(\varnothing \cdot \{\rho\})$. Il en résulte donc que si $\rho \in \mathcal{R}(\tau) \setminus \mathcal{R}(\tau_1, \ldots, \tau_n)$ alors $\rho \in \varphi$, et donc $\rho \notin \omega$, puisque ω et φ sont disjoints par définition.

En ce qui concerne les fonctions primitives, nous supposons que leurs signatures sont toutes de la forme $\nu_1 \times \ldots \times \nu_n \to \nu \cdot \bot$ et vérifient donc trivialement les trois conditions ci-dessus. Par ailleurs, on suppose que ces signatures sont cohérentes avec l'oracle sémantique δ, c'est-à-dire que si p est une fonction primitive de signature $\nu_1 \times \ldots \times \nu_n \to \nu \cdot \bot$, alors quelles que soient les constantes $c_1, \ldots, c_n$ telles que c_i ait le type ν_i pour $1 \leq i \leq n$, alors la constante $\delta(p, c_1, \ldots, c_n)$ est bien définie et a le type ν.

Deuxièmement, nous supposons que le corps d'une fonction est bien typé et que la dérivation du typage de e se termine par $\Gamma[x_i \mapsto \tau_i \ ^{i \in [1, \ldots, n]}] \cdot \Sigma \vdash e : \tau \cdot (\omega \cdot \varphi \cup \varphi'')$ où

$\varphi'' \cap \mathcal{R}(\tau_1, \ldots, \tau_n, \tau) = \varnothing$ (en particulier donc φ et φ'' sont deux ensembles disjoints). Les régions de φ'' représentent les adresses mémoire qui sont allouées pendant l'appel et mais resteront locales à la définition de la fonction dans le sens qu'elles ne font pas partie des régions du résultat. Notons que le caractère « local » de telles adresses est reflété *statiquement* par le fait que les régions φ'' ne sont pas exposées dans la signature de type de la fonction mais, dans notre formalisme, il n'est pas reflété *dynamiquement* : il n'y pas à proprement parler d'adresses mémoire locales du point de vue de la sémantique opérationnelle. Pour faire apparaître cette distinction, il faudrait par exemple introduire un mécanisme de ramasse-miettes qui supprimerait ces adresses de l'état mémoire lorsque le résultat de la fonction serait renvoyé ou encore modéliser l'exécution d'appel d'une fonction avec une structure de pile explicite séparée de l'état mémoire global : les adresses mémoire caractérisées par les régions de φ'' seraient alors typiquement celles que l'on allouerait sur la pile et non sur le tas. Cependant, la question de localité et de durée de vie des adresses mémoire a principalement trait à la sûreté d'accès à la mémoire, une question bien étudiée dans la littérature sur les régions mais qui n'est pas la nôtre, en particulier parce que nous n'avons pas introduit d'opération de désallocation. Nous n'en disons donc pas plus et présentons maintenant les règles du typage.

3.2.4 Règles du typage

Les règles du typage sont données dans la figure 3.5. Les règles T-Var, T-Cst, T-Loc décrivent le typage des expressions atomiques dont le type est donné par l'environnement du typage Γ ou Σ selon qu'il s'agisse d'une variable ou d'une valeur.

Le typage de l'accès au champs f d'une expression atomique a est donné par la règle T-Field. Les prémisses de la règle garantissent que le type de a est nécessairement une région ρ et que $\rho.f$ est un type bien défini. Comme pour les expressions atomiques, l'effet de l'expression $a.f$ est vide.

La règle T-Let décrit le typage d'une liaison locale `let` $x = e_1$ `in` e_2. En supposant que les sous-expressions e_1 et e_2 soient typées respectivement avec $\Gamma \cdot \Sigma \vdash e_1 : \tau_1 \cdot \varepsilon_1$ et $\Gamma[x \mapsto \tau_1] \cdot \Sigma \vdash e_2 : \tau_2 \cdot \varepsilon_2$, la prémisse

$$\forall \rho \in \Gamma(\mathcal{F}_v(e_2) \setminus \{x\}) \cup \Sigma(\mathcal{F}_\ell(e_2)).\ \varepsilon_1 \triangleright \rho$$

garantit que les régions des variables libres autres que x, ainsi que les adresses mémoire de e_2, sont toutes valides vis-à-vis de l'effet de e_1. C'est donc lors de la liaison locale que les effets sont non seulement propagés mais également utilisés pour vérifier que l'on est toujours en mesure de déterminer statiquement tous les alias.

Le typage de l'appel $p(a_1, \ldots, a_n)$ est donné par la règle T-Call. En supposant que la signature de la fonction p est $\tau_1 \times \ldots \times \tau_n \to \tau \cdot \varepsilon$, les prémisses de la règle imposent l'existence d'une substitution de régions θ telle que chaque argument a_i ait le type $\theta(\tau_i)$ et telle que le type du retour et l'effet de l'appel soient respectivement $\theta(\tau)$ et $\theta(\varepsilon)$. Le domaine de θ est donc égal à $\mathcal{R}(\tau_1, \ldots, \tau_n, \tau)$, ce qui par définition de

$$\frac{\Gamma(x) = \tau}{\Gamma \cdot \Sigma \vdash x : \tau \cdot \bot}\ (\text{T-Var}) \qquad \frac{\Sigma(c) = \nu}{\Gamma \cdot \Sigma \vdash c : \nu \cdot \bot}\ (\text{T-Cst})$$

$$\frac{\Gamma \cdot \Sigma \vdash a : \rho \cdot \bot \quad \rho.f \text{ est défini}}{\Gamma \cdot \Sigma \vdash a.f : \rho.f \cdot \bot}\ (\text{T-Field}) \qquad \frac{\Sigma(\ell) = \rho}{\Gamma \cdot \Sigma \vdash \ell : \rho \cdot \bot}\ (\text{T-Loc})$$

$$\frac{\begin{array}{c}\Gamma \cdot \Sigma \vdash e_1 : \tau_1 \cdot \varepsilon_1 \qquad \Gamma[x \mapsto \tau_1] \cdot \Sigma \vdash e_2 : \tau_2 \cdot \varepsilon_2 \\ \forall \rho \in \Gamma(\mathcal{F}_v(e_2) \setminus \{x\}) \cup \Sigma(\mathcal{F}_\ell(e_2)).\ \varepsilon_1 \triangleright \rho\end{array}}{\Gamma \cdot \Sigma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2 \cdot \varepsilon_1 \sqcup \varepsilon_2}\ (\text{T-Let})$$

$$\frac{\Gamma \cdot \Sigma \vdash a : \texttt{Bool} \cdot \bot \qquad \Gamma \cdot \Sigma \vdash e_1 : \tau \cdot \varepsilon_1 \qquad \Gamma \cdot \Sigma \vdash e_2 : \tau \cdot \varepsilon_2}{\Gamma \cdot \Sigma \vdash \texttt{if}\ a\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau \cdot \varepsilon_1 \sqcup \varepsilon_2}\ (\text{T-If})$$

$$\frac{\Gamma \cdot \Sigma \vdash a_i : \tau_i \cdot \bot \qquad \rho = \{f_i : \tau_i\ ^{i \in [1, \ldots, n]}\}_r \qquad f_i \text{ deux à deux distincts}}{\Gamma \cdot \Sigma \vdash \{f_i = a_i\ ^{i \in [1, \ldots, n]}\} : \rho \cdot (\varnothing \cdot \{\rho\})}\ (\text{T-Alloc})$$

$$\frac{\begin{array}{c}\Gamma \cdot \Sigma \vdash a_i : \rho_i \cdot \bot \qquad \Gamma \cdot \Sigma \vdash a'_{i,j} : \tau'_{i,j} \cdot \bot \qquad \rho_i.f_{i,j} \simeq \tau'_{i,j} \\ \rho_i \text{ deux à deux distincts} \qquad \forall i.\ f_{i,j} \text{ deux à deux distincts} \\ \varphi = \Phi(\rho_i.\{f_{i,j} \leftarrow \tau'_{i,j}\ ^{j \in [1, \ldots, k_i]}\}^{i \in [1, \ldots, n]})\end{array}}{\Gamma \cdot \Sigma \vdash a_i.\{f_{i,j} \leftarrow a'_{i,j}\ ^{j \in [1, \ldots, k_i]}\}^{i \in [1, \ldots, n]} : \texttt{Unit} \cdot (\{\rho_1, \ldots, \rho_n\} \cdot \varphi)}\ (\text{T-Assign})$$

$$\frac{p : \tau_1 \times \ldots \times \tau_n \to \tau \cdot \varepsilon \qquad \Gamma \cdot \Sigma \vdash a_i : \theta(\tau_i) \cdot \bot}{\Gamma \cdot \Sigma \vdash p(a_1, \ldots, a_n) : \theta(\tau) \cdot \theta(\varepsilon)}\ (\text{T-Call})$$

FIGURE 3.5 – Règles de typage.

la signature couvre également les régions de l'effet ε. Par ailleurs, θ étant injective, les régions distinctes de la signature sont instanciées par les régions distinctes à chaque endroit où l'appel de p se produit.

Remarquons que l'effet dans les règles que l'on vient de voir est soit vide, soit défini en fonction des effets de leur prémisses. C'est lors de la création d'un enregistrement et l'affectation parallèle que les effets sont engendrés. Expliquons en détails les règles T-Alloc et T-Assign correspondantes. Dans la règle T-Alloc, l'enregistrement $\{f_i =$

$a_i\,^{i\in[1,\ldots,n]}\}$ est typé avec une région fraîche ρ égale à $\{f_i : \tau_i\,^{i\in[1,\ldots,n]}\}_r$. L'indice r peut être choisi arbitrairement et rien n'impose *a priori* qu'il soit nécessairement distinct des indices des régions dans Γ et Σ. C'est en invalidant la région ρ dans l'effet $(\varnothing \cdot \{\rho\})$ associé que l'on exprime sa fraîcheur. En effet, une fois l'enregistrement introduit dans un programme, typiquement via une liaison locale $\texttt{let}\ x = \{f_i = a_i\,^{i\in[1,\ldots,n]}\}\ \texttt{in}\ e_2$, cela interdit dans e_2 l'utilisation de tout ancien habitant de ρ. Ainsi, dans le typage de l'expression $\texttt{let}\ x = \{f = 41\}\ \texttt{in}\ \texttt{let}\ y = \{f = 43\}\ \texttt{in}\ e$, les variables x et y peuvent avoir deux régions distinctes, auquel cas les deux peuvent être utilisées dans e. Mais il est également possible d'assigner à x et y la même région. Cependant, dans ce cas-là, x ne peut plus apparaître dans e, car sa région est invalidée par l'effet de l'allocation de y. Notons en passant que cela implique que notre système de type ne possède pas la propriété de jugement principal (« principal typing » en anglais, voir Wells [32]).

Le typage d'une affectation parallèle $a_i.\{f_{i,j} \leftarrow a'_{i,j}\,^{j\in[1,\ldots,k_i]}\}^{i\in[1,\ldots,n]}$ est donné par la règle T-Assign. Les types des expressions atomiques $a_1, \ldots, a_n$ ainsi que les noms des champs $f_{i,j}$ doivent être deux à deux distincts. Dans le cas contraire, une expression telle que

$$\texttt{let}\ x = \{f = 42\}\ \texttt{in}\ \texttt{let}\ y = x\ \texttt{in}\ x.f \leftarrow 42, y.f \leftarrow 32$$

serait évaluée après trois pas d'exécution (E-Alloc, E-ζ, E-ζ) en $\ell.f \leftarrow 42, \ell.f \leftarrow 32$ dont l'évaluation serait bloquée, car la prémisse de la règle E-Assign ne serait pas vérifiée. D'autre part, chaque expression $a'_{i,j}$ « à droite » de l'affectation doit être bien typée respectivement par le type bien défini $\tau'_{i,j}$. Par conséquent, la prémisse $\rho.f_{i,j} \simeq \tau'_{i,j}$ garantit que chacune des expressions de types $\rho.f_{i,j}$ est également bien définie. Enfin, l'affectation parallèle ne peut être bien typée que si les conflits d'aliasing qu'elle a produits peuvent être tous résolus. Cela est exprimé dans la règle T-Assign par la prémisse

$$\varphi = \Phi(\rho_i.\{f_{i,j} \leftarrow \tau'_{i,j}\,^{j\in[1,\ldots,k_i]}\}^{i\in[1,\ldots,n]})$$

où, Φ représente l'opération qui calcule l'ensemble des régions invalidées, lorsque les conflits d'aliasing peuvent être résolus. Détaillons la manière dont Φ est définie.

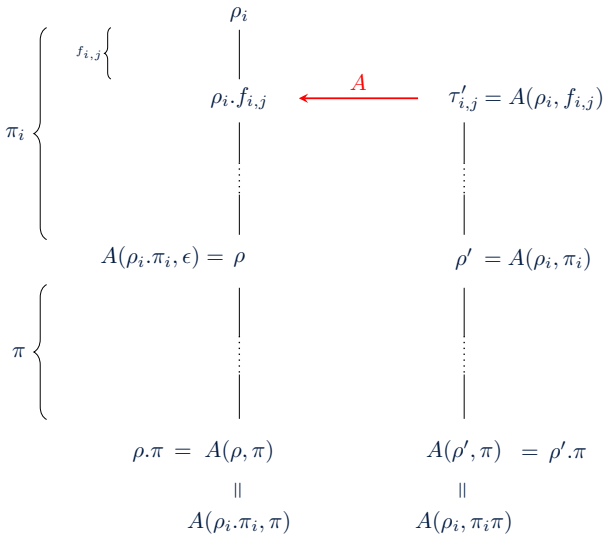
Désignons par A l'argument de Φ, qui est simplement la projection de l'affectation dans les types. Introduisons alors pour une région ρ et un champ g la notation $A(\rho, g)$ définie par

$$A(\rho, f) \triangleq \begin{cases} \tau'_{i,j} & \text{si } \rho = \rho_i \text{ et } f = f_{i,j} \\ \rho.f & \text{sinon} \end{cases}$$

Pour tout $i \in [1, \ldots, n]$ et tout $j \in [1, \ldots, k_i]$, on écrit donc $A(\rho_i, f_{i,j}) \triangleq \tau'_{i,j}$. Deuxièmement, généralisons cette notation pour les chemins de longueur arbitraire accessibles depuis les régions affectées. Pour toute région $\rho \in \mathcal{R}(\rho_1, \ldots, \rho_n)$ et tout chemin π tel que $\rho.\pi$ soit défini, posons

$$A(\rho, \pi) \triangleq \begin{cases} \rho & \text{si } \pi = \epsilon \\ A(\rho, f) & \text{si } \pi = f\epsilon \text{ et } A(\rho, f) = \nu \\ A(\rho', \pi') & \text{si } \pi = f\pi' \text{ et } A(\rho, f) = \rho' \end{cases}$$

On peut représenter la notation $A(\rho, \pi)$ graphiquement. Considérons par exemple $A = \rho_i.f_{i,j}$ et prenons une région ρ dans $\mathcal{R}(\rho_i)$, c'est-à-dire telle que $\rho = \rho_i.\pi_i$ pour un certain chemin π_i. En considérant un chemin π partant de la région ρ, la différence entre $A(\rho_i, \pi_i\pi)$ et $A(\rho_i.\pi_i, \pi)$ est illustrée dans la figure ci-dessous :



Remarquons par ailleurs que l'on n'a pas donné la définition pour $A(\rho, \pi)$ lorsque $\pi = f\pi'$, $A(\rho, f) = \nu$ et $\pi' \neq \epsilon$. En effet, une telle situation est simplement impossible : on ne peut écrire $A(\rho, \pi)$ que lorsque $\rho.\pi$ est défini. Remarquons aussi que la prémisse $\rho_i.f_{i,j} \simeq \tau'_{i,j}$ se réécrit $A(\rho_i, f) \simeq \rho_i.f$. On en déduit par récurrence immédiate que sur la longueur du chemin π que $A(\rho, \pi) \simeq \rho.\pi$ pour tout π. Ensuite, désignons par σ_A la relation binaire suivante :

$$\sigma_A \triangleq \{\, \langle A(\rho, \pi), \rho.\pi\rangle \mid \rho \in \{\rho_1, \ldots, \rho_n\} \,\}.$$

Nous posons alors que l'opération Φ valide l'affection (qui est donc acceptée par le système du typage) si et seulement si σ_A est bijective.

Lorsque l'affectation A est bijective, $\Phi(A)$ donne alors l'ensemble des régions invalidées par l'affectation défini de la manière suivante :

$$\Phi(A) \triangleq \{\, \rho \mid \text{il existe } \rho' \neq \rho \text{ telle que } \langle \rho, \rho'\rangle \in \sigma_A \text{ ou } \langle \rho', \rho\rangle \in \sigma_A \,\}$$

Autrement dit, nous invalidons toute région qui est accessible à gauche ou à droite de l'affectation et qui n'est pas égale à son image par σ_A. Remarquons que lorsqu'une région ρ est invalidée, cela n'implique pas que toutes les sous-régions de ρ le deviennent nécessairement. Ainsi, dans l'exemple de la figure 3.6, l'affectation remplace la région $\rho.f_i$ par une région structurellement égale mais distincte ρ_4. Néanmoins, l'alias entre $\rho.f_i.g$ et $\rho_4.g$ est préservé par l'affectation. En notant cet alias ρ_2, on a $\sigma_A(\rho_2) = \rho_2$, donc la région ρ_2 reste valide, alors que les régions $\rho.f_i$ et ρ_4 sont invalidées.

Intuitivement, lorsque σ_A n'est pas bijective, cela signifie que l'affectation contient un conflit d'aliasing qui ne peut pas être résolu. Par exemple, dans la figure 3.7,

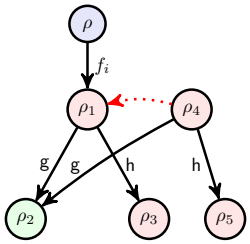


FIGURE 3.6 – $\sigma_A = (\langle\rho, \rho\rangle, \langle\rho_4, \rho_1\rangle, \langle\rho_5, \rho_3\rangle, \langle\rho_2, \rho_2\rangle)$.

l'opération d'affectation remplace la région associée au champ h_2 de la région ρ par une région ρ_4 structurellement égale mais distincte et rompt ainsi l'aliasing entre les régions $\rho.h_1.f$ et $\rho.h_2.g$. Par conséquent, la région ρ elle-même devient invalide, la contrainte ne peut être satisfaite, puisque l'ensemble des régions modifiées correspond exactement à l'ensemble $\{\rho_1, \ldots, \rho_n\}$ lequel, par définition, doit être disjoint de l'ensemble φ des régions invalidées. De manière similaire, la figure 3.8 montre l'exemple d'une affectation

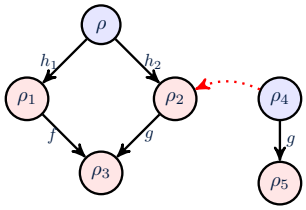


FIGURE 3.7 – $\langle\rho_3, \rho_3\rangle, \langle\rho_5, \rho_3\rangle \in \sigma_A$.

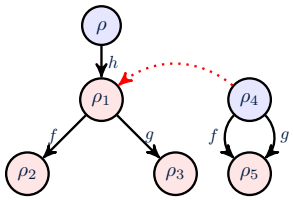


FIGURE 3.8 – $\langle\rho_5, \rho_2\rangle, \langle\rho_5, \rho_3\rangle \in \sigma_A$.

qui crée un nouvel alias entre les régions $\rho.h.f$ et $\rho.h.g$ auparavant non-aliasées, ce qui est interdit pour les mêmes raisons. Dans les deux cas, la relation σ_A correspondante n'est pas bijective et les affectations sont rejetées par le typage.

3.3 Correction du typage

Nous allons maintenant montrer que notre système de types est *sûr* et qu'il garantît que tous les alias dans programme typé sont connus statiquement. L'idée est de montrer que si le typage des adresses mémoire est cohérent avec la manière dont ces adresses sont stockées dans la mémoire, alors un pas d'évaluation préserve la cohérence. Commençons par définir formellement la notion de la cohérence entre le typage et la mémoire. Pour cela, introduisions deux définitions suivantes :

Définition 3.3.1. On dit qu'un état mémoire μ est *bien typé* dans l'environnement Σ sur l'ensemble des adresses mémoire L, noté $\Sigma \models \mu \cdot L$ si et seulement si, pour toute adresse mémoire $\ell \in L$ et tout chemin π, ou bien $\Sigma(\ell).\pi = \Sigma(\mu(\ell.\pi))$, ou bien ni $\Sigma(\ell).\pi$, ni $\mu(\ell.\pi)$ ne sont définis. On introduit également la notation $\Sigma \models \mu \cdot e$ pour $\Sigma \models \mu \cdot \mathcal{F}_\ell(e)$.

Définition 3.3.2. Étant donné un ensemble des adresses mémoire L et un état mémoire μ, on dit qu'un environnement du typage Σ est *injectif* sur le couple $\mu \cdot L$ si et seulement si, quelles que soient les adresses mémoire $\ell_1, \ell_2 \in L$ et les chemins π_1, π_2, lorsque $\Sigma(\ell_1).\pi_1$ et $\Sigma(\ell_2).\pi_2$ dénotent la même région, alors $\mu(\ell_1.\pi_1)$ et $\mu(\ell_2.\pi_2)$ dénotent la même adresse mémoire. Dans ce cas, on écrit $\Sigma \ltimes \mu \cdot L$. On introduit également la notation $\Sigma \ltimes \mu \cdot e$ pour $\Sigma \ltimes \mu \cdot \mathcal{F}_\ell(e)$.

Remarquons que la première définition implique que lorsqu'une expression e est bien typée et que $\Sigma \models \mu \cdot e$, alors $\mathcal{F}_\ell(e) \subseteq \operatorname{dom} \mu$. On peut également représenter ces définitions à l'aide de deux diagrammes commutatifs de la figure 3.9 (les flèches noires

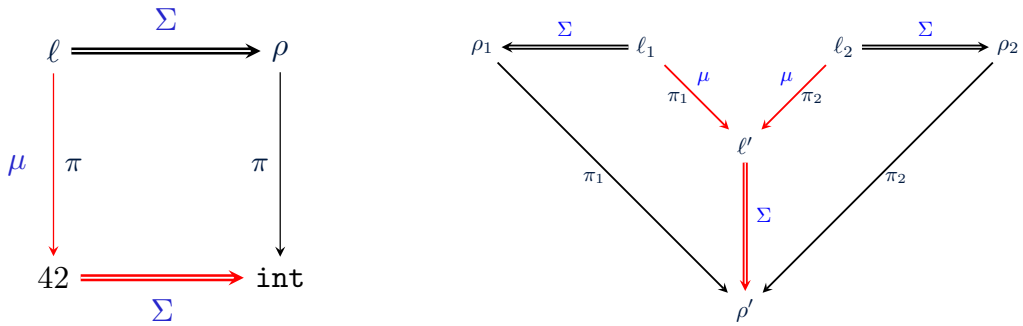


FIGURE 3.9 – Commutativité et injectivité du typage de l'état mémoire

correspondent aux hypothèses, les flèches rouges aux conclusions) où le diagramme à gauche montre la commutativité du typage avec l'état mémoire lorsque $\Sigma(\ell.\pi)$ est un type scalaire et le diagramme à droite montre à la fois leur commutativité lorsque les $\Sigma(\ell_1).\pi_1$ et $\Sigma(\ell_2).\pi_2$ sont des régions et l'injectivité du typage vis-à-vis des adresses mémoires.

Comme dans le chapitre précédent, nous montrons la correction du typage en adaptant une approche syntaxique de Wright et Felleisen [33]. Nous montrons donc séparément le lemme de progrès, à savoir qu'une expression bien typée est soit une valeur, soit une expression réductible, et le lemme de préservation, à savoir que, lorsqu'une expression réductible est bien typée et que son typage est cohérent avec l'état mémoire dans lequel on l'évalue, un pas d'évaluation, quelle que soit la règle sémantique qui s'applique, préserve la relation du typage de l'expression et la cohérence du typage avec le nouvel état mémoire.

Lemma 3.3.1 (Progrès)**.** Soit e une expression close telle que $\Gamma \cdot \Sigma \vdash e : \tau \cdot \varepsilon$. Ou bien e est une valeur, ou bien, quel que soit l'état mémoire μ tel que l'on ait $\Sigma \models \mu \cdot e$, il existe une configuration $\mu' \cdot e'$ telle que $\mu \cdot e \longrightarrow \mu' \cdot e'$.

Démonstration. Par récurrence sur la structure de e. Le cas de base où e est une expression atomique est trivial. Le cas où e est de la forme $a.f$ est vérifié en vertu de l'hypothèse $\Sigma \models \mu \cdot e$. En effet, l'expression e étant close, par typage on déduit que a est nécessairement une adresse mémoire ℓ dont le type est une région ρ. D'autre part, la prémisse de la règle du typage garantit que le type $\rho.f$ est défini. Donc, $\mu(\ell.f)$ est également définie en vertu de l'hypothèse $\Sigma \models \mu \cdot e$, ce qui permet de conclure que e est réductible.

Le cas où e est une création d'un enregistrement ou une affectation parallèle sont vérifiées de la même manière. En effet, grâce à l'hypothèse $\Sigma \models \mu \cdot e$ d'une part et les prémisses des règles du typage respectives T-Alloc et T-Assign d'autre part impliquent les prémisses des règles sémantiques respectives ℓ et E-Assign. Par conséquent, dans chacun des cas, e est réductible.

Lorsque e correspond à un appel d'une fonction primitive $q(c_1, \ldots, c_n)$, elle est encore réductible, puisque nous avons supposé que le typage des fonctions primitives est cohérent avec l'oracle sémantique δ, c'est-à-dire que la constante $\delta(q, c_1, \ldots, c_n)$ est bien définie.

Le cas où e est une liaison locale $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ avec e_1 qui n'est pas une valeur se déduit directement par hypothèse de récurrence, et tous les autres cas sont triviaux. □

Alors que démontrer le progrès consiste simplement à vérifier que les règles de typage ne « bloquent » pas la sémantique opérationnelle, démontrer la préservation nous demandera plus d'efforts. Clarifions d'abord ce que l'on veut prouver exactement. Considérons un pas de réduction $\mu \cdot e \longrightarrow \mu' \cdot e'$ où e est une expression bien typée telle que l'on a $\Gamma \cdot \Sigma \vdash e : \tau \cdot \varepsilon$, $\Sigma \models \mu \cdot e$ et $\Sigma \ltimes \mu \cdot e$. Supposons que l'on veuille montrer que le jugement $\Gamma' \cdot \Sigma' \vdash e' : \tau' \cdot \varepsilon'$ est dérivable pour certains Γ', Σ', τ' et ε'. Premièrement, il semble naturel de demander que $\tau' = \tau$. En effet, si le résultat de l'évaluation de e doit être un entier, alors on s'attend à ce qu'il en soit de même pour e'. Or, ceci est moins trivial lorsque le type de e est une région. Considérons par exemple la situation où e est égale à la séquence $\ell.f \leftarrow \ell'; \ell.f$ avec $\Sigma(\ell.f) \neq \Sigma(\ell')$.

D'après la règle du typage T-Let, e a le type $\Sigma(\ell.f)$. Or, après quelques étapes de réductions, on arrive à la configuration $\mu' \cdot \ell'$ où $\mu' = \mu[\ell.f \mapsto \ell']$. Par conséquent, si l'on veut que le type reste invariant par l'évaluation, on est amené à modifier Σ en accord avec les modifications de l'état mémoire initial. Dans notre exemple il suffit de poser pour cela $\Sigma[\Sigma(\ell') \mapsto \Sigma(\ell.f)]$ mais en général trouver un bon Σ' est moins immédiat, notamment parce que l'on doit établir que $\Sigma' \models \mu' \cdot e'$ et $\Sigma' \ltimes \mu' \cdot e'$.

Deuxièmement, il nous faut établir une relation entre les effets ε et ε'. Considérons par exemple la situation où e est égale à $\texttt{let}\ x = \{f = 42\}\ \texttt{in}\ x.f \leftarrow 0$. L'effet de e est donc $(\varnothing \cdot \{\rho\})$ où ρ est la région associée à x. Or, après un pas de réduction, on obtient que l'expression e' est égale à $\texttt{let}\ x = ()\ \texttt{in}\ x.f \leftarrow 0$ dont l'effet est $(\{\rho\} \cdot \varnothing)$. Cela montre que lors d'un pas d'évaluation l'effet d'une expression ne diminue pas nécessairement composante par composante, en particulier à cause du fait que les effets d'écriture peuvent être cachés par les effets d'invalidation de régions fraîches. Néanmoins, on peut remarquer que $(\varnothing \cdot \{\rho\}) = (\varnothing \cdot \{\rho\}) \sqcup (\{\rho\} \cdot \varnothing)$. Cela suggère que l'on ait en général $\varepsilon' \sqsubseteq \varepsilon$, c'est-à-dire que ε' soit un sous-effet de ε. Il se trouve que cela n'est vrai que lorsqu'on e ne contient aucun appel de fonctions : dans le cas contraire, la relation $\varepsilon' \sqsubseteq \varepsilon$ devient inexacte à cause du fait que le corps d'une fonction peut avoir strictement plus d'effets en comparaison avec les effets exposés dans sa signature. Au final, l'énoncé de préservation prend la forme suivante :

Lemma 3.3.2 (Préservation)**.** Quel que soit le pas d'évaluation $\mu \cdot e \longrightarrow \mu' \cdot e'$,

$$\begin{array}{l} \Gamma \cdot \Sigma \vdash e : \tau \cdot \varepsilon \\ \quad \Sigma \models \mu \cdot e \\ \quad \Sigma \ltimes \mu \cdot e \end{array} \quad \Longrightarrow \quad \exists \Sigma', \varepsilon', \varphi''. \quad \begin{array}{l} \Gamma \cdot \Sigma' \vdash e' : \tau \cdot \varepsilon' \sqcup (\varnothing \cdot \varphi'') \\ \quad \Sigma' \models \mu' \cdot e' \\ \quad \Sigma' \ltimes \mu' \cdot e' \end{array}$$

où $\varepsilon' \sqsubseteq \varepsilon$ et $\varphi'' \cap \Sigma'(\operatorname{dom} \mu') = \varnothing$.

Démonstration. Commençons par construire ε' explicitement par récurrence sur la dérivation du pas d'évaluation $\mu \cdot e \longrightarrow \mu' \cdot e'$ en vérifiant en même temps que $\varepsilon' \sqsubseteq \varepsilon$. Dans le cas des règles E-Alloc, E-Assign, E-Field et E-Op, où le résultat de l'évaluation est une valeur, on pose $\varepsilon' \triangleq \bot$, donc on a trivialement $\varepsilon' \sqsubseteq \varepsilon$. Dans le cas où e est un appel de fonction $p(v_1, \ldots, v_n)$ qui se réduit par la règle E-δ et dans le cas où e est une liaison locale $\texttt{let}\ x = v_1\ \texttt{in}\ e_2$ qui se réduit la règle E-ζ, on pose $\varepsilon' \triangleq \varepsilon$, donc on a toujours trivialement $\varepsilon' \sqsubseteq \varepsilon$. Lorsque e est une conditionnelle $\texttt{if}\ c\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ qui se réduit par la règle E-True (reps. E-False), on pose $\varepsilon' \triangleq \varepsilon_1$ (reps. $\varepsilon' \triangleq \varepsilon_2$), où ε_1 (reps. ε_2) est l'effet de e_1 (reps. e_2). Dans les deux cas, on a $\varepsilon = \varepsilon' \sqcup \hat{\varepsilon}$ où $\hat{\varepsilon}$ est l'effet de la branche ignorée, c'est-à-dire que l'on a toujours $\varepsilon' \sqsubseteq \varepsilon$. Enfin, supposons que e est une liaison locale $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ qui se réduit par la règle E-Ctx. L'expression e_1 se réduit alors elle-même. Notons ε'_1 l'effet donné par l'hypothèse de récurrence sur la réduction de e_1 et posons $\varepsilon' \triangleq \varepsilon'_1 \sqcup \varepsilon_2$ où ε_2 est l'effet de e_2. Vérifions que $\varepsilon' \sqsubseteq \varepsilon$. Par hypothèse de récurrence, on a $\varepsilon'_1 \sqsubseteq \varepsilon_1$. Il s'ensuit que $\varepsilon' \sqsubseteq \varepsilon$, car

$$\varepsilon_1 \sqcup \varepsilon_2 = (\varepsilon'_1 \sqcup \varepsilon_1) \sqcup \varepsilon_2 = \varepsilon'_1 \sqcup (\varepsilon_2 \sqcup \varepsilon_1) = \varepsilon'_1 \sqcup (\varepsilon_2 \sqcup \varepsilon_2) \sqcup \varepsilon_1 = (\varepsilon'_1 \sqcup \varepsilon_2) \sqcup (\varepsilon_1 \sqcup \varepsilon_2).$$

Deuxièmement, donnons la forme exacte du nouvel environnement Σ'. Pour cela, considérons non pas l'expression e mais le rédex correspondant Notons-le e_r. Si e_r correspond à l'allocation d'un nouvel enregistrement, on pose $\Sigma' \triangleq \Sigma[\ell \mapsto \rho]$. Si e_r est une affectation parallèle, alors notons σ_A la relation bijective correspondante et posons

$$\forall \ell.\, \Sigma'(\ell) \triangleq \begin{cases} \rho & \text{si } \langle \Sigma(\ell), \rho \rangle \in \sigma_A, \\ \Sigma(\ell) & \text{sinon.} \end{cases}$$

Enfin, dans tous les autres cas on pose $\Sigma' \triangleq \Sigma$, ce qui est cohérent avec le fait que si le rédex ne correspond à ni une allocation, ni une affectation parallèle, le pas d'évaluation ne change pas l'état mémoire. Nous savons maintenant comment typer les adresses libres de e' qui étaient accessibles auparavant dans l'état de mémoire μ : ou bien une telle adresse ℓ ne fait pas partie des modifications et alors son typage ne change pas, ou bien son contenu a été modifié mais, là encore, son typage ne change que lorsque dans la région $\Sigma(\ell)$ est distincte de son image par σ_A. Le fait que $\Sigma(\ell)$ se trouve à gauche dans $\langle \Sigma(\ell), \rho \rangle \in \sigma_A$ est important pour la préservation du type ; alors que ℓ est la nouvelle valeur associée à un champs affecté donné, c'est la région ρ de l'ancienne valeur du champs qui devient le nouveau type de l'adresse ℓ.

Jusqu'au présent nous avons défini ε, l'effet sur-approximant les effets de bord de l'expression évaluée e et l'effet ε' sur-approximant une partie des effets de bord du résultat d'évaluation e' (ignorons pour l'instant l'ensemble φ'' dans l'énoncé de préservation). Or, démontrer la préservation du typage et de l'injectivité du nouvel environnement Σ' vis-à-vis de la nouvelle configuration $(\mu' \cdot e')$, nécessite en quelque sorte de résoudre le « problème du cadre » (en anglais “frame problem” [21]), ce qui dans notre cas implique de fournir une description statique adéquate de ce qui a changé et de ce qui n'a pas changé dans l'état mémoire μ' par rapport à l'état mémoire μ. Pour cela, commençons par définir l'effet « réalisé » de e.

Définition 3.3.3 (Effet réalisé ε_0)**.** On définit l'effet réalisé de e, noté ε_0 ou $(\omega_0 \cdot \varphi_0)$, par récurrence sur la dérivation du pas de réduction :
– si e se réduit par la règle E-Alloc ou la règle E-Assign, on pose $\varepsilon_0 = \varepsilon$;
– si e est une liaison locale `let` $x = e_1$ `in` e_2 qui se réduit par la règle E-Ctx, on prend pour l'effet réalisé de e l'effet réalisé qui est donné par l'hypothèse de récurrence sur la réduction de e_1 ;
– dans tous les autres cas, le pas d'évaluation correspond à une réduction en tête qui ne modifie pas l'état mémoire et on pose alors $\varepsilon_0 = \bot$.

Remarquons au passage que l'on n'a pas de relation $\varepsilon = \varepsilon_0 \sqcup \varepsilon'$. Pour s'en persuader, il suffit de considérer une conditionnelle dont les branches ne sont pas pures. En revanche, à l'instar de l'effet ε', l'effet réalisé ε_0 est un sous-effet de ε :

Lemma 3.3.3. $\varepsilon_0 \sqsubseteq \varepsilon$.

Démonstration. Par récurrence sur la dérivation du pas d'évaluation $\mu \cdot e \longrightarrow \mu' \cdot e'$. Si e est une liaison locale $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ qui se réduit par la règle E-Ctx, par hypothèse de récurrence sur la dérivation de e_1 on obtient $\varepsilon_0 \sqsubseteq \varepsilon_1$. Ainsi $\varepsilon_1 \sqcup \varepsilon_2 = (\varepsilon_0 \sqcup \varepsilon_1) \sqcup \varepsilon_2 = \varepsilon_0 \sqcup (\varepsilon_1 \sqcup \varepsilon_2)$ par définition de $\sqsubseteq$ et l'associativité de $\sqcup$. Tous les autres cas sont immédiats. □

Le fait que $\varepsilon_0 \sqsubseteq \varepsilon$ implique que si une adresse mémoire $\ell \in \operatorname{dom} \mu$ apparaît dans $\mathcal{F}_\ell(e')$, alors la région $\Sigma'(\ell)$ est valide vis-à-vis de l'effet ε_0 et égale à $\Sigma(\ell)$:

Lemma 3.3.4. $\forall \ell.\, \ell \in \operatorname{dom} \mu \cap \mathcal{F}_\ell(e') \implies \Sigma'(\ell) = \Sigma(\ell) \wedge \varepsilon_0 \triangleright \Sigma'(\ell)$.

Démonstration. Remarquons d'abord que $\varepsilon_0 \triangleright \Sigma(\ell)$. En effet, si e correspond à une allocation ou à une affectation, le résultat est immédiat puisque $\operatorname{dom} \mu \cap \mathcal{F}_\ell(e') = \varnothing$. Si e est une liaison locale $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ avec e_1 réductible, alors notons ε_1 l'effet de l'expression e_1. Puisque e est bien typé, d'après les prémisses de la règle T-Let, nous avons $\varepsilon_1 \triangleright \Sigma(\ell)$ pour toute adresse $\ell \in \mathcal{F}_\ell(e_2)$. Or, par définition, l'effet ε_0 pour e et le même que pour e_1. D'après le lemme 3.3.3 on a alors $\varepsilon_0 \sqsubseteq \varepsilon_1$ et d'après le lemme 3.2.2 nous avons $\varepsilon_0 \triangleright \Sigma(\ell)$. Enfin, dans tous les autres cas ε_0 est vide, donc $\varepsilon_0 \triangleright \Sigma(\ell)$ est trivialement vrai.

D'autre part, quelle que soit l'adresse $\ell \in \operatorname{dom} \mu \cap \mathcal{F}_\ell(e')$, si $\Sigma(\ell) \neq \Sigma'(\ell)$, le rédex e_r correspond nécessairement à une affectation parallèle. Notons $\Phi(A)$ l'ensemble des régions invalidées qui est donné par le typage du rédex e_r. Puisque $\Sigma(\ell) \neq \Sigma'(\ell)$, les deux régions appartiennent à l'ensemble $\Phi(A)$ et donc à φ_0, puisque $\Phi(A) \subseteq \varphi_0$. Or, on vient de montrer $\varepsilon_0 \triangleright \Sigma(\ell)$. D'après le lemme 3.2.1, on a alors $\Sigma(\ell) \notin \varphi_0$, ce qui est absurde. On a donc bien $\Sigma(\ell) = \Sigma'(\ell)$ ce qui permet de conclure que $\varepsilon_0 \triangleright \Sigma'(\ell)$. □

Lemma 3.3.5 (Lemme du cadre)**.** Soit ℓ une adresse mémoire dans $\mathcal{F}_\ell(e')$. Soit π un chemin tel que $\mu(\ell.\pi)$ est défini et pour tout préfixe propre $\bar{\pi} \prec \pi$, $\Sigma(\ell).\bar{\pi} \notin \omega_0$. Dans ce cas, $\Sigma(\ell).\pi \notin \varphi_0$ et $\mu'(\ell.\pi) = \mu(\ell.\pi)$.

On peut représenter l'énoncé ci-dessus avec le dessin dans la figure 3.10 qui montre

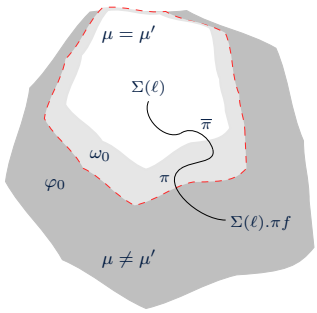


FIGURE 3.10 – Lemme du cadre

que tous les changements observables de l'état mémoire ainsi que toutes les régions invalidées sont « couverts » par l'ensemble des régions modifiées ω_0 de l'effet réalisé ε_0.

Démonstration. Par récurrence sur π. Si $\pi = \epsilon$, alors on a $\Sigma(\ell).\epsilon = \Sigma(\ell) \notin \varphi_0$. Par ailleurs, $\mu'(\ell.\epsilon) = \mu(\ell.\epsilon) = \ell$, puisque les règles de sémantique ne suppriment jamais des adresses mémoire de l'état mémoire μ. Sinon, considérons π égal à un chemin non vide $\pi' f$ tel que $\mu(\ell.\pi)$ soit défini et pour tout $\bar{\pi} \prec \pi$, on ait $\Sigma(\ell).\bar{\pi} \notin \omega_0$. Puisque l'on a $\varepsilon_0 \triangleright \Sigma(\ell)$ et comme aucune région qui se trouve sur le chemin depuis $\Sigma(\ell)$ jusqu'au type $\Sigma(\ell).\pi$ n'est pas dans ω_0, on en déduit que $\Sigma(\ell).\pi$ n'est pas dans φ_0. D'autre part, par hypothèse de récurrence sur π', on a $\mu'(\ell.\pi') = \mu(\ell.\pi')$. Par conséquent, le pas de réduction n'a pas modifié le contenu du champs f de l'adresse mémoire $\mu(\ell.\pi')$, car sinon la région $\Sigma(\mu(\ell.\pi')) = \Sigma(\ell).\pi'$ ferait nécessairement ou bien dans ω_0, ou bien dans φ_0. On conclut donc que $\mu'(\ell.\pi) = \mu(\ell.\pi)$. □

Nous sommes enfin prêts pour attaquer le lemme de préservation. Nous démontrons les trois propriétés de préservation (typage de l'expression, typage et injectivité de l'état mémoire) par récurrence sur la dérivation du pas de réduction avec l'analyse par cas sur la dernière règle de sémantique appliquée.

Cas de la règle E-Ctx. Supposons que e est de la forme $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$. Donc, e' est égale à $\texttt{let}\ x = e_1'\ \texttt{in}\ e_2$ où $\mu \cdot e_1 \longrightarrow \mu' \cdot e_1'$. Comme $\mathcal{F}_\ell(e_2) \subseteq \operatorname{dom} \mu$, d'après le lemme 3.3.4, pour toute $\ell \in \mathcal{F}_\ell(e_2)$, on a $\Sigma'(\ell) = \Sigma(\ell)$.

Préservation du typage de l'expression. Par hypothèse de récurrence sur la dérivation $\mu \cdot e_1 \longrightarrow \mu' \cdot e_1'$, on a $\Gamma \cdot \Sigma' \vdash e_1' : \tau_1 \cdot \varepsilon_1' \sqcup (\varnothing \cdot \varphi'')$, où τ_1 est le type de e_1 et où $\varphi'' \cap \Sigma'(\operatorname{dom} \mu') = \varnothing$. Par ailleurs, le rédex de e coïncide avec celui de e_1 et donc l'environnement Σ' défini en fonction de e_1 est le même que celui défini en fonction de $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$.

Notons τ_2 et ε_2 respectivement le type et l'effet de e_2 dans la dérivation du typage de $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$. Puisque pour toute adresse mémoire $\ell \in \mathcal{F}_\ell(e_2)$, Σ' coïncide avec Σ, on déduit que le jugement du typage $\Gamma[x \mapsto \tau_1] \cdot \Sigma' \vdash e_2 : \tau_2 \cdot \varepsilon_2$ est dérivable.

Montrons que pour toute adresse mémoire $\ell \in \mathcal{F}_\ell(e_2)$ on a $\varepsilon_1' \sqcup (\varnothing \cdot \varphi'') \triangleright \Sigma'(\ell)$. D'une part, on a $\varepsilon_1 \triangleright \Sigma(\ell)$ par hypothèse et puisque $\Sigma'(\ell) = \Sigma(\ell)$, on obtien $\varepsilon_1 \triangleright \Sigma'(\ell)$. Puisque $\varepsilon_1' \sqsubseteq \varepsilon_1$, le lemme 3.2.4 donne $\varepsilon_1' \triangleright \Sigma'(\ell)$. D'autre part, puisque $\varphi'' \cap \Sigma'(\operatorname{dom} \mu') = \varnothing$, on en déduit que $(\varnothing \cdot \varphi'') \triangleright \Sigma'(\ell)$. Or, e étant clos, l'ensemble $\mathcal{F}_v(e_2) \setminus \{x\}$ est vide. Le lemme 3.2.2 nous donne alors la troisième prémisse de la règle T-Let pour l'expression e'. Finalement, d'après le lemme 3.2.3, nous avons $\varepsilon_1' \sqcup (\varnothing \cdot \varphi'') \sqcup \varepsilon_2 = \varepsilon' \sqcup (\varnothing \cdot \varphi'')$, ce qui permet de conclure que $\Gamma \cdot \Sigma' \vdash e' : \tau \cdot \varepsilon' \sqcup (\varnothing \cdot \varphi'')$.

Préservation du typage de l'état mémoire. L'hypothèse de récurrence sur le pas de réduction de e_1 nous donne $\Sigma' \models \mu' \cdot e_1'$. Considérons une adresse mémoire ℓ dans e_2 et un chemin π quelconque. Montrons que soit $\Sigma'(\ell).\pi = \Sigma'(\mu'(\ell.\pi))$ ou bien ni $\Sigma'(\ell).\pi$, ni $\Sigma'(\mu'(\ell.\pi))$ ne sont définis.

Notons π' le plus grand préfixe du chemin π tel que $\mu(\ell.\pi')$ soit défini et pour tout préfixe $\bar{\pi} \prec \pi'$, on ait $\Sigma(\ell).\bar{\pi} \notin \omega_0$. Vérifions d'abord que de tels préfixes existent. En effet, l'hypothèse $\Sigma \models \mu \cdot e$ garantit que $\ell \in \operatorname{dom} \mu$, ce qui revient à dire que $\mu(\ell.\epsilon)$ est défini. On peut donc appliquer le lemme 3.3.5 à l'adresse ℓ et au chemin π', ce qui donne $\mu'(\ell.\pi') = \mu(\ell.\pi')$ et $\Sigma(\ell).\pi' \notin \varphi_0$. Puisque $\Sigma(\ell).\pi' = \Sigma(\mu(\ell.\pi'))$, on en déduit

que $\Sigma(\mu(\ell.\pi')) = \Sigma'(\mu(\ell.\pi'))$. D'autre part, d'après le lemme 3.3.4, $\Sigma'(\ell) = \Sigma(\ell)$. Ainsi obtient-on que $\Sigma'(\ell).\pi' = \Sigma(\ell).\pi' = \Sigma(\mu(\ell.\pi')) = \Sigma'(\mu(\ell.\pi')) = \Sigma'(\mu'(\ell.\pi'))$.

Notons $\pi = \pi'\pi''$. Si $\pi' = \pi$ (π'' est donc égal à ϵ), alors, d'après ce qui précède, on a montré que $\Sigma'(\ell).\pi = \Sigma'(\mu'(\ell.\pi))$. Supposons donc que $\pi' \prec \pi$. On a deux cas à considérer.

Supposons d'abord que pour tout chemin $\bar{\pi}$ tel que $\pi' \prec \bar{\pi} \preceq \pi$, $\mu(\ell.\bar{\pi})$ n'est pas défini. Dans ce cas-là, $\Sigma'(\ell).\pi$ n'est pas défini en vertu de l'égalité $\Sigma'(\ell).\pi = \Sigma(\ell).\pi = \Sigma(\mu(\ell.\pi))$ et $\Sigma'(\mu'(\ell.\pi))$ n'est pas défini en vertu de l'égalité $\mu'(\ell.\pi') = \mu(\ell.\pi')$.

Sinon, nécessairement $\Sigma(\ell).\pi' \in \omega_0$. Notons par ℓ' l'adresse mémoire $\mu(\ell.\pi')$. L'ensemble ω_0 n'étant pas vide, on en déduit que le rédex est nécessairement une affectation parallèle. Nous pouvons donc raisonner sur la relation σ_A correspondante. En particulier, on sait que $\ell' \in \operatorname{dom} A$ car $\Sigma(\ell') = \Sigma(\ell).\pi' \in \omega_0$. Deux cas sont possibles selon que $\Sigma(\ell').\pi''$ soit défini ou non.

Si $\Sigma(\ell').\pi''$ est défini, alors nécessairement $\langle A(\Sigma(\ell'), \pi''), \Sigma(\ell').\pi'' \rangle \in \sigma_A$. D'après la définition de Σ', on a l'égalité $\Sigma'(\mu'(\ell'.\pi'')) = \Sigma(\ell').\pi''$. Or, $\Sigma(\ell').\pi'' = \Sigma(\mu(\ell.\pi')).\pi'' = \Sigma(\ell).\pi = \Sigma'(\ell).\pi$, ce qui permet de conclure.

Enfin, si $\Sigma(\ell').\pi''$ n'est pas défini, alors $\Sigma'(\ell).\pi$ n'est pas défini en vertu de l'égalité $\Sigma'(\ell).\pi = \Sigma(\ell).\pi = \Sigma(\ell').\pi''$. Par ailleurs, $\mu(\ell'.\pi'')$ n'est pas défini non plus puisque μ est bien typé par hypothèse et $\Sigma(\ell').\pi''$ n'est pas défini. Or, dans l'affectation parallèle, les types des champs modifiés sont structurellement égaux aux types de nouvelles valeurs attribuées. Il s'ensuit que ℓ' dans μ (avant le pas de réduction) admet exactement les mêmes chemins que dans μ' (après le pas de réduction) : $\mu'(\ell'.\pi'') = \mu'(\mu(\ell.\pi').\pi'') = \mu'(\mu'(\ell.\pi').\pi'') = \mu'(\ell.\pi)$ est n'est donc pas défini et $\Sigma'(\mu'(\ell.\pi))$ n'est pas défini non plus, ce qui permet de conclure.

Préservation de l'injectivité de l'environnement du typage. Soient ℓ_1 et ℓ_2 deux adresses mémoire dans e'. Considérons deux chemins π_1 et π_2 arbitraires tels que $\Sigma'(\ell_1).\pi_1$ et $\Sigma'(\ell_2).\pi_2$ dénotent la même région. Montrons que $\mu'(\ell_1.\pi_1) = \mu'(\ell_2.\pi_2)$. Trois cas se présentent selon que ℓ_1 et ℓ_2 appartiennent ou non à $\operatorname{dom}\mu$.

Si aucune des deux adresses n'appartient au domaine de μ, cela signifie qu'elles ont été allouées lors du pas d'évaluation et qu'elles apparaissent toutes les deux dans e'_1. Par conséquent, l'hypothèse de récurrence sur $\mu \cdot e_1 \longrightarrow \mu' \cdot e'_1$ permet de conclure.

Supposons maintenant que seulement l'une des deux adresses appartient au domaine du μ. Sans perte de généralité, supposons que $\ell_1 \notin \operatorname{dom}\mu$ et que $\ell_2 \in \operatorname{dom}\mu$. Cela signifie que le rédex est l'allocation d'un nouvel enregistrement. Par conséquent, l'effet réalisé ε_0 est égal $(\omega_0 \cdot \varphi_0) = (\varnothing \cdot \{\Sigma'(\ell_1)\})$. Par ailleurs, la relation σ_A est nécessairement vide, donc $\Sigma'(\ell_2).\pi_2 = \Sigma(\ell_2).\pi_2$ par construction de Σ'. On en déduit donc que $\Sigma(\ell_2).\pi_2$ est défini et puisque l'état mémoire μ est bien typé, $\mu(\ell_2.\pi_2)$ est également défini. Par ailleurs, puisque $\Sigma(\ell_2.\pi_2) \notin \omega_0$, on peut donc appliquer le lemme 3.3.5, ce qui donne $\mu'(\ell_2.\pi_2) = \mu(\ell_2.\pi_2)$. Il s'ensuit que le chemin π_1 ne peut pas être vide, car s'il en était autrement, l'égalité $\mu'(\ell_1).\epsilon = \mu'(\ell_2.\pi_2)$, que l'on cherche à établir, serait en contradiction avec l'hypothèse que $\ell_1 \notin \operatorname{dom}\mu$. Le chemin π_1 est donc non vide. Notons $\pi_1 = f\pi'$ et notons ℓ' l'adresse mémoire $\mu'(\ell_1.f)$. Le diagramme

dans la figure 3.11 montre que $\mu(\ell'.\pi') = \mu'(\ell'.\pi')$. En effet, puisque le rédex est une allocation, l'adresse ℓ' est nécessairement dans $\mathcal{F}_\ell(e_1)$ et appartient donc au domaine de μ. Par ailleurs, σ_A étant vide, on a $\Sigma'(\ell') = \Sigma(\ell')$. Or, $\Sigma(\ell').\pi' = \Sigma(\mu(\ell'.\pi'))$ puisque μ est bien typé et $\Sigma'(\ell').\pi' = \Sigma'(\mu'(\ell_1.f\pi')) = \Sigma'(\ell_1).\pi_1$ car on a déjà établi que μ' était bien typé aussi. De plus, $\Sigma'(\ell_1).\pi_1 = \Sigma'(\ell_2).\pi_2 = \Sigma(\ell_2).\pi_2$. Il en résulte

$$\begin{array}{ccccc}
 & & \overset{\Sigma' \models \mu' \cdot e'}{} & & \overset{\text{hyp.}}{} \\
\Sigma'(\ell').\pi' & = & \Sigma'(\ell_1).\pi_1 & = & \Sigma'(\ell_2).\pi_2 \\
\| \ \sigma_A = \varnothing & & & & \sigma_A = \varnothing \ \| \\
\Sigma(\ell').\pi' & & & & \Sigma(\ell_2).\pi_2 \\
\| \ \Sigma \models \mu \cdot e & & & & \Sigma \models \mu \cdot e \ \| \\
\Sigma(\mu(\ell'.\pi')) & & \overset{\text{conclusion}}{=} & & \Sigma(\mu(\ell_2.\pi_2))
\end{array}$$

FIGURE 3.11 – $\mu(\ell'.\pi') = \mu'(\ell'.\pi')$

que $\Sigma(\mu(\ell'.\pi')) = \Sigma(\mu(\ell_2.\pi_2))$, ce qui, en vertu de l'injectivité de l'environnement Σ, donne l'égalité $\mu(\ell'.\pi') = \mu(\ell_2.\pi_2)$. Or, l'évaluation d'une allocation ne modifie pas les adresses mémoire du μ, donc $\mu(\ell'.\pi') = \mu'(\ell'.\pi')$ et $\mu'(\ell_2.\pi_2) = \mu(\ell_2.\pi_2)$, ce qui permet de conclure.

Sinon, ℓ_1 et ℓ_2 sont dans $\operatorname{dom}\mu$. Comme $\Sigma'(\ell_1) = \Sigma(\ell_1)$ et $\Sigma'(\ell_2) = \Sigma(\ell_2)$, on a l'égalité $\mu(\ell_1.\pi_1) = \mu(\ell_2.\pi_2)$ en vertu de l'injectivité de Σ : en effet, si $\ell_1 \in \mathcal{F}_\ell(e)$, l'injectivité s'applique immédiatement et si $\ell_1 \notin \mathcal{F}_\ell(e)$, ℓ_1 ne peut apparaître dans e' que si le rédex est de la forme $\ell_0.f$ pour une adresse et un champs f tels que $\mu(\ell_0.f) = \ell_1$. Dans ce cas-là, on a $\mu(\ell_1.\pi_1) = \mu(\ell_2.\pi_2)$, en vertu de l'injectivité de Σ et du bon typage de μ.

Soit alors π'_1 le plus long préfixe de π_1 tel que pour tout sous-chemin $\bar{\pi} \prec \pi'_1$, on ait $\Sigma(\ell_1).\bar{\pi} \notin \omega_0$. De même, soit π'_2 le plus long préfixe de π_2 tel que pour tout sous-chemin $\bar{\pi} \prec \pi'_2$, on ait $\Sigma(\ell_2).\bar{\pi} \notin \omega_0$. On peut donc appliquer le lemme 3.3.5, ce qui donne $\Sigma(\ell_1).\pi'_1 \notin \varphi_0$, $\Sigma(\ell_2).\pi'_2 \notin \varphi_0$, $\mu'(\ell_1.\pi'_1) = \mu(\ell_1.\pi'_1)$ et $\mu'(\ell_2.\pi'_2) = \mu(\ell_2.\pi'_2)$.

Si $\pi'_1 = \pi_1$ et $\pi'_2 = \pi_2$, alors on conclut immédiatement que $\mu'(\ell_1.\pi_1) = \mu'(\ell_2.\pi_2)$. Sinon, au moins l'une des régions $\Sigma(\ell_1).\pi'_1$ et $\Sigma(\ell_2).\pi'_2$ appartient à ω_0. Or, cela signifie que le rédex est une affectation parallèle et on peut alors raisonner sur la relation σ_A.

Sans perte de généralité, supposons que $\Sigma(\ell_1).\pi'_1 \in \omega_0$. Notons ℓ'_1 l'adresse mémoire $\mu(\ell_1.\pi'_1)$ et posons $\pi_1 = \pi'_1\pi''_1$. Nécessairement, la relation σ_A contient la paire $\langle A(\Sigma(\ell'_1), \pi''_1), \Sigma(\ell'_1).\pi''_1\rangle$ où $A(\Sigma(\ell'_1), \pi''_1) = \Sigma(\mu'(\ell'_1.\pi''_1))$ et où, par définition du Σ', $\Sigma(\ell'_1).\pi''_1 = \Sigma'(\mu'(\ell'_1.\pi''_1))$, On a alors trois cas à considérer selon que $\pi'_2 = \pi_2$ ou pas.

Le premier cas correspond à la situation où $\pi'_2 = \pi_2$ avec $A(\Sigma(\ell'_1), \pi''_1) \neq \Sigma(\ell'_1).\pi''_1$. Montrons que ce cas-là est absurde. En effet, $A(\Sigma(\ell'_1), \pi''_1) \neq \Sigma(\ell'_1).\pi''_1$ entraîne que $\Sigma(\ell'_1).\pi''_1 \in \varphi_0$ en vertu de la définition du $\Phi(A)$. Or, d'une part, $\Sigma(\ell'_1).\pi''_1 = \Sigma(\ell_1).\pi_1$

et, d'autre part, l'on a vu que $\Sigma(\ell_1).\pi_1 = \Sigma(\ell_2).\pi_2$. On en déduit que nécessairement $\Sigma(\ell_2).\pi_2 \in \varphi_0$, ce qui est absurde puisque l'on a supposé que $\pi'_2 = \pi_2$ et $\Sigma(\ell_2).\pi'_2 \notin \varphi_0$.

Le second cas correspond à la situation où $\pi'_2 = \pi_2$ avec $A(\Sigma(\ell'_1), \pi''_1) = \Sigma(\ell'_1).\pi''_1$. On a donc $\Sigma(\mu'(\ell'_1.\pi''_1)) = A(\Sigma(\ell'_1), \pi''_1) = \Sigma(\mu(\ell'_1.\pi''_1))$. Par conséquent, l'injectivité du Σ donne $\mu'(\ell'_1.\pi''_1) = \mu(\ell'_1, \pi''_1)$. Celà permet de conclure, puisque l'on a :

$$\mu'(\ell_1.\pi_1) = \mu'(\ell'_1.\pi''_1) = \mu(\ell'_1, \pi''_1) = \mu(\ell_1, \pi_1) = \mu(\ell_2, \pi_2) = \mu'(\ell_2, \pi_2).$$

Sinon, on a nécessairement $\pi'_2 \prec \pi_2$ avec $\Sigma(\ell_2).\pi'_2 \in \omega_0$. Notons ℓ'_2 l'adresse $\mu(\ell_2.\pi'_2)$ et posons $\pi_2 = \pi'_2\pi''_2$. La relation σ_A contient donc la paire $\langle A(\Sigma(\ell'_2), \pi''_2), \Sigma(\ell'_2).\pi''_2\rangle$. Or, σ_A étant bijective, l'égalité $\Sigma(\ell'_1).\pi''_1 = \Sigma(\ell'_2).\pi''_2$ que l'on a établie plus haut, entraîne $\Sigma(\mu'(\ell'_1.\pi''_1)) = A(\Sigma(\ell'_1), \pi''_1) = A(\Sigma(\ell'_2), \pi''_2) = \Sigma(\mu'(\ell'_2.\pi''_2))$. Ainsi, les adresses $\mu'(\ell'_1.\pi''_1)$ et $\mu'(\ell'_2.\pi''_2)$, qui sont touts les deux accessibles dans l'état mémoire μ depuis le rédex de e, sont égales en vertu de l'injectivité de l'environnement Σ, ce qui permet de conclure que $\mu'(\ell_1.\pi_1) = \mu'(\ell_2, \pi_2)$.

Cas de la règle E-Alloc. Soit e une allocation d'enregistrement $\{f_1 = v_1, \ldots, f_n = v_n\}$, et e' est une adresse mémoire ℓ fraîche.

Préservation du typage de l'expression. Notons ρ la région de e. Par construction, $\varepsilon' = \bot$ et $\Sigma'(\ell) = \Sigma[\ell \mapsto \rho](\ell) = \rho$. En prenant $\varphi'' = \varnothing$, on conclut directement que e' est bien typée par la règle T-Loc.

Préservation du typage de l'état mémoire. Soit π un chemin quelconque. Si $\pi = \epsilon$, alors $\mu'(\ell.\epsilon) = \ell$ et on a immédiatement $\Sigma'(\ell).\epsilon = \Sigma'(\mu'(\ell.\epsilon))$. Supposons donc que $\pi \neq \epsilon$. Notons $\pi = f\pi'$. Si f est différent de tous les champs $f_1, \ldots, f_n$, alors ni $\mu'(\ell.\pi)$, ni $\Sigma'(\ell).\pi = \rho.\pi$ ne sont définis. Supposons donc que $f = f_i$ pour $i \in [1..n]$. Si la valeur correspondante v_i est une constante c du type scalaire ν, alors ni $\mu'(\ell.f\pi')$, ni $\Sigma'(\ell).f\pi'$ ne sont définis que si $\pi' = \epsilon$. Dans ce cas-là, on a le résultat, puisque $\Sigma'(\ell).f = \nu = \Sigma'(c) = \Sigma'(\mu'(\ell.f))$. Sinon, notons ℓ' l'adresse mémoire qui correspond à la valeur v_i. Ou bien $\Sigma(\ell').\pi'$ est égal à $\Sigma(\mu(\ell'.\pi'))$, ou bien les deux expressions de type ne sont pas définis en vertu de l'hypothèse $\Sigma \models \mu \cdot e$. Cela permet alors de conclure que $\Sigma'(\ell).\pi = \rho.f\pi' = \Sigma(\ell').\pi' = \Sigma(\mu(\ell'.\pi')) = \Sigma'(\mu(\ell'.\pi')) = \Sigma'(\mu'(\ell'.\pi')) = \Sigma'(\mu'(\ell.\pi))$.

Préservation de l'injectivité de l'environnement du typage. Étant donné que $\mathcal{F}_\ell(e') = \{\ell\}$, il suffit de considérer deux chemins distincts π_1 et π_2 tels que $\rho.\pi_1$ et $\rho.\pi_2$ dénotent la même région. D'abord, remarquons qu'aucun des deux chemins ne peut être vide. En effet, la région ρ ne peut pas être égale à l'une de ses sous-régions stricte. Nécessairement donc π_1 est égal à $f_i\pi'_1$ et π_2 est égal à $f_j.\pi'_2$ avec i et j dans $[1..n]$. Réécrivons l'hypothèse $\rho.\pi_1 = \rho.\pi_2$ en $(\rho.f_i).\pi'_1 = (\rho.f_j).\pi'_2$. Puisque l'on a supposé que $\rho.\pi_1$ et $\rho.\pi_2$ étaient des régions, il en est de même pour $\rho.f_i$ et $\rho.f_j$. Il s'ensuit que la valeurs correspondantes v_i et v_j sont toutes les deux adresses mémoires dans e. Notons $v_i = \ell_i$ et $v_j = \ell_j$. Comme $\Sigma(\ell_i) = \rho.f_i = \rho.f_j = \Sigma(\ell_j)$, l'injectivité du Σ nous donne $\ell_i = \ell_j$. D'autre part, l'état mémoire μ étant bien typé, on a $\Sigma(\ell_i).\pi'_1 = \Sigma(\mu(\ell_i.\pi'_1))$ et $\Sigma(\ell_j).\pi'_2 = \Sigma(\mu(\ell_j.\pi'_2))$, ce qui, en vertu de l'injectivité du

Σ donne $\mu(\ell_i.\pi'_1) = \mu(\ell_j.\pi'_2)$. Finalement, puisque $\mu'(\ell.\pi_1) = \mu'(\ell_i.\pi'_1) = \mu(\ell_i.\pi'_1)$ et de même $\mu'(\ell.\pi_2) = \mu'(\ell_j.\pi'_2) = \mu(\ell_j.\pi'_2)$, on conclut que $\mu'(\ell.\pi_1) = \mu'(\ell.\pi_2)$.

Cas de la règle E-Assign. Soit e une affectation parallèle : on a donc $e' = \texttt{()}$ avec $\tau = \texttt{Unit}$, $\varepsilon' = \bot$ et $\Sigma' = \Sigma$. En prenant $\varphi'' = \varnothing$, on peut conclure que toutes les trois propriétés sont trivialement préservées, puisque $\mathcal{F}_\ell(e') = \varnothing$.

Cas de la règle E-δ. Supposons que e est un appel de fonction $p(v_1, \ldots, v_n)$. Notons $p(x_1, \ldots, x_n) \mapsto \hat{e}$ la définition de la fonction p. L'expression e' est donc égale à $\hat{e}[x_i/v_i \ ^{i\in[1,\ldots,n]}]$. Par ailleurs, $\mu' = \mu$, puisque le pas de réduction ne modifie pas l'état mémoire et $\Sigma' = \Sigma$ par construction et $\mathcal{F}_\ell(e') = \mathcal{F}_\ell(\hat{e}) = \varnothing$ par hypothèse. On a donc immédiatement $\Sigma' \models \mu' \cdot e'$ et $\Sigma' \ltimes \mu' \cdot e'$. Montrons maintenant la préservation du typage pour e'. Notons $\hat{\tau}_1 \times \ldots \times \hat{\tau}_n \to \hat{\tau} \cdot (\hat{\omega} \cdot \hat{\varphi})$ la signature de type de p et posons $\Gamma' = \Gamma[x_i \mapsto \hat{\tau}_i \ ^{i\in[1,\ldots,n]}]$. Par hypothèse, on a $\Gamma' \cdot \Sigma \vdash \hat{e} : \hat{\tau} \cdot (\hat{\omega} \cdot \hat{\varphi} \cup \hat{\varphi}'')$ où les ensembles $\hat{\varphi}''$ et $\mathcal{R}(\hat{\tau}_1, \ldots, \hat{\tau}_n, \hat{\tau})$ sont disjoints. Montrons d'abord le lemme suivant :

Lemma 3.3.6. Quelle que soit la région ρ apparaissant dans l'arbre de dérivation du jugement $\Gamma' \cdot \Sigma \vdash \hat{e} : \hat{\tau} \cdot (\hat{\omega} \cdot \hat{\varphi} \cup \hat{\varphi}'')$, ou bien $\rho \in \hat{\varphi}''$ ou bien $\rho \in \mathcal{R}(\hat{\tau}_1, \ldots, \hat{\tau}_n, \hat{\tau})$.

Démonstration. Notons $\mathcal{T}$ l'arbre de dérivation de $\hat{e}$. Soit ρ une région apparaissant dans $\mathcal{T}$. Supposons par absurde que ρ n'apparaît ni dans $\hat{\varphi}''$ ni dans $\mathcal{R}(\hat{\tau}_1, \ldots, \hat{\tau}_n, \hat{\tau})$. Ordonnons l'ensemble des nœuds de l'arbre $\mathcal{T}$ par l'ordre lexicographique qui correspond au parcours préfixe de $\mathcal{T}$: un nœud sera donc plus petit qu'un autre lorsque le premier se trouve soit sur une branche distincte plus à gauche, soit plus haut sur la même branche. Considérons alors le plus petit nœud $\mathcal{N}$ de $\mathcal{T}$ qui contient une occurrence de ρ. Cela veut dire que le nœud $\mathcal{N}$ est de la forme

$$\frac{\ldots \qquad \ldots}{\Gamma'[y_j \mapsto \tau_j \ ^{j\in[1,\ldots,m]}] \cdot \Sigma \vdash \tilde{e} : \tilde{\tau} \cdot (\tilde{\omega} \cdot \tilde{\varphi})}(\text{T-}\ldots)$$

où ρ apparaît soit dans $\tilde{\tau}$, soit dans $(\tilde{\omega} \cdot \tilde{\varphi})$. Raisonnons par cas selon la règle du typage associée à $\mathcal{N}$. Le cas T-Loc exclu puisque $\mathcal{F}_\ell(\hat{e}) = \varnothing$. Le cas T-Cst est trivialement exclu. Les cas T-Let, T-Field, T-Assign, T-If sont impossibles en vertu de la minimalité de $\mathcal{N}$. Il nous reste donc trois cas à considérer.

Supposons que $\mathcal{N}$ a été obtenu par la règle T-Alloc. Alors, par minimalité de $\mathcal{N}$, on a nécessairement $\rho = \tilde{\tau}$ et $\tilde{\varphi} = \{\rho\}$. Or, quelle que soit la règle de typage, les régions invalidées dans les prémisses apparaissent toujours dans l'effet de la conclusion. Il s'ensuit que $\rho \in \hat{\varphi} \cup \hat{\varphi}''$. Or, on a supposé que ρ n'apparaissait ni dans $\hat{\varphi}''$, ni dans $\mathcal{R}(\hat{\tau}_1, \ldots, \hat{\tau}_n, \hat{\tau})$ et comme $\hat{\varphi} \subseteq \mathcal{R}(\hat{\tau}_1, \ldots, \hat{\tau}_n)$, on en déduit une contradiction.

Supposons que $\mathcal{N}$ a été obtenu par la règle T-Var. La conclusion du nœud $\mathcal{N}$ est donc de la forme $\Gamma'[y_j \mapsto \tau_j \ ^{j\in[1,\ldots,m]}] \cdot \Sigma \vdash z : \rho \cdot \bot$ où z est soit l'un des paramètres formels, soit l'une des variables appartenant à $\{y_1, \ldots, y_m\}$. Si $z = x_i$, alors $\rho = \hat{\tau}_i \in \mathcal{R}(\hat{\tau}_1, \ldots, \hat{\tau}_n)$, ce qui est absurde. Supposons donc que $z = y_k$ avec $k \in [1, \ldots, m]$. Puisque $\mathcal{F}_v(\hat{e}) \subseteq \{x_1, \ldots, x_n\}$, y_k est forcément l'occurrence d'une variable liée de l'expression $\hat{e}$. Autrement dit, l'arbre $\mathcal{T}$ est de la forme

$$\dfrac{\dfrac{\dfrac{\dots}{\Gamma'' \cdot \Sigma \vdash \tilde{e}_1 : \rho \cdot \tilde{\varepsilon}'_1}(\mathcal{M}) \qquad \dfrac{\Gamma'[y_j \mapsto \tau_j \ ^{j\in[1,\dots,m]}](y_k) = \rho}{\Gamma'[y_j \mapsto \tau_j \ ^{j\in[1,\dots,m]}] \cdot \Sigma \vdash y_k : \rho \cdot \bot}(\mathcal{N}) \quad \cdots}{\Gamma'' \cdot \Sigma \vdash \texttt{let}\ y_k = \tilde{e}_1\ \texttt{in}\ \tilde{e}_2 : \tilde{\tau}' \cdot \tilde{\varepsilon}'}(\text{T-Let}) \quad \cdots}{\Gamma' \cdot \Sigma \vdash \hat{e} : \hat{\tau} \cdot (\hat{\omega} \cdot \hat{\varphi} \cup \hat{\varphi}'')}(\text{T-}\dots)$$

où $\Gamma'' = \Gamma'[y_j \mapsto \tau_j \ ^{j\in[1,\dots,k-1,k+1,m]}]$. Or, le nœud $\mathcal{M}$ qui contient ρ est strictement plus petit que $\mathcal{N}$, ce qui est absurde.

Enfin, supposons que $\mathcal{N}$ a été obtenu par la règle T-Call. $\mathcal{N}$ est donc de la forme

$$\dfrac{p : \tilde{\tau}_1 \times \ldots \times \tilde{\tau}_k \to \tilde{\tau} \cdot (\tilde{\omega} \cdot \tilde{\varphi}) \qquad \Gamma'[y_j \mapsto \tau_j \ ^{j\in[1,\dots,m]}] \cdot \Sigma \vdash a_i : \theta(\tilde{\tau}_i) \cdot \bot}{\Gamma'[y_j \mapsto \tau_j \ ^{j\in[1,\dots,m]}] \cdot \Sigma \vdash p'(a_1, \dots, a_k) : \theta(\tilde{\tau}) \cdot (\theta(\tilde{\omega}) \cdot \theta(\tilde{\varphi}))}\ (\text{T-Call})$$

Puisque θ est une substitution de régions, on sait que $\theta(\tilde{\omega}) \subseteq \mathcal{R}(\theta(\tilde{\tau}_1), \dots, \theta(\tilde{\tau}_k))$ et $\mathcal{R}(\theta(\tilde{\tau})) \setminus \mathcal{R}(\theta(\tilde{\tau}_1), \dots, \theta(\tilde{\tau}_k)) \subseteq \theta(\tilde{\varphi})$. Par minimalité de $\mathcal{N}$, on sait que ρ est nécessairement dans $\theta(\tilde{\varphi})$. Or, dans toute règle de typage, les régions invalidées dans les prémisses apparaissent dans l'effet de la conclusion : comme dans le cas de T-Alloc, on en déduit donc une contradiction. Ainsi, quelle que soit la forme de l'arbre de dérivation $\mathcal{T}$, ou bien $\rho \in \hat{\varphi}''$ ou bien $\rho \in \mathcal{R}(\hat{\tau}_1, \dots, \hat{\tau}_n, \hat{\tau})$. □

Revenons maintenant à l'expression e. Le jugement du typage de e est de la forme $\Gamma \cdot \Sigma \vdash p(v_1, \dots, v_n) : \theta(\hat{\tau}) \cdot (\omega \cdot \theta(\varphi))$ où θ une substitution de régions et où le type de chaque argument, $\Sigma(v_i)$ est égal à $\theta(\hat{\tau}_i)$. Soit alors θ' une substitution de régions qui prolonge θ de telle manière que l'image de chaque région dans φ'' soit une région fraîche. Grâce au lemme 3.3.6, on peut appliquer θ' sur chaque nœud de l'arbre de dérivation du jugement $\Gamma[x_i \mapsto \hat{\tau}_i \ ^{i\in[1,\dots,n]}] \cdot \Sigma \vdash \hat{e} : \hat{\tau} \cdot (\hat{\omega} \cdot \hat{\varphi} \cup \hat{\varphi}'')$, de sorte que le jugement du typage $\Gamma[x_i \mapsto \theta'(\hat{\tau}_i) \ ^{i\in[1,\dots,n]}] \cdot \Sigma \vdash \hat{e} : \theta'(\hat{\tau}) \cdot (\theta'(\hat{\omega}) \cdot \theta'(\hat{\varphi}) \cup \theta'(\hat{\varphi}''))$ soit dérivable. Posons alors $\varphi'' = \theta'(\hat{\varphi}'')$. Puisque toutes les régions dans φ'' sont fraîches, on a bien $(\omega \cdot \varphi \cup \varphi'') = (\omega \cdot \varphi) \sqcup (\varnothing \cdot \varphi'')$ et $\varphi'' \cap \Sigma'(\operatorname{dom} \mu') = \varnothing$. Enfin, chacun des arguments $x_1, \dots, x_n$ ayant maintenant le même type que les valeurs correspondantes $v_1, \dots, v_n$ passées en arguments, lemme de substitution, qui est un résultat standard ([25, p.106]), garantît que le jugement $\Gamma \cdot \Sigma' \vdash \hat{e}[x_i/v_i \ ^{i\in\{1,\dots,n\}}] : \tau \cdot \varepsilon \sqcup (\varnothing \cdot \varphi'')$ est dérivable, ce qui permet de conclure.

Dans tous les autre cas, le pas d'évaluation ne produit pas d'effets, donc $\varepsilon_0 = \bot$, l'état mémoire ne change pas et $\Sigma' = \Sigma$. De plus, toutes les adresses mémoire dans e' sont accessibles dans e. On a donc immédiatement la préservation du typage de l'état mémoire μ' et l'injectivité de l'environnement Σ'. Posons $\varphi'' = \varnothing$. On obtient la préservation du typage immédiatement pour les règles E-Op et E-Field. Pour la règle E-ζ, comme pour l'appel de fonction, le lemme de substitution permet de conclure. Enfin, lorsque e est une conditionnelle qui se réduit soit avec E-True soit avec E-False, on a par construction $\varepsilon' \sqsubseteq \varepsilon$, ce qui permet de conclure. □

3.4 Implémentation

Dans cette section nous discutons brièvement certaines fonctionnalités de WhyML que n'avons pas abordées dans notre formalisme.

Inférence de types. Comme nous l'avons remarqué dans l'introduction, en Why3 les annotations de types écrites par l'utilisateur ne mentionnent jamais les régions, celles-ci étant inférées automatiquement. Cette inférence se fait essentiellement dans l'esprit du système Hindley-Milner [16] et l'algorithme W [11]. L'aspect vernaculaire de l'inférence en Why3 réside dans le fait que le parcours de l'arbre de syntaxe abstraite du programme s'effectue en deux passes. D'abord le programme est traversé, en utilisant l'algorithme d'inférence classique qui attribue à chaque expression un type, en ignorant les identités des régions comme si elles étaient toutes distinctes. Le programme est ensuite traversé une deuxième fois, en faisant une unification de régions là où est elle nécessaire.

Polymorphisme de types. En Why3 les types, y compris les régions, peuvent également contenir des variables de types. On peut par exemple avoir la définition de type suivante :

```
type t 'a = { mutable f: 'a; mutable g: ref int }
```

La question se pose alors de savoir ce qui se passe si l'on instancie les variables de types par des régions. On peut alors remarquer que l'instanciation des variables de types peut amener à introduire de nouveaux alias. Par exemple, pour le type `t` ci-dessus, il suffit d'écrire

```
let x = ref 0 in
let y = { f = x; g = x } in
...
```

pour obtenir une valeur du type `t (ref int)` avec un alias entre les champs `f` et `g`. Une manière simple d'étendre notre formalisme à des variables de type serait de modifier la règle d'appel de fonctions de la façon suivante. Reprenons la notation $\tau_1 \times \ldots \times \tau_n \to \tau_0 \cdot \varepsilon$ pour la signature de la fonction appelée et $\theta(\tau_i)$ $(0 \leq i \leq n)$ pour les types du résultat et des arguments factuels. Il suffit alors de vérifier que, lorsqu'une variable de type dans τ_i est instanciée par un type $\tau \in \theta(\tau_i)$, l'ensemble $\mathcal{R}(\tau)$ (c'est-à-dire les régions qui apparaissent dans τ) soit disjoint de l'ensemble des régions qui instancient les régions dans $\mathcal{R}(\tau_1, \ldots, \tau_n, \tau_0)$.

Types algébriques. Pour simplifier la présentation, notre formalisme suppose que tous les champs d'un enregistrement sont mutables. Bien entendu, en pratique, certains champs peuvent être immuables. D'autre part, les types algébriques non-récursifs

peuvent être vus comme une généralisation des types enregistrements. Il ne serait pas difficile d'étendre RegML à des types algébriques tels que `opt_int = Some int | None` ou même `opt_ref = Some (ref int) | None` où les régions apparaissent à une profondeur bornée. Il serait même possible d'étendre notre formalisme avec des types récursifs tant qu'ils sont purs, c'est-à-dire, qu'ils ne contiennent aucune région. En revanche, comme nous l'avons expliqué dans l'introduction, les types algébriques récursifs non purs ne peuvent pas être formalisés dans notre système de type qui exige que toutes les régions doivent être connues statiquement.

Code fantôme. Dans le chapitre précédent, nous avons insisté sur le fait que le statut fantôme concerne les variables et les expressions de programme et non pas les types. Cela reste également vrai pour les régions : en Why3, il n'y a pas de « régions fantômes » : ce sont des champs, qu'ils soient mutables ou pas, qui peuvent être déclarés comme fantômes ou réels. On peut alors se poser la question de savoir comment le contrôle de non-interférence et le contrôle statique des alias interagissent. La réponse est que la seule contrainte que les champs fantômes imposent, pour que le principe de non-interférence soit respecté, consiste à vérifier que si une région ρ est accessible depuis une variable réelle à travers un chemin qui est lui-même « réel », c'est-à-dire qu'il passe seulement par des champs réels pour atteindre ρ, alors toute modification de la région ρ doit être faite uniquement depuis les parties réelles du programme.

Génération des conditions de vérification. Comme nous l'avons remarqué au début du chapitre, les obligations de preuve générées par Why3 restent simples au sens qu'elles correspondent à des formules du premier ordre et ne font pas intervenir un modèle mémoire explicite. Or, puisque WhyML autorise l'aliasing dans les programmes, la règle de Hoare pour l'opération d'affectation ne peut pas être utilisée telle quelle. Ce problème est résolu en Why3 grâce au contrôle statique des alias (en particulier grâce au fait que le typage et l'injectivité de la mémoire sont préservés par l'exécution). En simplifiant beaucoup, la règle logique pour l'affectation ressemblerait à un triplet

$$\{\, [E/x, E/y, \ldots, E/z]P \,\}\ x := E\ \{\, P \,\}$$

où $x, y, \ldots, z$ sont tous les alias de x. Cependant, la forme exacte d'une telle règle serait plus complexe, notamment à cause des enregistrements imbriqués. Concrètement, en se servant des informations inférées par le typage sur la relation d'aliasing entre les variables de programmes, le calcul de plus faibles préconditions de Why3 effectue une mise à jour des régions dépendant des régions affectées. Cette mise à jour consiste, dans un premier temps, à introduire de nouvelles variables pour tous les alias d'une région dont le contenu est affecté, puis, dans un deuxième temps, à reconstruire les enregistrements depuis lesquels cette région est accessible.

3.5 Travaux connexes

Comme nous venons de voir, notre approche du contrôle statique des alias s'appuie sur un système de types avec des régions et des effets. Cette méthodologie prend ses racines dans les travaux de Baker [2], Lucassen et Gifford [20], et ceux de Tofte and Talpin [30] sur la gestion de la mémoire à base de régions. Dans tous ces travaux, les régions sont utilisées pour garantir que deux pointeurs qui appartiennent à des régions différentes ne sont jamais aliasés. En revanche, lorsque deux pointeurs se trouvent à l'intérieur d'une même région, le système ne permet de savoir s'ils sont aliasés ou non. Dit autrement, les régions dans ces travaux peuvent être vues comme des classes d'équivalence de pointeurs pour une relation d'« aliasing possible » dont la nature peut être connue statiquement. Contrairement à ces approches, dans notre système, une région dénote non pas un ensemble d'adresses mémoire, mais toujours une et une seule adresse. Cet invariant nous permet de décrire statiquement la forme exacte de la mémoire utilisée dans le programme, puisque deux noms symboliques sont alors aliasés si et seulement s'ils sont typés par la même région. Une telle représentation de la mémoire est d'ailleurs proche de la manière dont l'identité des pointeurs est encodée dans les approches comme *alias types* [28] et *typed regions* [22]. Néanmoins, ces deux approches s'appuient sur la notion « strong update », c'est-à-dire sur la possibilité de remplacer, au fil de l'exécution, le type d'une expression donnée par un autre type. Or, dans le cas des *alias types*, ce choix impose des limitations sur le flot de contrôle et, dans le cas des *typed regions*, ce choix introduit la lourde machinerie des types dépendants.

L'*analyse de la forme de la mémoire* (en anglais *Shape analysis* [27, 15]) fournit des techniques similaires à la nôtre pour inférer automatiquement les invariants de représentation de la mémoire. Par exemple, l'analyse « must-alias » [13, 29] s'appuie sur la traçabilité de chemins d'accès dans les enregistrements. En particulier, une adresse mémoire y est caractérisée par un ensemble de chemins d'accès qui amènent jusqu'à elle depuis les noms symboliques déclarés dans le programme. Cependant, dans le contexte de la vérification déductive qui est le nôtre, nous ne pouvons pas nous permettre de sur-approximer la relation d'aliasing. Ainsi, lorsque le typage rencontre une situation qui pourrait amener à une telle sur-approximation (ce que nous avons appelé *un conflit d'aliasing* au début de ce chapitre), il restreint l'utilisation des noms symboliques de telle sorte que la représentation statique de la mémoire par le typage reste exacte. Comme nous l'avons vu, le prix à payer est que notre approche ne permet pas actuellement de typer certaines structures de données telles que des listes chaînes et ou des tableaux de références.

Dans le cadre de la programmation orientée-objet, l'*ownership* [8, 24, 12] et des approches statiques similaires comme *islands* [18], *balloon types* [1], et *universe types* [23] fournissent une méthodologie qui vise un contrôle d'aliasing et une encapsulation forte des données mutables. Ainsi, dans le paradigme « owners-as-dominators », le système

exige que tous les accès *externes* à la représentation *interne* d'un objet se fassent à travers l'interface de celui qui possède les droits d'écriture. De ce point de vue, les contraintes de validité que notre système de type engendre dans la règle pour les expressions de la forme `let` $x = e_1$ `in` e_2, peuvent être considérées comme le fait que dans l'expression e_2, les « possesseurs » des régions invalidées de l'expression e_1 sont exactement les régions qui peuvent y accéder en passant par une région figurant dans l'effet d'écriture de e_1.

Cependant, il ne s'agit ici que d'une analogie, car notre approche est en fait orthogonale à celle d'*ownership*. En effet, le but principal de celle-ci n'est pas d'obtenir une représentation exacte de la mémoire, mais plutôt de garantir une notion forte d'encapsulation en s'appuyant sur les annotations de types de l'utilisateur pour déterminer quelles composantes d'un objet sont accessibles à des objets extérieurs.

Il est intéressant de noter que notre effet d'invalidation présente quelques similarités avec le concept de *variable unique* (en anglais *unique variable*) [31, 18, 3]. Une *variable unique* correspond soit à un pointeur nul, soit se réfère à un objet qui n'est aliasé avec personne d'autre. Dans notre cas, la règle du typage pour la création d'une nouvelle adresse mémoire à partir d'un enregistrement produit un effet qui interdit tous les noms existants qui font référence à la région de l'adresse allouée. Lorsque la création d'un enregistrement est liée à une variable x, notre système de types fait donc de x une variable unique. Ceci est d'ailleurs très similaire au concept d'« *enfouissement d'alias* » (en anglais « alias burying ») de Boyland [5] où, lorsqu'un champ annoté comme unique est lu, tous les alias existants sont considérés comme invalides et ne doivent plus jamais être utilisés dans la suite du programme.

Enfin, notons que notre manière d'utiliser les effets non seulement pour décrire la forme de la mémoire, mais également pour imposer des contraintes d'accessibilité, lui donne un certain goût des *permissions* [10, 7]. Cependant, plutôt que de transmettre des « jetons de permissions » entre les producteurs et des consommateurs, notre effet d'invalidation peut être vu comme une révocation totale de permission. Par ailleurs, comme dans le cas des « alias types », les systèmes à base des permissions font le plus souvent usage de « strong updates ».

Bibliographie

[1] Paulo Sérgio Almeida, *Balloon types : Controlling sharing of state in data types*, Proceedings ECOOP'97, LNCS, vol. 1241, Springer, June 1997, pp. 32–59.

[2] Henry G. Baker, *Unify and conquer*, LISP and Functional Programming, 1990, pp. 218–226.

[3] Henry G. Baker, *"Use-once" variables and linear objects : Storage management, reflection and multi-threading*, SIGPLAN Not. **30** (1995), no. 1, 45–52.

[4] Josh Berdine and Peter W. O'Hearn, *Strong update, disposal, and encapsulation in bunched typing*, Electr. Notes Theor. Comput. Sci. **158** (2006), 81–98.

[5] John Boyland, *Alias burying : unique variables without destructive reads*, j-SPE **31** (2001), no. 6, 533–553.

[6] Rod Burstall, *Some techniques for proving correctness of programs which alter data structures*, Machine Intelligence **7** (1972), 23–50.

[7] Arthur Charguéraud and François Pottier, *Functional translation of a calculus of capabilities*, ACM SIGPLAN International Conference on Functional Programming (ICFP), September 2008, pp. 213–224.

[8] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad, *Ownership types : A survey*, in Clarke et al. [9], pp. 15–58.

[9] David Clarke, James Noble, and Tobias Wrigstad (eds.), *Aliasing in object-oriented programming : Types, analysis and verification*, Lecture Notes in Computer Science, vol. 7850, Springer, 2013.

[10] Karl Crary, David Walker, and Greg Morrisett, *Typed memory management in a calculus of capabilities*, ACM Symposium on Principles of Programming Languages (POPL), ACM Press, 1999, pp. 262–275.

[11] Luis Damas and Robin Milner, *Principal type-schemes for functional programs*, POPL '82 : Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), ACM Press, 1982, pp. 207–212.

[12] Werner Dietl and Peter Müller, *Object ownership in program verification*, in Clarke et al. [9], pp. 289–318.

[13] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay, *Effective typestate verification in the presence of aliasing*, ACM Trans. Softw. Eng. Methodol. **17** (2008), no. 2, 9 :1–9 :34.

[14] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen, *The essence of compiling with continuations*, SIGPLAN Not. **28** (1993), no. 6, 237–247.

[15] Brian Hackett and Radu Rugina, *Region-based shape analysis with tracked locations*, SIGPLAN Not. **40** (2005), no. 1, 310–323.

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

[17] C. A. R. Hoare, *An axiomatic basis for computer programming*, Communications of the ACM **12** (1969), no. 10, 576–580 and 583.

[18] John Hogg, *Islands : Aliasing protection in object-oriented languages*, SIGPLAN Not. **26** (1991), no. 11, 271–285.

[19] Ioannis T. Kassios, *Dynamic frames : Support for framing, dependencies and sharing without restrictions*, 14th International Symposium on Formal Methods (FM'06) (Hamilton, Canada) (Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, eds.), Lecture Notes in Computer Science, vol. 4085, 2006, pp. 268–283.

[20] J. M. Lucassen and D. K. Gifford, *Polymorphic effect systems*, Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (New York, NY, USA), POPL '88, ACM, 1988, pp. 47–57.

[21] John Mccarthy and Patrick J. Hayes, *Some philosophical problems from the standpoint of artificial intelligence*, Machine Intelligence, Edinburgh University Press, 1969, pp. 463–502.

[22] Stefan Monnier, *Typed regions*, Second workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE'2004) (Venice, Italy), January 2004.

[23] Peter Müller and Arsenii Rudich, *Ownership transfer in universe types*, ACM SIGPLAN conference on Object-oriented programming systems and applications (OOPSLA), ACM, 2007, pp. 461–478.

[24] Alan Mycroft and Janina Voigt, *Notions of aliasing and ownership*, in Clarke et al. [9], pp. 59–83.

[25] Benjamin C. Pierce, *Types and programming languages*, MIT Press, 2002.

[26] J. C. Reynolds, *Separation logic : a logic for shared mutable data structures*, 17th Annual IEEE Symposium on Logic in Computer Science, IEEE Comp. Soc. Press, 2002.

[27] Mooly Sagiv, Thomas W. Reps, and Reinhard Wilhelm, *Parametric shape analysis via 3-valued logic*, **24** (2002), no. 3, 217–298.

[28] Frederick Smith, David Walker, and J. Gregory Morrisett, *Alias types*, Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings (Gert Smolka, ed.), Lecture Notes in Computer Science, vol. 1782, Springer, 2000, pp. 366–381.

[29] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav, *Alias analysis for object-oriented programs*, in Clarke et al. [9], pp. 196–232.

[30] Mads Tofte and Jean-Pierre Talpin, *Region-based memory management*, Information and Computation (1997).

[31] Philip Wadler, *Linear types can change the world !*, Programming Concepts and Methods, North, 1990.

[32] J. B. Wells, *The essence of principal typings*, pp. 913–925, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[33] Andrew K. Wright and Matthias Felleisen, *A syntactic approach to type soundness*, Information and Computation **115** (1992), 38–94.

4 Raffinement des données

4.1 Introduction

Ce chapitre aborde certains aspects de la vérification des programmes conçus d'une manière *modulaire*, notamment le cas où la vérification de tels programmes peut être elle-même modulaire. Avant de rentrer dans les détails techniques et d'expliquer notre contribution, rappelons d'abord ce que l'on entend par le développement modulaire et l'obtention de programmes corrects par construction, deux notions qui nous intéressent ici particulièrement.

4.1.1 Développement modulaire des programmes

D'une manière générale, le développement modulaire consiste à découper la réalisation d'un programme en un ensemble de composantes, appelées ici *modules*, dans le but d'implémenter d'une manière plus ou moins autonome des parties du programme indépendantes du point de vue logique. L'idée est que chaque module n'a pas besoin de connaître le contenu exact des autres modules qu'il utilise, mais seulement leur description abstraite, par exemple en terme d'opérations fournies. Une telle description spécifiant le comportement d'un module peut être donnée dans une *interface* qui permet ainsi aux modules de communiquer entre eux. Enfin, lorsque chacun des modules est écrit, ils peuvent être assemblés afin de produire le code exécutable.

Un outil classique du développement modulaire est l'abstraction des structures des données. L'idée clé est que pour être capable d'utiliser une structure de données, il n'est pas nécessaire de connaître la manière exacte dont elle est réalisée. Pour cela, il suffit souvent de connaître l'ensemble des opérations permettant de manipuler les objets du type de données abstrait. Par exemple, lors d'un parcours de graphe en profondeur, on se sert souvent d'un ensemble pour savoir quels sont les sommets déjà visités. La structure d'ensemble peut être réalisée par un tableau, un arbre binaire de recherche ou encore une table de hachage. Néanmoins, lorsqu'on implémente un parcours de graphe en profondeur, on n'a pas besoin de savoir laquelle de ces trois réalisations va être choisie. La seule chose dont on a besoin, c'est que l'on puisse effectuer les opérations d'ajout et de recherche d'un élément dans un ensemble, ce qui peut être spécifié dans l'interface.

L'abstraction d'une structure de données fait donc apparaître trois composantes conceptuellement distinctes : l'*interface* qui spécifie le type de données, mais ne donne pas son implémentation ; le code *client* qui utilise la structure de données en se servant uniquement de l'interface ; et l'*implémentation* qui réalise la structure de donnée conformément à la manière dont elle est spécifiée. Une telle manière de procéder permet, entre autres, de réaliser d'abord la structure de données d'une manière naïve pour pouvoir tester le programme client, puis de la remplacer par une représentation plus efficace.

En contrepartie d'une telle flexibilité, l'interface établit entre le client et l'implémentation une *barrière d'abstraction* qu'aucun des deux ne doit briser. Du côté du client, cela signifie qu'en utilisant l'abstraction de données, il ne doit jamais faire aucune hypothèse au delà de ce qui est révélé par l'interface, afin qu'il puisse être compatible avec toute implémentation que l'on juge conforme à l'interface. Du côté de l'implémentation, cela signifie qu'elle doit prendre en compte tout et être conforme à tout ce qui est énoncé dans l'interface afin qu'elle puisse être compatible avec tout client que l'on juge conforme à l'interface (qui utilise l'interface correctement).

Vérifier que le client et l'interface respectent la barrière d'abstraction est nécessaire pour que les opérations décrites dans l'interface puissent effectivement être exécutées et aient le comportement attendu lorsque le module d'implémentation et le module du client sont assemblés. L'avantage de développer les modules séparément est que l'on peut aussi effectuer ces vérifications d'une manière indépendante pour chacun des modules. C'est d'ailleurs ce qui se passe pendant la phase de *la compilation séparée* dans les langages de programmation typés où le typage vérifie non seulement que le code de chacun des modules est bien typé, mais également qu'il utilise correctement le code des modules extérieurs.

4.1.2 Obtention de programmes corrects par construction

Il est légitime maintenant de se poser la question de savoir si la spécification et la vérification des programmes conçus d'une manière modulaire peuvent être elle-mêmes modulaires. Bien entendu, on peut toujours attendre que toutes les composantes soient d'abord implémentées, obtenir le code assemblé et montrer ensuite la correction d'un seul coup, mais cette façon de faire peut être fastidieuse et inefficace. Elle peut être fastidieuse, car, dans le code assemblé, le raisonnement sur l'implémentation des structure de données abstraites et le raisonnement sur l'implémentation de l'algorithme qui les utilise vont s'entremêler, alors qu'*a priori* ces deux raisonnements sont orthogonaux ; elle peut être inefficace notamment lorsque l'on décide de changer la réalisation de l'abstraction, car il faudra alors reprouver entièrement la nouvelle version du code assemblé. Or, si le code client et l'implémentation respectent chacun la barrière d'abstraction établie par l'interface, il devient envisageable de spécifier et de prouver chacun des deux modules séparément, de sorte que lorsque le code est assemblé, il n'y a pas besoin de prouver sa correction : il sera *correct par construction*.

Il est important de noter que la dichotomie interface/module est utile pour comprendre les principes derrière l'abstraction, notamment pour dégager les notions de barrière d'abstraction et d'obtention du code correct par construction. Cependant, ce n'est pas la seule manière de rendre le développement et la vérification modulaires. On peut également adopter une approche où il n'y a qu'un seul concept de module qui permet à la fois d'exprimer l'abstraction et de l'utiliser/implémenter. Dans cette approche, certaines parties d'un module peuvent être implémentées, alors que d'autres parties restent abstraites. La réalisation de l'abstraction est alors effectuée lorsque l'on remplace un module par un autre dans le but de rendre certaines parties du code plus concrètes (par exemple, en rendant les calculs plus déterministes, en implémentant des structures des données, etc.) Dans ce cas-ci, on dit que le nouveau module est le *raffinement* de l'autre module si toutes les propriétés postulées ou établies au niveau abstrait sont préservées dans le nouveau module. Un tel développement de programmes, à partir de la spécification jusqu'à l'obtention du code exécutable, effectué par des couches successives préservant chacune la correction des niveaux au-dessus, est appelé souvent lui-même le *raffinement*. Parmi les approches qui mettent en œuvre le raffinement on peut citer Z [8], VDM [3] et la méthode B [1]. Notons que l'on peut distinguer deux types de raffinement : le *raffinement algorithmique*, qui consiste à dériver le code exécutable à partir de la spécification (par exemple, dériver le corps d'une procédure à partir de son contrat), et le *raffinement des structures de données* qui consiste à transformer une structure de données abstraite par une structure de donnée plus concrète. En terminologie de la programmation orientée objet, le raffinement de données peut être expliqué à l'aide du principe de substitution de Liksov [7] :

> *Si $Q(x)$ est une propriété démontrable pour tout objet x de type T, alors la propriété $Q(y)$ doit être vraie pour tout objet y de type S tel que S est un sous-type T.*

Bien que ce principe soit énoncé en termes de la programmation orientée objet, il exprime l'idée que le raffinement des structures de données est une notion sémantique. C'est ce second type de raffinement que nous allons aborder dans ce chapitre. Plus spécifiquement, nous nous concentrons sur le raffinement des types enregistrements en Why3. L'intérêt de l'étudier dans cette thèse est double.

Premièrement, des structures de données abstraites peuvent aisément être modélisées par des types enregistrements. Cela est dû au fait que pour prouver le code client manipulant des données d'un type abstrait, il est souvent nécessaire de raisonner sur le modèle logique correspondant. Or, doter un type enregistrement d'un champ fantôme permet justement d'y enchâsser ce modèle logique. Ceci est d'autant plus pratique qu'un tel champ modèle peut être déclaré mutable pour refléter les changements du modèle lorsque le contenu des données change.

Deuxièmement, les types enregistrements peuvent être vus comme des régions au sens du chapitre précédent. Comme nous allons le voir, le problème d'aliasing va réapparaître avec l'introduction de nouvelles composantes mutables dans des types enregistrements. Il sera utile de s'appuyer sur le formalisme que l'on a développé dans

le chapitre précédent, pour tenir compte du contrôle statique des alias lors du raffinement.

4.1.3 Raffinement des types enregistrements

Avant de rentrer dans les détails techniques et d'expliquer quelle est la contribution de ce chapitre vis-à-vis du raffinement des types enregistrements, esquissons l'idée en grandes lignes. En Why3 le développement modulaire est géré par un système de modules. Il n'y a pas de notion d'interface à proprement parler : la même notion de module sert à la fois pour exprimer, utiliser et implémenter l'abstraction. Un module peut contenir des définitions purement logiques tels que des prédicats inductifs ou des axiomes, ainsi que des déclarations de programme : des procédures, des types de données mutables, des exceptions, etc. Par exemple, pour implémenter l'algorithme du parcours d'un graphe [1] en profondeur, on peut définir le module `DFS` à l'intérieur duquel le module `MutableSet` est utilisé pour marquer les sommets visités :

```
module DFS
  use import Graph as G
  use import MutableSet as M

  let rec dfs (c: G.node) (visited: M.t) : unit =
    if c ≠ G.null && not M.contains c visited then begin
      M.add c visited;
      dfs (G.left c)  visited;
      dfs (G.right c) visited;
    end

  let traverse () : unit =
    let visited = M.create () in
    dfs G.root visited
end
```

Pour représenter l'interface de la structure d'ensemble mutable des nœuds d'un graphe, on pourrait envisager de déclarer le module `MutableSet` de la manière suivante :

```
module MutableSet
  use import Graph as G
  type t
  val create   ()                 : t
  val add      (x: G.node) (t: t) : unit
  val contains (x: G.node) (t: t) : bool
end
```

1. On considère ici un type de graphes très simple avec une seule racine `root` et des sommets de degré au plus deux.

Bien sûr, cela ne suffit pas pour raisonner sur l'utilisation des objets du type `t` : rien ne dit qu'il s'agit effectivement d'une structure d'ensemble mutable. Comme nous l'avons remarqué, la mutabilité des données est exprimée en Why3 à l'aide des types enregistrements et le fait qu'il s'agit d'une structure d'*ensemble* peut alors être formulé à l'aide d'un champ modèle. On commence donc par importer la théorie des ensembles mathématiques de la bibliothèque standard de Why3 :

```
module MutableSet
  use import Graph as G
  use import set.Fset as S
  ...
```

On définit ensuite le type `t` comme un type enregistrement (ignorons pour l'instant le mot `private` que nous expliquerons plus tard) :

```
  type t = private { ghost mutable repr: S.set G.node }
```

On peut maintenant expliciter le comportement des trois opérations `create`, `add` et `contains` :

```
  val create () : t
    ensures { S.is_empty result.repr }

  val add (x: G.node) (t: t) : unit
    writes  { t }
    ensures { t.repr = S.add x (old t.repr) }

  val contains (x: G.node) (t: t) : bool
    ensures { result ↔ S.mem x t.repr }
```

Notons que la spécification de l'opération `add` comporte une clause `writes { t }` qui indique que les composantes mutables de `t` (ici limitées au champ `repr`) sont susceptibles d'être modifiées par la fonction. Notons également que le champ `repr` est fantôme, ce qui est une façon de dire qu'il s'agit d'un champ modèle, introduit pour les besoins de la spécification. Rien n'empêche d'introduire également des composantes mutables réelles. Ainsi, on peut ajouter au type `t` un champ mutable `size` et exprimer à l'aide d'un invariant de type qu'il doit être égal au nombre d'éléments de l'ensemble `repr` :

```
  type t = private { ghost mutable repr: S.set G.node
                            mutable size: int        }
         invariant { size = S.cardinal repr }
```

Dans la définition du type `t` ci-dessus, le mot `private` joue un rôle important, car sa présence permet d'établir une barrière d'abstraction que le client est tenu de respecter : la création des objets d'un type privé, ainsi que la modification de leur contenu n'est possible qu'au travers des opérations fournies par le module où le type est défini. S'il en

était autrement, le code client pourrait briser la barrière d'abstraction. Par exemple, le code

```
let marked = { repr = Set.empty; size = 0 } in
...
```

n'est compatible avec aucun raffinement du type `t` qui introduirait un champ supplémentaire. De même, le code

```
let marked = create () in
marked.repr ← add x marked.repr;
()
```

casse l'invariant `size = S.cardinal repr`. La mise en place des types privés permet ainsi d'exprimer l'*encapsulation* de l'état interne des données mutables, un ingrédient nécessaire pour mettre en place le raffinement correctement.

Maintenant que l'on a introduit `DFS`, le module du client, et `MutableSet`, le module de l'« interface » des ensembles mutables, l'étape suivante consiste à raffiner la structure d'ensembles mutables. Pour faire cela, nous pouvons créer un autre module dans lequel le type `t` et les opérations `create`, `add` et `contains` sont redéfinis avec plus de précision.

En ce qui concerne la définition du type, on doit pouvoir fournir des champs supplémentaires. Cela peut servir notamment pour réaliser la structure des ensembles par un type concret, par exemple des tables de hachage avec chaînage. On commence le module d'implémentation de la façon suivante :

```
module MutableSetByHashtbl
  use import Graph as G
 (* ici on importe tous les autres modules et les theories:
    de la bibliotheque standard: ensembles, listes, tableaux, ...  *)

  function hash G.node : int
  axiom hash_nonneg: ∀k: G.node. 0 ≤ hash k

  function bucket (k: G.node) (n: int) : int = mod (hash k) n

  type t  = { ghost mutable repr: F.set G.node;
                    mutable size: int;
                    mutable buckets: array (list G.node) }
  invariant { size = S.cardinal repr ∧ les elements du tableau buckets
              sont exactement les elements de l'ensemble repr
              et sont dans les bons buckets }
  ...
end
```

Ici, le type `t` contient un nouveau champ mutable `buckets`. Notons que l'on a exprimé l'invariant de collage entre la représentation et le modèle (écrit ci-dessus en français

pour simplifier) comme un renforcement du l'invariant du type `t` dans l'interface. Notons aussi que l'on a enlevé le mot `private`. Cela indique au système qu'il s'agit effectivement d'une réalisation des ensembles mutables destinée à être assemblée avec le code client pour générer le code exécutable. Il peut sembler que l'on ait ouvert au client de `MutableSet` la possibilité de briser la barrière d'abstraction. Néanmoins, si l'on suppose que le code du client est assemblé tel qu'il a été écrit en premier lieu, la barrière d'abstraction reste effectivement intacte.

En ce qui concerne le raffinement des opérations `create`, `add` et `contains`, on doit pouvoir d'une part préciser leurs spécifications (en affaiblissant les préconditions, en renforçant les post-conditions si nécessaire) et, d'autre part, les implémenter. Les fonctions de création et d'appartenance peuvent être implémentées (car le type `t` n'est plus privé) de la manière suivante :

```
let create ()
  ensures { S.is_empty result.repr }
= { repr = S.empty; size = 0; buckets = Array.make 17 Nil }

let contains (x: G.node) (t: t) : bool
  ensures { result ↔ S.mem x t.repr }
= let i = bucket x t.size in
  List.mem x t.buckets[i]
```

En ce qui concerne l'opération d'ajout, supposons que l'on a écrit une fonction `resize` du type `t → unit` qui remplace le tableau stocké dans le champ `buckets` par un tableau deux fois plus grand et recopie tous les éléments de l'ancien tableau. On peut alors implémenter la fonction `add` de la façon suivante :

```
let add (x: G.node) (t: t) : unit
  ensures { t.repr = S.add x (old t.repr) }
= if t.size = t.buckets.length then resize t;
  let i = bucket x t.buckets.length in
  t.buckets[i] ← Cons x t.buckets[i];
  t.size ← t.size + 1;
  t.repr ← S.add x t.repr
```

Supposons maintenant que nous avons spécifié l'algorithme du parcours de graphe et montré sa correction. Dans ce cas, le but est d'assembler le module `DFS` avec le module d'implémentation `MutableSetByHashtbl` afin d'obtenir un code correct par construction. Comme nous l'avons expliqué, pour que le code assemblé soit correct, le système doit vérifier au préalable que la barrière d'abstraction établie par l'interface est respectée à la fois par le client et par l'implémentation. En ce qui concerne le client, cette vérification est déjà mise en place en Why3 : le système de types vérifie notamment que le client ne manipule des données d'un type privé qu'au travers des opérations fournies. Ce qui manque actuellement, c'est donc un mécanisme de vérification que le module d'implémentation est bien un raffinement valide du module d'interface, au

sens du principe de substitution de Liskov.

Une partie des vérifications à faire qui découlent de ce principe concerne exactement ce que l'on vient de faire dans l'exemple du module `MutableSetByHashtbl` : on doit s'assurer que l'implémentation contient au moins tous les symboles des types enregistrements, des champs et des procédures déclarés dans le module `MutableSet`, que le sens des contrats est préservé, etc. Mais il y a aussi une partie des vérifications qui concerne le problème d'aliasing et qui vient du fait que le raffinement introduit de nouvelles composantes mutables. Pour mieux expliquer ce problème, revenons aux notions des *régions* et du *contrôle statique des alias* que nous avons introduites dans le chapitre précédent.

4.1.4 Le problème d'aliasing

Comme nous l'avons expliqué, les types enregistrements et les régions sont la même chose. Nous pouvons d'ailleurs reformuler le principe de Liskov en termes de régions :

> *Si $Q(\rho)$ est une propriété démontrable pour toute région privée ρ, alors la propriété $Q(\rho')$ doit être vraie pour toute région ρ' telle que ρ' est un raffinement de la région ρ.*

Comme nous l'avons vu dans le chapitre précédent, le contrôle statique des alias fait partie des propriétés qui sont assurées par le typage de WhyML. Rappelons que cela signifie que pour toute paire de pointeurs qui apparaissent dans un programme bien typé, le système connaît statiquement si ces deux noms réfèrent à la même case mémoire ou pas. Le principe de Liksov implique donc que si le code client était bien typé en premier lieu, il doit rester bien typé après le raffinement, y compris en ce qui concerne le contrôle statique des alias. Or, le raffinement d'une région peut y introduire des champs supplémentaires contenant eux-mêmes de nouvelles régions. Si l'on ne met aucune restriction sur la relation entre ces régions introduites et les régions qui existaient déjà dans le code client avant le raffinement, il devient possible de briser la barrière d'abstraction et de mettre le principe de Liskov en défaut. Par exemple, si l'ensemble des régions introduites par le raffinement n'est pas disjoint de l'ensemble des régions déjà connues du client, on peut facilement casser le typage du code client. En effet, supposons que l'on dispose de deux opérations `newA ()` et `newB ()` qui créent respectivement deux régions distinctes ρ_1 et ρ_2. Dans le code ci-dessous, les types et les effets seront donc :

```
let x = newA () : ρ1 · (∅ · ρ1) in
let y = newB () : ρ2 · (∅ · ρ2) in
x
```

Maintenant, supposons que l'on raffine la région ρ_2 en introduisant un champ f et que $\rho_2.f = \rho_1$. Dans ce cas-ci, l'effet de l'appel `newB ()` devient $(\varnothing \cdot \{\rho_2, \rho_1\})$ où la région ρ_1 devient donc invalidée. Or, la variable `x` est justement de type ρ_1 et donc ne peut pas être utilisée : le code ci-dessus devient donc rejeté par le typage, alors qu'il

était bien typé auparavant. Ainsi, on doit imposer que les régions introduites par le raffinement doivent être disjointes des régions connues auparavant.

Mais la fraîcheur des régions introduites vis-à-vis des régions existantes ne suffit pas : il y a d'autres moyens, plus subtils, d'introduire des alias inconnus du client. Considérons par exemple le code client qui définit deux ensembles mutables que l'on modifie ensuite :

```
let bar (x: G.node) =
  let marked   = MutableSet.create () in
  let on_stack = MutableSet.create () in
  add x marked;
  add x on_stack;
```

Pour que ce code soit bien typé, il est nécessaire de supposer que les deux ensembles sont typés avec deux régions distinctes, disons ρ_1 et ρ_2. En effet, si ces régions étaient égales, la création de l'ensemble `on_stack` invaliderait l'utilisation de l'ensemble `marked`. Imaginons maintenant que l'on a choisi l'implémentation des ensembles mutables avec les tables de hachage, comme dans le module `MutableSetbyHashtbl`. Notons qu'*a priori*, rien n'empêche les régions ρ_1 et ρ_2 de partager le même tableau à l'intérieur du champ `buckets`. Or, même si ce tableau-là appartenait à une région ρ_3 fraîche, dès lors que l'on a l'égalité ρ_1`.data` $= \rho_2$`.data` (où les deux expressions dénotent la même région ρ_3), le code de la fonction `bar` ci-dessus devient nécessairement mal typé. En effet, dans l'implémentation de la fonction `add`, la première ligne

```
if t.size = t.buckets.length then resize t;
```

a pour effet de restreindre l'utilisation de la région ρ_3, puisque la fonction `resize` est susceptible de remplacer le tableau stocké dans `t.buckets` par un tableau frais. La région ρ_3 ne sera dorénavant accessible que depuis la région ρ_1. Par conséquent, la dernière ligne (`add x on_stack;`) dans le code ci-dessus devient mal typée, car nous avons supposé que $\rho_1 \neq \rho_2$.

D'une manière encore plus subtile, si l'on raffine deux régions équivalentes (c'est-à-dire qui ont la même structure d'aliasing, voir la définition $\rho_1 \equiv \rho_2$ dans section 3.2.1 du chapitre précédent) par deux régions qui ne le sont pas, il devient encore possible de mettre le principle de Liskov en défaut. Illustrons ce propos sur l'exemple suivant. Considérons une interface :

```
module GF
  type t
  val f: unit → t
  val g: t → unit
end
```

et le code client très simple qui l'utilise :

```
let () = g(f())
```

Maintenant, supposons que l'on raffine le type `t` par un type enregistrement avec deux composantes mutables :

```
type t = { mutable a: array int;
           mutable b: array int }
```

A priori, rien n'empêche de construire des données du type `t` correspondant à des régions structurellement égales mais non équivalentes, selon que les champs `a` et `b` sont aliasés ou non. Supposons donc que l'on donne aux fonctions `f` et `g` respectivement les signatures suivantes (ignorons ci-dessous les effets qui ne sont pas pertinents pour notre propos ici)

```
val f: unit → {a : ρa; b : ρa}r
val g: {a : ρa; b : ρb}r' → unit
```

où l'on suppose que les régions ρ_a et ρ_b sont distinctes. Or, puisque la règle du typage de l'appel (voir la section 3.2.4) impose que les régions des paramètres formels et les régions des arguments soient équivalentes, l'appel `g(f())` devient mal typé, alors qu'il était accepté par le typage en premier lieu.

Il est donc nécessaire de trouver des conditions suffisantes, mais pas trop restrictives, sur la manière dont les nouvelles régions sont introduites, pour garantir que le raffinement ne mette pas en défaut le principe de Liskov. Comme nous l'avons illustré avec les trois exemples ci-dessus, trouver ce genre de conditions n'est pas trivial. Par ailleurs, cette question peut être étudiée indépendamment des autres conditions de la validité du raffinement qui sont elles plus en rapport avec la préservation des propriétés logiques. La contribution de ce chapitre est de proposer de telles conditions et de montrer formellement leur validité. Concrètement, nous allons adapter le formalisme du chapitre précédent aux notions de régions privées, puis définir le raffinement des régions et des signatures de fonctions. Ensuite, nous montrerons que le raffinement ainsi défini préserve la relation du typage.

Insistons encore fois sur le fait que la préservation des notions logiques telles que les invariants de types et les contrats de fonctions doivent également être prises en compte pour montrer la validité du raffinement. Comme il s'agit de notions bien étudiées dans la littérature, nous ne les formalisons pas dans ce chapitre, en nous consacrant entièrement à la problématique des alias.

4.2 Extension de RegML avec des régions privées

La figure 4.1 reprend la grammaire des types de RegML, en introduisant une distinction entre régions *publiques* et régions *privées*. Les régions publiques, que l'on note $\{f : \tau, \ldots, f : \tau\}_r$, sont exactement les régions que l'on a présentées dans le chapitre précédent. Les régions privées, que l'on note $[f : \nu, \ldots, f : \nu]_r$, sont des régions dont tous les champs ont des types scalaires. Nous supposons par ailleurs que les noms des champs dans les régions privées sont disjoints des noms des champs dans les régions publiques. Ainsi, on ne peut pas avoir à la fois $\{f : \texttt{Int}\}_{r_1}$ et $[f : \texttt{Int}]_{r_2}$.

τ	::=	TYPES
	$\mid \nu$	*type scalaire*
	$\mid \rho$	*région*
ν	::=	TYPES SCALAIRES
	$\mid \texttt{Int} \mid \texttt{Bool} \mid \texttt{Unit}$	*types scalaires*
ρ	::=	RÉGIONS
	$\mid \{f : \tau, \ldots, f : \tau\}_r$	*région publique*
	$\mid [f : \nu, \ldots, f : \nu]_r$	*région privée*

FIGURE 4.1 – Types avec les régions publiques et privées.

Adaptons maintenant les règles du typage pour RegML (voir la figure 3.5, page 92) à la nouvelle grammaire des types avec les régions privées. Ceci est en fait extrêmement simple : le seul changement à faire consiste à expliciter le statut *public* des régions $\rho_1, \ldots, \rho_n$ dans la règle pour l'affectation parallèle :

$$\frac{\begin{array}{c}\Gamma \cdot \Sigma \vdash a_i : \rho_i \cdot \bot \qquad \rho_i \text{ publique} \qquad \Gamma \cdot \Sigma \vdash a'_{i,j} : \tau'_{i,j} \cdot \bot \qquad \rho_i.f_{i,j} \simeq \tau'_{i,j} \\ \rho_i \text{ deux à deux distincts} \qquad \forall i.\ f_{i,j} \text{ deux à deux distincts} \\ \varphi = \Phi(\rho_i.\{f_{i,j} \leftarrow \tau'_{i,j}\ {}^{j\in[1,\ldots,k_i]}\}^{i\in[1,\ldots,n]})\end{array}}{\Gamma \cdot \Sigma \vdash a_i.\{f_{i,j} \leftarrow a'_{i,j}\ {}^{j\in[1,\ldots,k_i]}\}^{i\in[1,\ldots,n]} : \texttt{Unit} \cdot (\{\rho_1, \ldots, \rho_n\} \cdot \varphi)}$$

Remarquons que les régions privées modélisent les types enregistrements privés que l'on a présentés dans l'introduction : on ne peut ni les utiliser pour construire explicitement un enregistrement ni modifier explicitement leurs champs. En revanche, accéder à un champ reste possible comme pour les régions publiques.

Sans le démonter formellement, notons aussi que l'introduction des régions privées ne met en défaut aucun résultat que nous avons établi dans le chapitre précédent.

Enfin, notons que pour exprimer l'abstraction par un type enregistrement privé, il suffit dans beaucoup de cas de l'instrumenter avec un champ modèle dont le type est un type logique, exactement comme nous l'avons illustré dans l'introduction. Un tel type enregistrement se modélise donc bien par une région privée dont tous les champs sont d'un type de base. Dans notre formalisme, les types de base comprennent des types d'entiers, de booléens et le type unit, mais, bien entendu, on pourrait considérer qu'un type logique, comme par exemple un type d'ensembles, fait également partie des types de base (décrits dans notre formalisme par la méta-variable ν). Pour cette raison-là, nous allons nous contenter de ne considérer dans ce chapitre que des régions privées « plates », c'est-à-dire dont toutes les composantes sont d'un type de base, comme c'est défini dans la figure 4.1 ci-dessus.

4.3 Raffinement

Maintenant que nous avons étendu le langage RegML avec des régions privées, nous pouvons définir le raffinement des types et des signatures de fonctions.

4.3.1 Raffinement des types

L'idée clé est que lors du raffinement, ou bien une région privée reste privée auquel cas elle peut comporter éventuellement de nouvelles composantes scalaires ; ou bien elle devient une région publique, au quel cas les nouvelles composantes peuvent contenir aussi bien des types scalaires que des sous-régions. C'est dans ce dernier cas que l'on doit faire attention à la manière dont ces nouvelles régions sont introduites, d'où la définition suivante :

Définition 4.3.1 (Raffinement des types)**.** Le *raffinement des types* est une fonction totale Ψ définie sur l'ensemble des types qui à chaque type τ dans son domaine renvoie un type $\Psi(\tau)$ défini selon la forme de τ :

$$
\begin{array}{lcl}
\nu & \overset{\Psi}{\longmapsto} & \nu \\
\{f_1 : \tau_1, \ldots, f_n : \tau_n\}_r & \overset{\Psi}{\longmapsto} & \{f_1 : \Psi(\tau_1), \ldots, f_n : \Psi(\tau_n)\}_r \\
[f_1 : \nu_1, \ldots, f_n : \nu_n]_r & \overset{\Psi}{\longmapsto} & \{f_1 : \nu_1, \ldots, f_n : \nu_n, f'_1 : \tau'_1, \ldots, f'_m : \tau'_m \ ^{m \geq 0}\}_r \\
[f_1 : \nu_1, \ldots, f_n : \nu_n]_r & \overset{\Psi}{\longmapsto} & [f_1 : \nu_1, \ldots, f_n : \nu_n, f'_1 : \nu'_1, \ldots, f'_m : \nu'_m \ ^{m \geq 0}]_r
\end{array}
$$

et qui vérifie les trois hypothèses suivantes :

(Fraîcheur)	Tous les champs et les indices créés par le raffinement sont frais.
(Séparation)	Quelles que soient les régions privées ρ_1 et ρ_2 distinctes, les ensembles de régions $\mathcal{R}(\Psi(\rho_1))$ et $\mathcal{R}(\Psi(\rho_2))$ sont disjoints.
(Homogénéité)	Quelles que soient les régions privées ρ_1 et ρ_2 telles que $\rho_1 \equiv \rho_2$, alors $\Psi(\rho_1) \equiv \Psi(\rho_2)$.

Tout d'abord remarquons que dans la définition ci-dessus le raffinement d'une région (privée ou publique) préserve son indice. Plus généralement, le raffinement est injectif et préserve les indices de toutes ses sous-régions. En particulier, lorsque deux types ont une région en commun, le raffinement préserve cet alias. Par ailleurs, les hypothèses de fraîcheur et de séparation ont pour conséquence le résultat suivant :

Lemma 4.3.1 (Lemme de séparation)**.** Lorsque deux types n'ont aucune région en commun, c'est-à-dire lorsque $\mathcal{R}(\tau_1) \cap \mathcal{R}(\tau_2)$ est vide, leurs raffinements ne comportent aucun alias non plus, c'est-à-dire que l'intersection $\mathcal{R}(\Psi(\tau_1)) \cap \mathcal{R}(\Psi(\tau_2))$ reste vide.

Démonstration. Par récurrence structurelle double sur τ_1 et τ_2. Il y a quatre cas de base. Le cas où soit τ_1 soit τ_2 est un type scalaire, est trivial. Le cas où τ_1 et τ_2 correspondent à deux régions privées, est vrai par définition, d'après l'hypothèse de séparation. Le cas où l'un des deux types est une région privée, et l'autre est une région publique dont toutes les composantes sont scalaires est vrai également par définition en vertu de l'hypothèse de fraîcheur. Enfin, le cas où les deux types sont des régions publiques dont toutes les composantes sont scalaires est trivial. Tous les autres cas se font en appliquant directement l'hypothèse de récurrence. □

Quelles sont alors les conséquences de l'hypothèse d'homogénéité ? Tout d'abord, remarquons que, grâce à l'hypothèse de l'homogénéité, le raffinement préserve l'égalité structurelle entre les types, ce qui se montre encore par récurrence structurelle double. D'une manière plus importante, l'homogénéité garantit que le raffinement préserve la relation d'équivalence modulo aliasing entre deux types. Pour montrer cette dernière affirmation, il suffit de montrer que pour tout type τ et toute substitution θ définie sur τ, il existe une substitution θ' définie sur $\Psi(\tau)$ telle que le diagramme donné dans la figure 4.2 commute. Pour montrer ce lemme, on pourrait songer à faire la preuve

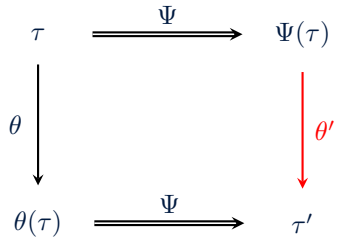


FIGURE 4.2 – Lemme de substitution pour le raffinement.

par récurrence sur la structure du type τ. Néanmoins, dans le cas où τ est une région $\{f_1 : \tau_1, \ldots, f_n : \tau_n\}_r$, en appliquant l'hypothèse de récurrence aux sous-types $\tau_1, \ldots \tau_n$, on obtiendrait n substitutions distinctes $\theta'_1, \ldots, \theta'_n$ qu'il faudrait ensuite utiliser pour montrer l'existence de la substitution θ'. Nous allons donc procéder différemment, en généralisant l'énoncé de la façon suivante :

Lemma 4.3.2 (Extension de substitution de régions par raffinement)**.** Soit un ensemble fini de types s et soit une substitution de régions θ définie sur s. Alors, il existe une substitution de régions θ' définie sur $\Psi(s)$ telle pour tout $\tau \in s$, on a

$$\theta'(\Psi(\tau)) = \Psi(\theta(\tau)).$$

Démonstration. Pour montrer le lemme, nous allons utiliser la récurrence sur une mesure bien choisie. Commençons par définir $h(\tau)$, la hauteur d'un type τ :

$$h(\tau) \triangleq \begin{cases} 0 & \text{si } \tau = \nu \\ 1 + \max\limits_{\tau.f \text{ défini}} h(\tau.f) & \text{sinon} \end{cases}$$

Par exemple, si τ est une région privée, alors $h(\tau) = 1$. Maintenant, définissons $H(s)$, la hauteur maximale d'un ensemble de types s comme l'entier égal à

$$H(s) \triangleq \max_{\tau \in s} h(\tau).$$

Soit alors s et θ satisfaisant les hypothèses de l'énoncé. Montrons le résultat par récurrence sur $H(s)$.

Le cas de base $H(s) = 0$ est trivial. Traitons le cas $H(s) = 1$. Cela signifie que l'ensemble s ne contient aucune région imbriquée. En particulier, si $\tau \in s$ est une région (privée ou publique), alors tous ses champs sont d'un type scalaire. Alors, si τ est une région publique ou un scalaire, on a $\Psi(\tau) = \tau$. Il suffit donc de poser simplement $\theta' = \theta$. Sinon, τ est une région privée. Remarquons d'abord que les régions τ et $\theta(\tau)$ sont équivalentes. Or, par hypothèse d'homogénéité, puisque $\tau \equiv \theta(\tau)$, on a également $\Psi(\tau) \equiv \Psi(\theta(\tau))$. Par conséquent, il existe une substitution θ'_τ telle que $\theta'_\tau(\Psi(\tau)) = \Psi(\theta(\tau))$. Or, l'hypothèse de séparation garantit que pour toute paire de régions privées $(\rho_1, \rho_2) \in s$, on a $\mathcal{R}(\Psi(\rho_1)) \cap \mathcal{R}(\Psi(\rho_2)) = \varnothing$. Par conséquent, on peut construire une substitution de régions θ'_s à partir de l'ensemble $\{\theta'_\tau \mid \tau \in s\}$, telle que pour tout $\tau \in s$, on a $\theta'_s(\Psi(\tau)) = \theta'_\tau(\Psi(\tau))$, ce qui permet de conclure le cas de base.

Sinon, supposons que $H(s) > 1$. Introduisons deux ensembles s_1 et s_2 tels que

$$s \triangleq s_1 \uplus s_2, \qquad H(s_1) < H(s_2) \quad \text{et} \quad \forall \tau_1, \tau_2 \in s_2 . h(\tau_1) = h(\tau_2)$$

Cela signifie donc que les ensembles s_1 et s_2 partitionnent s en deux et que les types de s_2 sont tous les éléments maximaux de s au sens de la hauteur. En particulier, l'ensemble s_2 ne contient que des régions. Maintenant, introduisons également l'ensemble s_0 défini par

$$s_0 \triangleq \{\rho.\pi \text{ est un type défini} \mid \rho \in s_2,\ \pi \neq \epsilon\}.$$

Enfin, posons $s_3 = s_0 \cup s_1$. Puisque $H(s_3) < H(s)$, on peut appliquer l'hypothèse de récurrence, ce qui donne l'existence d'une substitution θ' telle que pour tout $\tau \in s_3$ on a

$$\theta'(\Psi(\tau)) = \Psi(\theta(\tau)).$$

Considérons maintenant une région ρ quelconque dans l'ensemble s_2. Posons

$$\theta'(\Psi(\rho)) = \Psi(\theta(\rho)).$$

Remarquons d'abord que θ' ainsi étendue reste injective. En effet, d'une part, le raffinement est injectif, donc préserve l'indice de la région ρ ; d'autre part, ρ n'apparaît jamais dans $\mathcal{R}(s_3)$ puisque $h(\rho) > H(s_3)$. Il reste donc à montrer que pour tout chemin π, ou bien $\theta'(\Psi(\rho).\pi) = \theta'(\Psi(\rho)).\pi$ ou bien aucune des deux expressions n'est définie. Considérons alors un chemin π tel que le type $\theta'(\Psi(\rho).\pi)$ est défini. Introduisons la notation suivante pour le chemin π :

$$\pi = \pi^\circ \pi^\bullet$$

où π° est le plus long préfixe de π tel que $\rho.\pi^\circ$ soit défini. Avec cette notation, on a

$$\theta'(\Psi(\rho).\pi^\circ\pi^\bullet) = \theta'(\Psi(\rho.\pi^\circ).\pi^\bullet)$$

et puisque le type $\rho.\pi^\circ \in s_3$, on a par hypothèse de récurrence

$$\theta'(\Psi(\rho.\pi^\circ).\pi^\bullet) = \theta'(\Psi(\rho.\pi^\circ)).\pi^\bullet = \Psi(\theta(\rho.\pi^\circ)).\pi^\bullet.$$

Enfin, θ étant une substitution de régions, on continue la chaîne des égalités avec

$$\Psi(\theta(\rho.\pi^\circ)).\pi^\bullet = \Psi(\theta(\rho)).\pi^\circ\pi^\bullet = \Psi(\theta(\rho)).\pi = \theta'(\Psi(\rho)).\pi$$

ce qui permet de conclure. □

4.3.2 Raffinement des signatures de fonctions

Rappelons que dans RegML, chaque fonction possède une signature de la forme

$$\tau_1 \times \ldots \times \tau_n \to \tau \cdot (\omega \cdot \varphi)$$

où $(\omega \cdot \varphi)$ correspond à l'effet exposé par la signature et qui vérifie les trois conditions suivantes : $\omega \subseteq \mathcal{R}(\tau_1, \ldots, \tau_n)$, $\varphi \subseteq \mathcal{R}(\tau_1, \ldots, \tau_n, \tau)$ et $\mathcal{R}(\tau) \setminus \mathcal{R}(\tau_1, \ldots, \tau_n) \subseteq \varphi$. Le raffinement d'une signature ne peut donc être défini simplement par

$$\Psi(\tau_1) \times \cdots \times \Psi(\tau_n) \to \Psi(\tau) \cdot (\Psi(\omega) \cdot \Psi(\varphi)).$$

En effet, puisque le raffinement peut ajouter de nouvelles régions dans le type du résultat, elles doivent toutes être invalidées. La troisième hypothèse n'est donc plus vérifiée, c'est-à-dire que l'on n'a plus

$$\mathcal{R}(\Psi(\tau)) \setminus \mathcal{R}(\Psi(\tau_1), \ldots, \Psi(\tau_n)) \subseteq \Psi(\varphi).$$

Plutôt que de chercher à ajuster cette définition pour les trois conditions, notre idée est de proposer une définition dans l'esprit du principe de substitution de Liskov : toutes les propriétés que la signature vérifie avant le raffinement doivent rester vraies après. Or, si l'on regarde quelles sont les propriétés que les effets peuvent avoir en plus des trois conditions ci-dessus (et le fait que par définition, les ensembles ω et φ sont disjoints), on s'aperçoit que la propriété la plus importante est le prédicat de la validité (définition 3.2.1 sur la page 87) entre un type et un effet $(\omega \cdot \varphi) \triangleright \tau$. On pourrait alors songer à définir le raffinement d'une signature par

$$\Psi(\tau_1) \times \cdots \times \Psi(\tau_n) \to \Psi(\tau) \cdot (\omega' \cdot \varphi')$$

où $(\omega' \cdot \varphi')$ est un certain effet tel que pour tout type $\tau \in \operatorname{dom} \Psi$, si $(\omega \cdot \varphi) \triangleright \tau$, alors $(\omega' \cdot \varphi') \triangleright \Psi(\tau)$. Cependant, compte tenu du fait que par définition $\omega \subseteq \mathcal{R}(\tau_1, \ldots, \tau_n)$ et $\varphi \subseteq \mathcal{R}(\tau_1, \ldots, \tau_n, \tau)$, on peut donner une condition moins forte où l'on ne quantifie pas sur tous les types dans le domaine de Ψ. Concrètement, nous adoptons la définition suivante :

Définition 4.3.2. Soit Ψ un raffinement de types. Le *raffinement d'une signature*

$$\tau_1 \times \ldots \times \tau_n \to \tau \cdot (\omega \cdot \varphi)$$

est une signature de la forme

$$\Psi(\tau_1) \times \cdots \times \Psi(\tau_n) \to \Psi(\tau) \cdot (\omega' \cdot \varphi')$$

où $(\omega' \cdot \varphi')$ est un effet tel que, quelle que soit la région $\rho \in \mathcal{R}(\tau_1, \ldots, \tau_n)$ valide vis-à-vis des effets de la signature $(\omega \cdot \varphi)$, c'est-à-dire $(\omega \cdot \varphi) \triangleright \rho$, le raffinement de ρ est valide vis-à-vis des effets de la signature raffinée, c'est-à-dire $(\omega' \cdot \varphi') \triangleright \Psi(\rho)$.

Notons que l'avantage de cette définition est qu'elle offre une grande liberté de choix pour le raffinement d'une fonction : tant que la condition de la validité sur l'effet $(\omega' \cdot \varphi')$ est vérifiée, le raffinement d'une fonction peut avoir moins de régions écrites, plus de régions invalidées, etc. Il est important de noter que, du point de vue de la preuve, raffiner une signature par une autre qui aurait *moins* de régions écrites serait incorrect. Là encore, rien n'empêche d'ajuster la définition ci-dessus pour tenir compte de cette condition. Néanmoins, du point de vue du typage, le seul qui nous intéresse ici, ne pas préserver des régions écrites lors du raffinement ne pose pas de problème vis-à-vis du contrôle statique des alias, tant que cela préserve la validité du prédicat d'accessibilité. Par conséquent, nous gardons dans la suite la définition 4.3.2 telle qu'elle.

4.3.3 Préservation du typage par le raffinement

Nous sommes prêts maintenant pour énoncer le théorème de préservation du typage par le raffinement.

Théorème 4.3.1. Soit un jugement de typage $\Gamma \cdot \Sigma \vdash e : \tau \cdot \varepsilon$ dérivable. Alors il existe un effet ε' tel que le jugement

$$\Psi(\Gamma) \cdot \Psi(\Sigma) \vdash e : \Psi(\tau) \cdot \varepsilon'$$

est dérivable et quel que soit un type $\tau \in \operatorname{dom} \Psi$, si $\varepsilon \triangleright \tau$, alors $\varepsilon' \triangleright \Psi(\tau)$.

Démonstration. Par récurrence sur la dérivation du typage. Remarquons au préalable que si l'effet $\varepsilon = \bot$, poser $\varepsilon' = \bot$ vérifie trivialement l'implication de l'énoncé. Par conséquent, les cas ou e est une expression atomique (variable, constante, adresse mémoire) ainsi que le cas où e est de forme $a.f$ (accès au champs f) se vérifient directement en appliquant la définition du raffinement. Le cas où e est une conditionnelle se montre en appliquant directement l'hypothèse de récurrence. Traitons donc les quatre cas restants.

Cas de la règle T-Let. La dérivation du typage de e se termine par

$$\frac{\Gamma \cdot \Sigma \vdash e_1 : \tau_1 \cdot \varepsilon_1 \qquad \Gamma[x \mapsto \tau_1] \cdot \Sigma \vdash e_2 : \tau_2 \cdot \varepsilon_2 \qquad \forall \rho \in \Gamma(\mathcal{F}_v(e_2) \setminus \{x\}) \cup \Sigma(\mathcal{F}_\ell(e_2)).\ \varepsilon_1 \triangleright \rho}{\Gamma \cdot \Sigma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2 \cdot \varepsilon_1 \sqcup \varepsilon_2}$$

En appliquant l'hypothèse de récurrence, nous avons

$$\Psi(\Gamma) \cdot \Psi(\Sigma) \vdash e_1 : \Psi(\tau_1) \cdot \varepsilon_1' \qquad \text{et} \qquad \Psi(\Gamma)[x \mapsto \Psi(\tau_1)] \cdot \Psi(\Sigma) \vdash e_2 : \Psi(\tau_2) \cdot \varepsilon_2'$$

avec des effets ε_1' et ε_2' tels que

$$\forall \tau \in \operatorname{dom} \Psi.\ (\varepsilon_1 \triangleright \tau \implies \varepsilon_1' \triangleright \Psi(\tau)) \wedge (\varepsilon_2 \triangleright \tau \implies \varepsilon_2' \triangleright \Psi(\tau))$$

D'après le lemme 3.2.4, on a donc

$$\forall \tau \in \operatorname{dom} \Psi.\ \varepsilon_1 \sqcup \varepsilon_2 \triangleright \tau \implies \varepsilon_1' \sqcup \varepsilon_2' \triangleright \Psi(\tau).$$

De plus, si une région ρ' est dans $\Psi(\Gamma(\mathcal{F}_v(e_2) \setminus \{x\}) \cup \Sigma(\mathcal{F}_\ell(e_2)))$, elle s'écrit alors $\Psi(\rho)$ pour une certaine région ρ dans $\Gamma(\mathcal{F}_v(e_2) \setminus \{x\}) \cup \Sigma(\mathcal{F}_\ell(e_2))$. Or, puisque par hypothèse $\varepsilon_1 \triangleright \rho$, on a, d'après ce qui précède, $\varepsilon_1' \triangleright \rho'$. Ainsi, on peut conclure en dérivant le jugement

$$\frac{\Psi(\Gamma) \cdot \Psi(\Sigma) \vdash e_1 : \Psi(\tau_1) \cdot \varepsilon_1' \qquad \Psi(\Gamma)[x \mapsto \Psi(\tau_1)] \cdot \Psi(\Sigma) \vdash e_2 : \Psi(\tau_2) \cdot \varepsilon_2' \qquad \forall \rho' \in \Psi(\Gamma(\mathcal{F}_v(e_2) \setminus \{x\}) \cup \Sigma(\mathcal{F}_\ell(e_2))).\ \varepsilon_1' \triangleright \rho'}{\Psi(\Gamma) \cdot \Psi(\Sigma) \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \Psi(\tau_2) \cdot \varepsilon_1' \sqcup \varepsilon_2'}$$

Cas de la règle T-Alloc. La dérivation du typage de e se termine par

$$\frac{\Gamma \cdot \Sigma \vdash a_i : \tau_i \cdot \bot \qquad \rho = \{f_i : \tau_i\ ^{i \in [1,\ldots,n]}\}_r \qquad f_i \text{ deux à deux distincts}}{\Gamma \cdot \Sigma \vdash \{f_i = a_i\ ^{i \in [1,\ldots,n]}\} : \rho \cdot (\varnothing \cdot \{\rho\})}$$

Par définition, $\Psi(\rho) = \{f_i : \Psi(\tau_i)\ ^{i\in[1,\ldots,n]}\}_r$. Par hypothèse de récurrence, nous avons pour $i \in [1, \ldots, n]$, que le $\Psi(\Gamma) \cdot \Psi(\Sigma) \vdash a_i : \Psi(\tau_i) \cdot \bot$ est dérivable. On peut donc dériver

$$\frac{\Psi(\Gamma) \cdot \Psi(\Sigma) \vdash a_i : \Psi(\tau_i) \cdot \bot \quad \Psi(\rho) = \{f_i : \Psi(\tau_i)\ ^{i\in[1,\ldots,n]}\}_r \quad f_i \text{ deux à deux distincts}}{\Psi(\Gamma) \cdot \Psi(\Sigma) \vdash \{f_i = a_i\ ^{i\in[1,\ldots,n]}\} : \Psi(\rho) \cdot (\varnothing \cdot \{\Psi(\rho)\})}$$

Montrons maintenant la partie de l'énoncé concernant ε' qui est égal dans ce cas à $(\varnothing \cdot \{\Psi(\rho)\})$. Considérons un type $\tau \in \operatorname{dom}\Psi$ tel que $(\varnothing \cdot \{\rho\}) \triangleright \tau$. Puisque l'ensemble des régions écrites est vide, on a $(\varnothing \cdot \{\rho\}) \triangleright \tau$ si est seulement si ou bien τ est une sous-région stricte de ρ, ou bien $\mathcal{R}(\rho) \cap \mathcal{R}(\tau) = \varnothing$. Dans le premier cas, $\Psi(\tau)$ est clairement une sous-région stricte de $\Psi(\rho)$. Dans le deuxième cas, $\mathcal{R}(\Psi(\rho)) \cap \mathcal{R}(\Psi(\tau)) = \varnothing$ d'après le lemme de séparation. On a donc $(\varnothing \cdot \{\Psi(\rho)\}) \triangleright \Psi(\tau)$ ce qui permet de conclure.

Cas de la règle T-Assign. La dérivation du typage de e se termine par

$$\frac{\begin{array}{c}\Gamma \cdot \Sigma \vdash a_i : \rho_i \cdot \bot \qquad \Gamma \cdot \Sigma \vdash a'_{i,j} : \tau'_{i,j} \cdot \bot \qquad \rho_i.f_{i,j} \simeq \tau'_{i,j} \\ \rho_i \text{ deux à deux distincts} \qquad \forall i.\ f_{i,j} \text{ deux à deux distincts} \\ \varphi = \Phi(\rho_i.\{f_{i,j} \leftarrow \tau'_{i,j}\ ^{j\in[1,\ldots,k_i]}\}^{i\in[1,\ldots,n]})\end{array}}{\Gamma \cdot \Sigma \vdash a_i.\{f_{i,j} \leftarrow a'_{i,j}\ ^{j\in[1,\ldots,k_i]}\}^{i\in[1,\ldots,n]} : \texttt{Unit} \cdot (\{\rho_1, \ldots, \rho_n\} \cdot \varphi)}$$

En appliquant l'hypothèse de récurrence, on a pour $i \in [1, \ldots, n]$ et $j \in [1, \ldots, k_i]$ la dérivabilité des jugements

$$\Psi(\Gamma) \cdot \Psi(\Sigma) \vdash a_i : \Psi(\rho_i) \cdot \bot \qquad \text{et} \qquad \Psi(\Gamma) \cdot \Psi(\Sigma) \vdash a'_{i,j} : \Psi(\tau'_{i,j}) \cdot \bot.$$

De plus, le raffinement préserve l'égalité structurelle, donc on a $\Psi(\rho_i).f_{i,j} \simeq \Psi(\tau'_{i,j})$ en vertu des relations $\rho_i.f_{i,j} \simeq \tau'_{i,j}$.

Posons $\varphi' = \Phi(\Psi(\rho_i).\{f_{i,j} \leftarrow \Psi(\tau'_{i,j})\ ^{j\in[1,\ldots,k_i]}\}^{i\in[1,\ldots,n]})$ et désignons par A' l'argument de l'opération Φ. Rappelons que A' correspond à la projection de l'affectation parallèle au niveau des types. Tout d'abord, montrons que l'opération Φ est bien définie sur A'. Pour cela, introduisons les notations suivantes. Comme au chapitre précédent[2], notons $A'(\Psi(\rho_i), f_{i,j}) \triangleq \Psi(\tau'_{ij})$ et $A'(\rho, g) \triangleq \rho.g$ lorsque la région ρ n'est pas dans $\{\Psi(\rho_1), \ldots, \Psi(\rho_n)\}$. Notons aussi que pour $\rho \in \mathcal{R}(\{\Psi(\rho_1), \ldots, \Psi(\rho_n)\})$, on définit $A'(\rho, \pi)$ comme dans le chapitre précédent, à savoir par les équations :

$$A'(\rho, \pi) \triangleq \begin{cases} \rho & \text{si } \pi = \epsilon \\ A'(\rho, f) & \text{si } \pi = f\epsilon \text{ et } A'(\rho, f) = \nu \\ A'(\rho', \pi') & \text{si } \pi = f\pi' \text{ et } A'(\rho, f) = \rho' \end{cases}$$

Posons maintenant la relation binaire $\sigma_{A'} \triangleq \{\, \langle A'(\Psi(\rho_i), \pi), \Psi(\rho_i).\pi\rangle \mid i \in [1, \ldots, n]\}$ et montrons que $\sigma_{A'}$ est bijective.

2. Voir le paragraphe 2 de la page 94

Tout d'abord, pour tout chemin π tel que $\Psi(\rho_i).\pi$ est défini, notons $\pi \triangleq \pi^\circ\pi^\bullet$, où π° est le plus long préfixe de π tel que $\rho_i.\pi^\circ$ soit défini. Remarquons alors que

$$A'(\Psi(\rho_i), \pi^\circ\pi^\bullet) = \Psi(A(\rho_i, \pi^\circ)).\pi^\bullet$$

ce que l'on peut encore représenter graphiquement à l'aide de la figure 4.3 laquelle montre comment chaque paire dans $\sigma_{A'}$ est structurée :

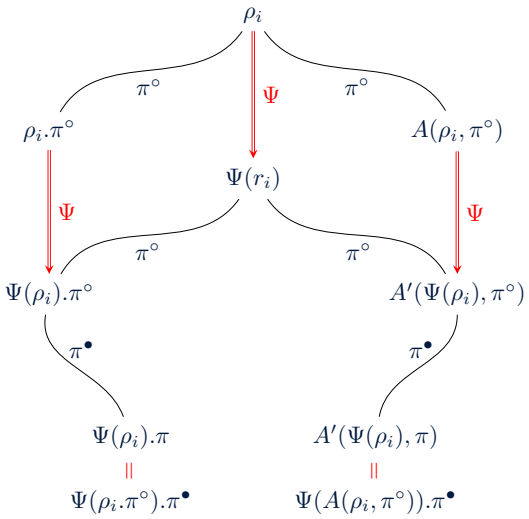


FIGURE 4.3 – La structure de σ'_A.

En effet, le raffinement préserve la structure d'aliasing, donc $A(\rho_i, \pi^\circ)$ étant bien défini, on a $A'(\Psi(\rho_i), \pi^\circ) = \Psi(A(\rho_i, \pi^\circ))$. De plus, si le suffixe $\pi^\bullet$ n'est pas vide, alors nécessairement $\rho_i.\pi^\circ$ est un type privé. Par conséquent, il ne fait pas partie des types $\{\Psi(\rho_1), \ldots, \Psi(\rho_n)\}$ et donc a bien

$$A'(\Psi(\rho_i), \pi^\circ\pi^\bullet) = A'(\Psi(A(\rho_i, \pi^\circ), \pi^\bullet)) = \Psi(A(\rho_i, \pi^\circ)).\pi^\bullet.$$

Considérons maintenant dans $\sigma_{A'}$ deux paires

$$\langle A'(\Psi(\rho_i), \pi_i), \Psi(\rho_i).\pi_i\rangle \quad \text{et} \quad \langle A'(\Psi(\rho_j), \pi_j), \Psi(\rho_j).\pi_j\rangle$$

pour lesquelles posons respectivement $\pi_i \triangleq \pi_i^\circ\pi_i^\bullet$ et $\pi_j \triangleq \pi_j^\circ\pi_j^\bullet$. Il suffit alors de montrer l'équivalence 4.1 ci-dessous :

$$\Psi(\rho_i).\pi_i^\circ\pi_i^\bullet = \Psi(\rho_j).\pi_j^\circ\pi_j^\bullet \iff A'(\Psi(\rho_i), \pi_i^\circ\pi_i^\bullet) = A'(\Psi(\rho_j), \pi_j^\circ\pi_j^\bullet). \tag{4.1}$$

Pour cela, remarquons que pour tout $\langle \rho, \rho'\rangle \in \sigma_{A'}$, on a clairement $\rho \simeq \rho'$. Par conséquent, lorsque l'un des quatre types $\Psi(\rho_i).\pi_i, \Psi(\rho_j).\pi_j, A'(\Psi(\rho_i), \pi_i), A'(\Psi(\rho_j), \pi_j)$ est un type scalaire, l'équivalence 4.1 est trivialement vérifiée.

Sinon, les quatre types dénotent nécessairement des régions avec le même statut public/privé. Or, dans ce cas-là, le chemin $\pi_i^\bullet$ est vide si et seulement si le chemin $\pi_j^\bullet$

l'est. En effet, si l'un des deux chemins est vide, alors que l'autre ne l'est pas, cela contredit l'hypothèse de fraîcheur des indices introduits par le raffinement.

Lorsque $\pi_i^\bullet = \epsilon = \pi_j^\bullet$, l'équivalence 4.1 est vérifiée, car d'une part le raffinement est injectif et d'autre part la relation σ_A est bijective par hypothèse du typage de e.

Supposons enfin que ni $\pi_i^\bullet$, ni $\pi_j^\bullet$ ne sont vides. Or, dans ce cas-là, les quatre types considérés étant des régions, on a, en vertu de l'hypothèse de séparation, les deux équivalences ci-dessous :

$$\Psi(\rho_i).\pi_i = \Psi(\rho_j).\pi_j \iff \rho_i.\pi_i^\circ = \rho_j.\pi_j^\circ$$

$$A'(\Psi(\rho_i), \pi_i) = A'(\Psi(\rho_j), \pi_j) \iff A(\rho_i, \pi_i^\circ) = A(\rho_j, \pi_j^\circ)$$

lesquelles, mises ensemble, impliquent l'équivalence 4.1 et donc la bijectivité de $\sigma_{A'}$.

En résumé, on a montré que Φ sur A' est bien défini, ce qui permet de poser

$$\varphi' \triangleq \{\, \rho \mid \text{il existe } \rho' \neq \rho \text{ telle que } \langle \rho, \rho' \rangle \in \sigma_{A'} \text{ ou } \langle \rho', \rho \rangle \in \sigma_{A'} \,\}$$

et dériver le jugement de typage

$$\frac{\begin{array}{cc} \Psi(\Gamma) \cdot \Psi(\Sigma) \vdash a'_{i,j} : \Psi(\tau'_{i,j}) \cdot \bot & \Psi(\Gamma) \cdot \Psi(\Sigma) \vdash a_i : \Psi(\rho_i) \cdot \bot \\ \Psi(\rho_i) \text{ deux à deux distincts} & \forall i.\, f_{i,j} \text{ deux à deux distincts} \\ \Psi(\rho_i).f_{i,j} \simeq \Psi(t'_{i,j}) & \varphi' = \Phi(\Psi(\rho_i).\{f_{i,j} \leftarrow \Psi(\tau'_{i,j})^{\ j\in[1,\ldots,k_i]}\}^{i\in[1,\ldots,n]}) \end{array}}{\Psi(\Gamma) \cdot \Psi(\Sigma) \vdash a_i.\{f_{i,j} \leftarrow a'_{i,j}{}^{\ j\in[1,\ldots,k_i]}\}^{\,i\in[1,\ldots,n]} : \mathtt{Unit} \cdot (\{\Psi(\rho_1), \ldots, \Psi(\rho_n)\} \cdot \varphi')}$$

On a donc posé $\varepsilon' = (\{\Psi(\rho_1), \ldots, \Psi(\rho_n)\} \cdot \varphi')$ et il reste à montrer que pour $\tau \in \operatorname{dom} \Psi$ on a l'implication $\varepsilon \triangleright \tau \implies \varepsilon' \triangleright \Psi(\tau)$. Montrons ce résultat par récurrence sur la dérivation du jugement $\varepsilon' \triangleright \tau$. Le résultat est immédiat lorsque τ est un type scalaire ou bien quand τ est l'une des régions parmi $\rho_1, \ldots, \rho_n$. Traitons le cas inductif où τ est une région ρ telle que $\rho \notin \{\rho_1, \ldots, \rho_n\}$, $\rho \notin \varphi$ et telle que pour tout f, si $\rho.f$ est défini, alors $(\{\rho_1, \ldots, \rho_n\} \cdot \varphi) \triangleright \rho.f$. Il est clair que $\Psi(\rho) \notin \{\Psi(\rho_1), \ldots, \Psi(\rho_n)\}$. Supposons par absurde que $\Psi(\rho) \in \varphi'$. Puisque $\rho \notin \varphi$, on a $\Psi(\rho) \notin \Psi(\varphi)$. Donc, si $\Psi(\rho) \in \varphi'$, c'est que $\Psi(\rho)$ est nécessairement une région fraîche introduite par le raffinement. Or, cela contredit l'hypothèse que $\rho \in \operatorname{dom} \Psi$. Donc, $\Psi(\rho) \notin \varphi'$.

Il reste donc à montrer que pour tout champ f tel que $\Psi(\rho).f$ est défini, on a $\varepsilon' \triangleright \Psi(\rho).f$. Par hypothèse de récurrence, on a, pour tout f tel que $\rho.f$ est défini, $\varepsilon' \triangleright \Psi(\rho).f$. Bien entendu, cela ne suffit pas pour conclure, car il faut considérer également les champs f tel que $\Psi(\rho).f$ soit une région fraîche introduite par le raffinement. Supposons par absurde qu'il existe un champ f tel que la région $\Psi(\rho).f$, introduite par le raffinement, est dans φ'. Cela signifie qu'il existe une région ρ_i et un chemin $\pi \triangleq \pi^\circ\pi^\bullet$ tels que $\langle \Psi(A(\rho_i.\pi^\circ), \pi^\bullet),\ \Psi(\rho_i.\pi^\circ).\pi^\bullet \rangle \in \sigma_{A'}$ avec $\Psi(A(\rho_i.\pi^\circ), \pi^\bullet) \neq \Psi(\rho_i.\pi^\circ).\pi^\bullet$ et tels que $\Psi(\rho).f$ est égal ou bien à $\Psi(\rho_i.\pi^\circ).\pi^\bullet$, ou bien à $\Psi(A(\rho_i.\pi^\circ).\pi^\bullet$. En prenant la contraposée de l'hypothèse de séparation, on a alors, selon le cas, ou bien $\rho = \rho_i.\pi^\circ$, ou bien $\rho = A(\rho_i, \pi^\circ)$.

D'autre part, par hypothèse ρ n'est pas dans φ, donc nécessairement $A(\rho_i, \pi^\circ) = \rho_i.\pi^\circ$. Par conséquent, $\Psi(A(\rho_i, \pi^\circ)) = \Psi(\rho_i.\pi^\circ)$, ce qui contredit le fait que $\Psi(A(\rho_i.\pi^\circ), \pi^\bullet) \neq \Psi(\rho_i.\pi^\circ).\pi^\bullet$. Ainsi, on a bien $\varepsilon' \triangleright \Psi(\rho).f$ pour tout f tel que $\Psi(\rho).f$ est défini, ce qui permet de conclure.

Cas de la règle T-Call. La dérivation du typage de e se termine par

$$\frac{p : \tau_1 \times \ldots \times \tau_n \to \tau_0 \cdot (\omega \cdot \varphi) \qquad \Gamma \cdot \Sigma \vdash a_i : \theta(\tau_i) \cdot \bot}{\Gamma \cdot \Sigma \vdash p(a_1, \ldots, a_n) : \theta(\tau_0) \cdot (\theta(\omega) \cdot \theta(\varphi))}$$

Le raffinement de la signature de p est égale à

$$\Psi(\tau_1) \times \cdots \times \Psi(\tau_n) \to \Psi(\tau_0) \cdot (\omega' \cdot \varphi')$$

où pour toute région ρ dans $\mathcal{R}(\tau_1, \ldots, \tau_n)$, $(\omega \cdot \varphi) \triangleright \rho$ implique $(\omega' \cdot \varphi') \triangleright \Psi(\rho)$. Appliquons le lemme 4.3.2 aux types $\tau_0, \ldots, \tau_n$ et à la substitution θ, ce qui donne l'existence d'une substitution θ' telle que pour tout $i \in [0, \ldots, n]$ on a

$$\theta'(\Psi(\tau_i)) = \Psi(\theta(\tau_i)).$$

La substitution de régions θ' et le raffinement Ψ étant injectifs, leur composition l'est également. Par conséquent, pour tout $\tau \in \{\tau_0, \ldots, \tau_n\}$ et tout chemin π tel que $\tau.\pi$ est défini, on a $\theta'(\Psi(\tau.\pi)) = \theta'(\Psi(\tau)).\pi$. En particulier, les régions dans l'effet $(\omega' \cdot \varphi')$ étant incluses dans $\mathcal{R}(\Psi(\tau_0), \ldots, \Psi(\tau_n))$, l'effet $(\theta'(\omega') \cdot \theta'(\varphi'))$ est bien défini.

Appliquons maintenant l'hypothèse de récurrence au typage des atomes $a_1, \ldots, a_n$, ce qui donne $\Psi(\Gamma) \cdot \Psi(\Sigma) \vdash a_i : \Psi(\theta(\tau_i)) \cdot \bot$ pour tout $i \in [1, \ldots, n]$. Par conséquent, on peut dériver le jugement

$$\frac{p : \Psi(\tau_1) \times \cdots \times \Psi(\tau_n) \to \Psi(\tau_0) \cdot (\omega' \cdot \varphi') \qquad \Psi(\Gamma) \cdot \Psi(\Sigma) \vdash a_i : \theta'(\Psi(\tau_i)) \cdot \bot}{\Psi(\Gamma) \cdot \Psi(\Sigma) \vdash p(a_1, \ldots, a_n) : \theta'(\Psi(\tau_0)) \cdot (\theta'(\omega') \cdot \theta'(\varphi'))}$$

Il reste à monter que quel que soit un type $\tau \in \operatorname{dom} \Psi$, lorsqu'on a $(\theta(\omega) \cdot \theta(\varphi)) \triangleright \tau$, alors on a $(\theta'(\omega') \cdot \theta'(\varphi')) \triangleright \Psi(\tau)$. Montrons le résultat par récurrence sur la dérivation du jugement $(\theta(\omega) \cdot \theta(\varphi)) \triangleright \tau$ en raisonnant par cas, selon que τ est dans l'image de θ ou pas. Lorsque τ est un type scalaire, le résultat est trivial. Supposons donc que τ est une certaine région ρ. Lorsque $\rho \in \operatorname{Im} \theta$, il existe donc $\rho^\star \in \operatorname{dom} \theta$ telle que $\rho = \theta(\rho^\star)$. Dans ce cas-là, on a $(\theta(\omega) \cdot \theta(\varphi)) \triangleright \theta(\rho^\star)$ et donc $(\omega \cdot \varphi) \triangleright \rho^\star$, puisque θ est injective. Par ailleurs, comme $\rho^\star \in \operatorname{dom} \theta$, on a $\rho^\star \in \mathcal{R}(\tau_0, \tau_1, \ldots, \tau_n)$, Montrons que nécessairement $\rho^\star \in \mathcal{R}(\tau_1, \ldots, \tau_n)$. En effet, si $\rho^\star \in \mathcal{R}(\tau_0) \setminus \mathcal{R}(\tau_1, \ldots, \tau_n)$, alors $\rho^\star \in \varphi$, ce qui est impossible, puisque $(\omega \cdot \varphi) \triangleright \rho^\star$. Donc, $\rho^\star \in \mathcal{R}(\tau_1, \ldots, \tau_n)$. D'après la définition du raffinement, on a alors $(\omega' \cdot \varphi') \triangleright \Psi(\rho^\star)$, ce qui permet d'écrire

$$(\theta'(\omega') \cdot \theta'(\varphi')) \triangleright \theta'(\Psi(\rho^\star))$$

et puisque $\rho^\star \in \mathcal{R}(\tau_1, \ldots, \tau_n)$, on a d'après ce qui précède, $\theta'(\Psi(\rho^\star)) = \Psi(\theta(\rho^\star)) = \Psi(\rho)$, ce qui permet de conclure dans le cas où $\rho \in \operatorname{Im} \theta$.

Sinon, considérons le cas où $\rho \notin \operatorname{Im}\theta$. On en déduit que $\rho \notin \theta(\omega)$ et $\rho \notin \theta(\varphi)$. Par conséquent, comme $(\theta(\omega)\cdot\theta(\varphi)) \triangleright \rho$, on déduit que, pour tout champ f tel que $\rho.f$ est défini, on a $(\theta(\omega)\cdot\theta(\varphi)) \triangleright \rho.f$. Par hypothèse de récurrence, on a alors pour tout f tel que $\rho.f$ soit défini $(\theta'(\omega')\cdot\theta'(\varphi')) \triangleright \Psi(\rho).f$. Montrons également que pour tout f' tel que le type $\Psi(\rho).f'$ soit défini avec f' frais, on a $(\theta'(\omega')\cdot\theta'(\varphi')) \triangleright \Psi(\rho).f'$. Supposons par l'absurde que ce n'est pas le cas pour un certain f'. Alors, il existe un chemin π tel que la région $\Psi(\rho).f'\pi \in \theta'(\varphi')$ et tel que pour tout $\pi' \prec \pi$, $\Psi(\rho).f'\pi' \notin \theta'(\omega')$. Or, puisque $\Psi(\rho).f'\pi \in \theta'(\varphi')$, on peut écrire $\Psi(\rho).f'\pi = \theta'(\Psi(\tau_j).\pi_j^\circ\pi_j^\bullet)$ pour un certain type τ_j parmi les types $\tau_0,\dots,\tau_n$ et un certain chemin $\pi_j^\circ\pi_j^\bullet$. On en déduit l'égalité $\Psi(\rho).f'\pi = \Psi(\theta(\tau_j.\pi_j^\circ)).\pi_j^\bullet$ où $\pi_j^\bullet$ est nécessairement un chemin non-vide, car sinon, on aurait l'égalité entre deux régions où l'une d'entre elles aurait un indice frais alors que l'autre non, donc l'égalité entre deux régions ayant des indices distincts, ce qui est absurde. Mais alors, cela signifie que ρ et $\theta(\tau_j.\pi_j^\circ)$ sont deux régions privées telles que l'intersection de $\Psi(\rho)$ avec $\Psi(\theta(\tau_j.\pi_j^\circ))$ n'est pas vide. D'après l'hypothèse de séparation, cela implique que $\rho = \theta(\tau_j.\pi_j^\circ)$ et contredit ainsi l'hypothèse que ρ n'appartient pas à l'image de θ. Ainsi, pour tout champ f' tel que $\Psi(\rho).f'$ est défini, on a $(\theta'(\omega')\cdot\theta'(\varphi'))\triangleright\Psi(\rho).f'$. Il suffit donc de vérifier que $\Psi(\rho) \notin \theta'(\omega')$ et que $\Psi(\rho) \notin \theta'(\varphi')$, ce qui se démontre par l'absurde de la manière identique au raisonnement que nous venons de faire. Par conséquent, on peut dériver $(\theta'(\omega')\cdot\theta'(\varphi')) \triangleright \theta'(\Psi(\rho_0))$ par la règle VP_{IND}, ce qui permet de conclure dans le cas de l'appel de fonction. □

4.4 Travaux connexes

Le raffinement des régions que nous avons présenté dans ce chapitre a de nombreux points en commun avec des approches existantes, notamment avec la méthode B. Dans la méthode B, la notion centrale est celle de machine abstraite, qui permet notamment de commencer le développement par la construction d'un modèle abstrait. Chaque machine abstraite décrit son propre état à travers l'ensemble des attributs et des opérations qui permettent de les manipuler, en respectant des invariants sur l'état. Le raffinement d'une machine abstraite consiste alors en une machine qui décrit un état plus précis : de nouveaux attributs peuvent être introduits, des opérations peuvent être rendues plus déterministes, etc. La validité du raffinement est alors assurée en montrant que toutes les propriétés vraies pour l'état de la machine abstraite sont préservée par son raffinement. On voit donc que si l'on considère en Why3 un objet mutable donné, on peut le rapprocher d'une machine abstraite, lorsqu'il est d'un type enregistrement privé, et son raffinement est, *mutatis mutandis*, une machine concrète qui contient plus de champs, dont l'invariant est renforcé, etc. Néanmoins, à la différence de ce qui est fait dans la méthode B, un type d'enregistrement permet de construire plusieurs objets mutables correspondant à des régions de mémoire séparées, alors qu'une machine B décrit toujours un état global.

Dans un article paru récemment [4] Leino et Koenig proposent d'intégrer le raffinement des modules dans l'outil de vérification déductive Dafny [5]. Leur travail est à plusieurs égards similaire au nôtre. D'une part, Dafny ne fait pas non plus de distinction interface/ module : la même notion de module sert pour exprimer et raffiner l'abstraction. D'autre part, les auteurs proposent également de raffiner de structures de données. Cependant, le contexte est plus orienté programmation objet où le raffinement de données est véhiculé par des classes et non pas par des types enregistrements. Par ailleurs, la problématique d'aliasing est différente : au lieu d'imposer un contrôle statique des alias, Dafny utilise l'approche des *dynamic frames* pour raisonner sur les données mutables allouées dynamiquement.

Enfin, il est également intéressant de comparer notre système de module de et celui d'OCaml [6]. En OCaml, la dichotomie interface/module est explicite, alors que pour nous, il n'y a qu'une seule notion de module. La conséquence de ce choix de conception est qu'en Why3, à la différence des langages ML, il n'y a pas de notion explicite de *foncteur* : la paramétrisation d'un module se fait avec la clause `use` comme dans l'exemple du module `MutableSet` dans l'introduction où l'on importe le module des graphes avec l'instruction `use import Graph as G`. Quant à l'instanciation d'un foncteur, elle est exprimée par un mécanisme différent, appelé le *clonage* et qui sert en même temps pour vérifier qu'un module est un raffinement d'un autre module. Le clonage d'un module, à la différence de l'utilisation d'un module qui rend simplement visibles les déclarations du module utilisé, a pour effet de créer une copie fraîche de chaque déclaration du module cloné, éventuellement en lui associant une nouvelle définition plus

précise. Pour indiquer au système que le module `MutableSetByHashtbl` est un raffinement du module `MutableSet`, l'idée est de clore le module `MutableSetByHashtbl` par une déclaration de clonage

```
clone MutableSet with type t = t, val create = create, ...
```

L'instanciation d'un foncteur est exprimée alors de la même manière, en substituant lors du clonage les modules de paramètres par leurs implémentations. Pour obtenir le code du parcours de graphe avec les tables de hachages comme implémentation des ensembles mutables, il faudra écrire

```
module DFSWithHashtbl
  clone DFS with MutableSet = MutableSetByHashtbl
end
```

La figure 4.4 résume brièvement ces différences de conception entre le système de modules OCaml et celui de Why3.

Concept	OCaml	Why3

Interface	`module type S = sig`
	`  ...`
	`end`
		`module S =`
		`  ...`
		`end`

Foncteur	`module F(X:S) = struct`
	`  ...`
	`end`
		`module F`
		`  use S as X`
		`  ...`
		`end`

Implémentation	`module (A:S) = struct`
	`  ...`
	`end`
		`module A`
		`  ...`
		`  clone S with ...`
		`end`

Instanciation	`module Main = F(A)`	`module Main`
		`  clone F with X = A`
		`end`

FIGURE 4.4 – Comparaison des systèmes de modules OCaml et Why3.

Dès l'introduction, nous avons séparé les conditions de vérification qu'impose le principe de substitution de Liksov, en deux catégories : la première catégorie concerne tout ce qui relève du contenu logique, préservation des invariants, le sous-typage des contrats de fonctions, etc ; la deuxième catégorie concerne tout ce qui relève des propriétés de sûreté assurés par le système du typage. Concrètement, dans ce chapitre,

nous avons considéré comme une telle propriété le contrôle statique des alias. Il est intéressant de remarquer que l'égalité polymorphe constitue un autre cas subtil pour le raffinement où l'implémentation peut casser la barrière d'abstraction, comme l'explique Appel dans [2]. En effet, l'utilisation d'égalité polymorphe, lorsqu'elle est appliquée à des types que l'on ne peut par comparer structurellement, produit une erreur à l'exécution, sans que cela soit détecté par le typage. Illustrons ce propos en OCaml. Déclarons une interface `A` de la manière suivante :

```
module type A = sig
  type t 'a
  val create   : 'a   → 'a t
  val equal    : 'a t → 'a t → bool
end
```

Supposons maintenant que l'on a écrit un client très simple :

```
module Client (X: A) = struct
  let v = X.create (fun x → x)
  let b = X.equal v v
end
```

Ensuite, donnons le module d'implémentation de l'interface `A` suivante :

```
module B : A = struct

  type 'a t = 'a option
  exception Empty

  let get x = match x with
    | None → raise Empty
    | Some x  → x

  let equal x y = get x = get y
  let create x = Some x
end
```

Comme on le voit, ci-dessus, la fonction d'égalité `equal` est réalisée à l'aide de l'égalité polymorphe. Maintenant, on peut donc relier le code du client avec l'implémentation :

```
  module ClientB = Client(B)
```

Or, bien que le typage accepte cette instanciation, elle produit une erreur, puisque l'égalité structurelle ne s'applique pas aux fonctions : l'égalité polymorphe peut donc briser la barrière d'abstraction. Bien que nous avons écrit l'exemple en OCaml, l'exemple reste pertinent pour nous : nous devrions étudier comment restreindre l'usage de l'égalité polymorphe dans le même esprit que nous avons étudié comme restreindre l'usage des régions introduites par le raffinement.

Bibliographie

[1] Jean-Raymond Abrial, *The B-book, assigning programs to meaning*, Cambridge University Press, 1996.

[2] Andrew W. Appel, *A critique of Standard ML.*, Tech. Report CS-TR-364-92, Princeton University, 1992.

[3] Cliff B. Jones, *Systematic software development using VDM (2nd ed.)*, Prentice-Hall, 1990.

[4] Jason Koenig and K. Rustan M. Leino, *Programming language features for refinement*, Proceedings 17th International Workshop on Refinement, Refine@FM 2015, Oslo, Norway, 22nd June 2015., 2015, pp. 87–106.

[5] K. Rustan M. Leino and Valentin Wüstholz, *The Dafny integrated development environment*, Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014. (Catherine Dubois, Dimitra Giannakopoulou, and Dominique Méry, eds.), Electronic Proceedings in Theoretical Computer Science, vol. 149, 2014, pp. 3–15.

[6] Xavier Leroy, *A modular module system*, Journal of Functional Programming **10** (2000), no. 3, 269–303.

[7] Barbara H. Liskov and Jeannette M. Wing, *A behavioral notion of subtyping*, ACM Transactions on Programming Languages and Systems **16** (1994), no. 6, 1811–1841.

[8] Jim Woodcock and Jim Davies, *Using Z : Specification, refinement, and proof*, Prentice-Hall, Inc., 1996.

5 Conclusion

Dans cette thèse nous avons exploré des solutions qu'une approche à base de systèmes de types peut apporter à la vérification des programmes. Concrètement, nous avons étudié trois aspects du langage de programmation WhyML, à savoir le *code fantôme*, le *contrôle statique des alias* et le *raffinement des données*. Pour chacune de ces trois notions, nous avons d'abord présenté, d'une manière informelle, les idées clés de l'approche, puis nous l'avons formalisée afin de montrer sa correction.

5.1 Choix et limitations

Précisons que notre but était de modéliser non pas le langage WhyML dans son intégralité, mais plutôt ces trois aspects pris séparément. Avec cet objectif en ligne de mire, nous avons fait le choix de circonscrire, pour chacune des trois notions, un fragment de WhyML que nous avons jugé pertinent au sens que nous expliquons dans les paragraphes ci-dessous.

D'une part, les fragments formalisés devaient refléter aussi bien les idées clés que les points subtils des solutions implantées dans l'outil Why3. Ainsi, pour le code fantôme, le formalisme GhostML reflète l'expressivité avec laquelle WhyML établit où le code fantôme a le droit d'apparaître et ce qui peut apparaître dans le code fantôme. De même, dans le chapitre sur les régions, le formalisme RegML contient l'essentiel du contrôle statique des alias tel qu'il est assuré par le typage en WhyML.

D'autre part, ces fragments devaient être raisonnablement petits. Ce dernier point mérite une discussion à part sur la manière dont on formalise aujourd'hui les concepts des langages de programmation. En effet, les formalisations de ce genre se font aujourd'hui de plus en plus non sur papier, mais d'une manière mécanisée, typiquement à l'aide d'un assistant de preuve comme Coq ou Isabelle/HOL. Non seulement la preuve mécanisée apporte un degré de confiance inégalé, mais elle devient inévitable lorsque l'on formalise un langage de programmation dans son intégralité. En effet, le nombre de cas à considérer, y compris des cas de base qui demandent un raisonnement simple et répétitif, devient alors tellement important qu'il n'est pratiquement plus possible de les écrire sur papier et encore moins de convaincre le lecteur que les raisonnements sont corrects. Néanmoins, lorsque l'on formalise un aspect du langage en particulier, il est souvent possible d'en isoler un fragment qui, tout en restant pertinent, est suffisamment

petit pour qu'il puisse être formalisé sur papier d'une façon claire et compréhensible et que les raisonnements menés convainquent le lecteur. Par ailleurs, alors qu'une preuve mécanisée offre de nombreux avantages par rapport à une preuve papier, elle requiert en contrepartie un temps de développement non négligeable et supérieur à celui que demande une preuve sur papier. Notre choix de présenter les formalismes dans un style « papier-crayon » nous paraît donc argumenté, d'une part, par l'adéquation des fragments choisis, et d'autre part, par les contraintes du temps pour mener notre travail à son terme.

Enfin, précisons que les simplifications qui éloignent les fragments formalisés de ce qui est implanté dans WhyML pour le code fantôme et les régions sont discutées dans les deux cas dans la sous-section « Implémentation » des chapitres correspondants où nous expliquons comment le formalisme pourrait être étendu pour tenir compte des traits du langage tels que le polymorphisme des types, les exceptions, les types récursifs, etc.

5.2 Perspectives

Nous voyons trois directions pour continuer le travail entrepris dans cette thèse.

Premièrement, on pourrait compléter le chapitre sur le raffinement de données, en montrant la correction du raffinement des modules, c'est-à-dire que le code du client dans lequel la réalisation est substituée à l'abstraction, est correct par construction, si la réalisation est un raffinement de l'abstraction et si le client écrit en premier lieu était lui-même correct. Pour que ce résultat ait un sens, il faudrait au préalable définir formellement la notion d'invariant de types et celle de contrats de fonctions. Par ailleurs, la question de savoir comment les invariants de type doivent être préservés par le raffinement est intéressante en soi. On pourrait suggérer par exemple que tant que la validité d'un invariant est assurée à l'entrée et à la sortie des fonctions exposées dans l'interface, l'implémentation aurait le droit de briser un tel invariant temporairement à de façon interne.

Deuxièmement, il serait intéressant de formaliser la manière dont le calcul de plus faibles préconditions est implanté en Why3 et montrer sa correction. En effet, comme nous l'avons vu dans le chapitre sur les régions, ce calcul s'appuie sur les informations fournies par le typage pour générer des conditions de vérification simples même en présence d'aliasing, mais cela nécessite de modifier les règles de la logique de Hoare d'une manière non-triviale pour refléter correctement modifications survenues lors des opérations d'affectation et des appels de fonctions.

Enfin, il serait intéressant d'étudier comment on pourrait relâcher les restrictions imposées par le système de types avec régions que nous avons présenté. En effet, dans notre approche les structures données telles que des tableaux ou des listes contenant de références peuvent être spécifiées, mais ne peuvent pas être implémentées. Actuellement, la seule manière de vérifier avec Why3 les programmes manipulant des structures

de données avec des pointeurs arbitraires est d'utiliser un modèle mémoire explicite, comme le fait le plug-in Jessie pour Frama-C. Néanmoins, il serait intéressant d'étudier comment on pourrait changer le système de type lui-même pour lever au moins une partie de ces restrictions.

A Étude de cas : Tables de hachage « Coucou »

A.1 Introduction

Le trait caractéristique de toute méthode de hachage est la manière dont elle gère la collision des clés, c'est-à-dire le cas où la fonction de hachage attribue à des clés différentes la même case de la table de hachage. Les solutions les plus connues au problème des collisions consistent à utiliser soit la méthode de chaînage en associant à chaque case une liste de clés qui ont la même image par la fonction de hachage, soit la méthode d'adressage ouvert en stockant les valeurs de hachage dans des cases contiguës.

Contrairement à ces méthodes, l'approche connue dans la littérature anglophone sous le nom de « cuckoo hashing », introduite par Pagh et Rodler [1], gère les collisions en mettant en jeu plusieurs tables de hachage simultanément.

Dans cette annexe, nous illustrons comment le système de types avec régions que nous avons présenté dans le chapitre 3 permet de vérifier une implémentation de « cuckoo hashing ».

Notre étude de cas est organisée de la manière suivante. Nous commençons par décrire brièvement l'approche, en suivant la présentation de Pagh et Rodler. Puis, nous présentons notre réalisation de cette approche en Why3 en donnant le code et la spécification de chaque fonction. Enfin, nous analysons notre implémentation du point de vue du typage et expliquons en quoi elle présente un cas d'étude intéressant et non trivial pour le système de types en question.

A.2 Présentation de l'approche

Dans la présentation de Pagh et Rodler, une table de hachage H est matérialisée par deux tableaux T_1 et T_2 auxquels sont associées respectivement deux fonctions de hachage h_1 et h_2. Les deux tableaux ont la même taille ℓ. Pour chaque élément x appartenant à H, on suppose qu'il est stocké soit dans la case $T_1[h_1(x)]$, soit dans la case $T_2[h_2(x)]$, mais pas dans les deux à la fois. La conséquence immédiate de cet

invariant est que le test d'appartenance se fait en temps constant. De la même manière, la suppression d'un élément est faite en temps constant, si l'on ignore le coût possible de amincissement des tableaux au cas où ils deviennent épars.

Supposons maintenant que l'on veuille ajouter un nouvel élément x dans la table de hachage H. Considérons d'abord le cas où l'ajout ne nécessite pas de rédimensionnement des tableaux T_1 et T_2, c'est-à-dire que, en choisissant un entier μ pour fixer la charge maximale de remplissage, on a :

$$1 + \text{nombre d'éléments de } H \leq \frac{\ell}{\mu}$$

L'insertion se passe de la façon suivante : on commence par regarder si la case $T_1[h_1(x)]$ est disponible, auquel cas il suffit de faire l'affectation $T_1[h_1(x)] \leftarrow x$. Sinon, $T_1[h_1(x)]$ correspond à un certain élément y différent de x (car on a supposé que x était un nouvel élément vis-à-vis de H) et avec lequel x rentre donc en collision. Dans ce cas-là, on effectue toujours l'affectation $T_1[h_1(x)] \leftarrow x$, mais en « dénichant » cette fois l'élément y (d'où l'image de « coucou » de l'approche).

Bien entendu, on doit remettre l'élément y dans H et c'est là que l'on utilise le tableau T_2, en ramenant le problème de l'insertion de x dans le tableau T_1 à celui de l'insertion de y dans le tableau T_2. Or, la case $T_2[h_2(y)]$ peut être elle-même occupée par un autre élément z qui, délogé par y, doit à son tour être réinséré à la table H dans le tableau T_1, et ainsi de suite, jusqu'à ce que chaque élément trouve son nid. Bien entendu, ce processus migratoire peut ne pas terminer. Le dessin de la figure A.1 illustre un exemple simple d'une telle situation. Comme on le voit sur le dessin, si l'on

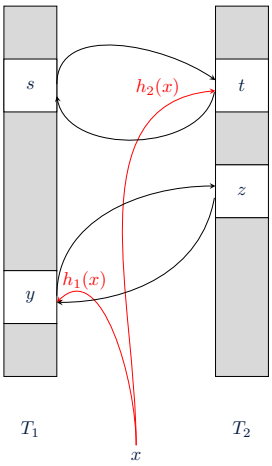


FIGURE A.1 – L'insertion d'un élément engendrant un cycle.

suppose que les éléments y, z, s, t, qui appartiennent à la table H, vérifient les égalités suivantes :

$$h_1(y) = h_1(z), \qquad h_1(s) = h_1(t)$$
$$h_2(y) = h_2(z), \qquad h_2(s) = h_2(t),$$

alors, si x vérifie les égalités $h_1(x) = h_1(y)$ et $h_2(x) = h_2(t)$, l'insertion de x dans H ne termine pas. Notons que dans l'exemple de la figure A.1 les deux cycles sont de la même longueur égale à trois. En général, de tels cycles peuvent, bien sûr, avoir des longueurs différentes et aussi longues que le nombre d'éléments stockés dans H.

Pour éviter qu'un tel cycle se produise pour une paire de fonctions de hachage donnée une solution possible est de borner le nombre maximal d'itérations par une constante `cutoff` laquelle, une fois atteinte, requiert un remplacement des fonctions de hachage h_1 et h_2 par une nouvelle paire de fonctions h'_1, h'_2 pour ensuite redémarrer le processus d'insertion. Une autre solution serait de remplacer les fonctions de hachage, non pas en bornant le nombre maximal d'itérations, mais en détectant la présence d'un cycle engendré par l'insertion.

Néanmoins il reste toujours possible que, même pour les nouvelles fonctions de hachage h'_1 et h'_2, l'insertion engendre un nouveau cycle (ou qu'elle atteigne simplement la constante `cutoff` selon l'approche choisie). En général, l'insertion d'un élément dans la table peut, au moins du point de vue théorique, échouer dans l'approche « cuckoo hashing ». Nous invitons le lecteur à consulter l'article de Pagh et Rodler qui expliquent les raisons pour lesquelles cette situation est peu probable et, plus généralement, qui montrent que « cuckoo hashing » est une méthode de hachage très efficace de point de vue de complexité des opérations de base.

Enfin, il se peut que l'ajout de x dans H nécessite de redimensionner les tableaux T_1 et T_2. Dans ce cas-là, le redimensionnement peut être fait en temps constant amorti par rapport à l'opération d'ajout, en doublant la taille des tableaux à chaque redimensionnement. Nous présentons maintenant une implémentation possible de cette approche en Why3 [1].

1. Le lecteur peut trouver l'intégralité du code sur le lien suivant :
https://www.lri.fr/~gondelman/cuckoo_hashing.mlw

A.3 Implémentation

Dans un premier temps nous présentons le code et la spécification des fonctions de notre implémentation pour donner ensuite quelques détails sur la manière dont nous avons mené la preuve.

A.3.1 Préliminaires : prédicat d'égalité, fonctions de hachage

Avant de définir le type des tables de hachage, il nous faut convenir sur la nature des éléments stockés. Nous supposons disposer pour cela d'un type abstrait `elt`

```
type elt
```

sur lequel on déclare un prédicat d'égalité `eq` ainsi qu'une fonction `hash` :

```
val predicate eq (x y: elt) : bool

val function hash (vers: int) (x: elt) : int
  ensures { 0 ≤ result }
```

Notons bien que `hash` représente non pas *une* fonction de hachage, mais une *famille* de fonctions. En effet, le symbole fonctionnel `hash` est paramétré par un entier `vers` que l'on appelle ici « le numéro de version » et qui permet de fixer parmi un ensemble de fonctions une fonction en particulier. De manière similaire, la fonction `bucket` ci-dessous est paramétrée par un entier `vers` pour calculer l'indice d'un élément `e` à l'aide de la fonction de hachage `hash vers e` :

```
let function bucket (e: elt) (vers n : int) : int
  requires { 0 < n }
  ensures  { 0 ≤ result < n }
= mod (hash vers e) n
```

Pour simplifier la preuve et la présentation, nous supposons dans la suite que le symbol d'égalité `eq` se comporte exactement comme l'égalité mathématique, c'est-à-dire qu'il est déclaré avec la post-condition suivante :

```
val predicate eq (x y: elt) : bool
  ensures { result ↔ x = y }
```

D'autre part, pour distinguer les cases occupées par un élément des cases vides, nous utilisons le type option `option elt` de sorte qu'une case vide est représentée par la valeur `None` et une case occupée par un élément `e` par la valeur `Some e`. Adaptons la relation d'égalité `eq` pour le type `option elt` :

```
let predicate eqo (x y: option elt)
  ensures { result ↔  x = y }
```

```
= match (x, y) with
    | None, None       → true
    | Some x1, Some y1 → eq x1 y1
    | _                → false
  end
```

A.3.2 Définition du type des tables de hachage

Nous pouvons maintenant introduire le type des tables de hachage. Le type des tables de hachage correspond au type enregistrement `t` suivant :

```
type t = { mutable data1: array (option elt);
           mutable vers1: int;
           mutable data2: array (option elt);
           mutable vers2: int;
     ghost mutable view : set elt;
           mutable size : int; }
```

Les champs `data1` et `data2` correspondent aux tableaux T_1 et T_2 respectivement. Le champ `view` contient le modèle logique d'une table de hachage vu en tant qu'ensemble d'éléments. Le champ `vers1` fixe dans la famille des fonctions de hachage la fonction de hachage h_1 associée au tableau T_1. Symétriquement, le champ `vers2` fixe la fonction h_2 pour le tableau T_2.

Les contraintes sur les tables de hachage que l'on a vues dans la section précédente (les deux tableaux ont la même taille, leur contenu sont disjoints, etc.) peuvent être exprimées en tant qu'un invariant de type que nous allons maintenant expliciter. Pour cela, introduisons quelques définitions auxiliaires. À l'aide du prédicat `eqo`, définissons deux prédicats `in_data1 (e: elt) (h: t)` et `in_data2 (e: elt) (h: t)` qui caractérisent la relation d'appartenance d'un élément `e` au tableau `h.data1` ou au tableau `h.data2` respectivement :

```
let predicate in_data1 (e: elt) (h: t)
  requires { 0 < h.data1.length }
= eqo (Some e) h.data1[bucket e h.vers1 h.data1.length]

let predicate in_data2 (e: elt) (h: t)
  requires { 0 < h.data2.length }
=  eqo (Some e) h.data2[bucket e h.vers2 h.data2.length]
```

Avec ces prédicats, l'appartenance d'un élément à une table de hachage s'exprime directement par la disjonction suivante :

```
let predicate in_data (e: elt) (h: t)
  requires { 0 < h.data1.length = h.data2.length }
= in_data1 e h || in_data2 e h
```

Par ailleurs, introduisons la définition suivante pour exprimer le fait que deux tableaux n'ont pas d'éléments en commun :

```
predicate disjoint (s t: array (option elt)) =
  ∀ i e. 0 ≤ i < s.length → s[i] = Some e →
    ∀ j. 0 ≤ j < t.length → not eqo s[i] t[j]
```

L'invariant du type `t` correspond alors au prédicat `inv h` défini comme la conjonction suivante :

```
predicate inv (h: t) =
  (*C1*)    0 < h.data1.length = h.data2.length ∧
  (*C2*)    0 ≤ h.size = h.view.cardinal  ∧
  (*C3*)    (∀ e. mem e h.view → in_data e h) ∧
  (*C4*)    (∀ e. in_data e h  → mem e h.view) ∧
  (*C5*)    (∀ i, e. 0 ≤ i < h.data1.length →
              h.data1[i] = Some e →
                i = bucket e h.vers1 h.data1.length) ∧
  (*C6*)    (∀ i, e. 0 ≤ i < h.data2.length →
              h.data2[i] = Some e →
                i = bucket e h.vers2 h.data2.length) ∧
  (*C7*)    disjoint h.data1 h.data2 ∧
  (*C8*)    h.vers2 < 0 < h.vers1 ∧
  (*C9*)    max_load * h.size ≤ h.data1.length
```

La première clause impose que la taille des deux tableaux soit la même et qu'elle soit strictement positive (ce qui est nécessaire pour que le calcul de la case d'un élément dans la fonction `bucket` soit correct). L'égalité `h.size = h.view.cardinal` exprime le fait que l'entier stocké dans le champ `size` correspond exactement au nombre des éléments contenus dans le modèle logique.

La troisième et la quatrième clauses garantissent ensemble la cohérence de représentation entre les tableaux stockés dans les champs `h.data1` et `h.data2` et le modèle logique stocké dans le champ `h.view` : un élément doit appartenir à celui-ci si et seulement s'il appartient à l'un des deux tableaux.

Les deux clauses suivantes postulent que dans les tableaux `h.data1` et `h.data2` un élément est toujours stocké dans la bonne case, c'est-à-dire dans la case calculée à l'aide des fonctions de hachage utilisant les numéros de version courants `h.vers1` et `h.vers2` respectivement. En particulier ces deux clauses impliquent qu'aucun des deux tableaux ne contient de doublons.

Le fait que les fonctions pour `h.data1` sont toujours paramétrées par un entier strictement positif et celles pour `h.data2` par un entier strictement négatif (huitième clause) est une façon de garantir que les fonctions de hachage pour le premier et le second tableau peuvent être distinguées par celui qui les fournit. Enfin, en ce qui concerne la dernière clause, on suppose avoir introduit au préalable une constante `max_load` strictement positive :

```
val constant max_load
axiom max_load_pos: 1 ≤ max_load
```

Cette constante correspond à la charge maximale de la table. Passons maintenant à l'implémentation des fonctions permettant de manipuler ces tables.

A.3.3 Opération de création et test d'appartenance

Pour la création d'une nouvelle table de hachage, il suffit de donner sa taille initiale ainsi que les numéros de version des fonctions de hachage :

```
let create (n: int) (v1 v2: int) : t
  requires { 0 < n }
  requires { v2 < 0 < v1 }
  ensures  { inv result }
  ensures  { result.data1.length = n }
  ensures  { result.view = empty }
 = { data1 = Array.make n None;  vers1 = v1;
     data2 = Array.make n None;  vers2 = v2;
     view  = empty;
     size  = 0; }
```

Quant à la fonction d'appartenance, elle peut être implémentée directement à l'aide du prédicat `in_data` :

```
let function mem (h: t) (e: elt) : bool
  requires { inv h }
  ensures  { result ↔ S.mem e h.view }
= in_data e h
```

La seule différence entre `mem h e` et `in_data e h` est la précondition `inv h` présente dans le contrat de `mem`.

A.3.4 Ajout d'un élément sans redimensionnement

Comme dans notre présentation de l'approche dans la section précédente A.2, commençons par le cas où l'on n'a pas besoin de redimensionner les tableaux pour ajouter un élément dans la table de hachage. Tout d'abord, fixons un nombre maximal d'itérations engendrées par l'insertion d'un élément donné avant que l'on ne change des fonctions de hachage :

```
val constant cutoff : int
axiom cutoff_pos: 1 ≤ cutoff
```

L'opération d'ajout est alors réalisée à l'aide d'une fonction récursive `insert` suivante :

```
let rec insert (h: t) (e: elt) (step: int) (b: bool) : unit
   requires { 0 ≤ step ≤ cutoff }
   requires { not (S.mem e h.view) }
   requires { max_load * (h.size + 1) ≤ h.data1.length }
   requires { inv h }
   ensures  { inv h }
   ensures  { h.view = S.add e (old h).view }
   ensures  { h.data1.length = (old h).data1.length }
   diverges (* la terminaison n'est pas assurée *)
 = ...
```

Le but de la fonction `insert` est d'insérer un nouvel élément `e` dans la table `h`. Le fait que `e` est un nouvel élément est exprimé par la précondition `not (S.mem e h.view)`. Comme on voit, la fonction `insert` prend deux arguments supplémentaires, à savoir, un entier `step` et un booléen `b`. En ce qui concerne ce dernier, il indique dans lequel des deux tableaux de `h` l'élément `e` doit être inséré : lorsque `b = true`, l'élément `e` est ajouté dans le tableau `h.data1` et, sinon, il est ajouté dans le tableau `h.data2`. Quant à l'argument `step`, son rôle est explicité par la précondition `0 ≤ step ≤ cutoff`, c'est-à-dire que `step` sert donc à borner le nombre d'appels récursifs de `insert` par `cutoff` avant que les numéros de version `h.vers1` et `h.vers2` ne soient changés. Notons que la clause `inv h` apparaît à la fois dans la pré- et post-conditions de `insert`, c'est-à-dire que celle-ci doit préserver l'invariant de type de la table `h`.

Notons également la présence du mot clé `diverges`. Ce mot signifie que l'on ne requiert pas la preuve de terminaison pour la fonction `insert`. Cela reflète le fait que, en général, la fonction `insert` peut ne pas terminer (quels que soient les numéros de version `h.vers1` et `h.vers2`).

Comme nous allons voir, il peut arriver que les tableaux stockés dans les champs `h.data1` et `h.data2` soient remplacés par d'autres tableaux, c'est-à-dire qu'il peut arriver que `h.data1` et `(old h).data1` (respectivement, `h.data2` et `(old h).data2`) soient deux tableaux différents. Or, puisque l'on considère ici le cas où l'on ne redimensionne pas les tableaux de `h`, la dernière post-condition est nécessaire, car elle exprime le fait que la taille des tableaux stockés dans `h.data1` et `h.data2` ne change pas dans tous les cas.

Regardons maintenant le code de `insert`. On commence par tester si la valeur `cutoff` est atteinte :

```
 = let step =
     if step = cutoff then
       then begin change_hash h; 0 end
       else step + 1 in
   ...
```

Si c'est le cas, on change les numéros de version en appelant la fonction `change_hash` (que l'on présentera dans un instant) et on remet le compteur `step` à zéro. On calcule

ensuite l'indice `i` de l'élément `e`, puis on regarde le contenu de la case dans laquelle on va mettre l'élément `e`, en distinguant le cas où cette case est vide et le cas où est elle déjà occupée par un « oisillon » :

```
let i = index_of h e b in
let y = if b then h.data1[i] else h.data2[i] in
match y with
    ...
end
```

Lorsque la case est vide, on y met directement la valeur `Some e`, sans oublier de mettre à jour le modèle logique `h.view` et l'entier `h.size` :

```
| None → h.size ← h.size + 1;
         h.view ← S.add e h.view;
         if b then h.data1[i] ← Some e else h.data2[i] ← Some e;
```

Sinon, lorsque la case en question est déjà occupée par un autre élément, le code est un peu plus complexe :

```
| Some x → h.view ← S.add e (S.remove x h.view);
           if b then h.data1[i] ← Some e else h.data2[i] ← Some e;
           insert h x step (not b)
```

En effet, puisque l'on remplace l'élément `x` par `e`, il faut non seulement ajouter `e` au modèle logique mais également en retirer `x`. Par ailleurs, il faut réinsérer `x` dans la table `h`, ce qui l'on fait en appelant récursivement `insert`. Notons que la valeur du booléen `b` est alors inversée : si l'on ajoute `e` dans le tableau `h.data1` (resp. `h.data2`), l'élément déniché `x` sera donc ajouté au tableau `h.data2` (resp. `h.data1`). Cela correspond donc bien à l'idée « coucou » de l'approche.

A.3.5 Changement des fonctions de hachage

Lorsque le nombre d'itérations d'insertion atteint le seuil autorisé, c'est-à-dire quand `step = cutoff`, on doit changer les fonctions de hachage, ce qui dans notre implémentation revient à changer les numéros de version. Or, on ne peut pas simplement mettre à jour les champs `h.vers1` et `h.vers2` correspondants, car on doit également recalculer les indices des éléments stockés dans les tableaux `h.data1` et `h.data2` en fonction de leurs nouvelles valeurs.

Une manière simple de procéder est d'introduire une table auxiliaire `g` avec deux nouvelles fonctions de hachage, puis y recopier les éléments de la table `h`. De cette manière-là, on peut ensuite utiliser `g` pour remplacer les tableaux contenus dans les champs `h.data1` et `h.data2` par les tableaux `g.data1` et `g.data2` respectivement. Ces trois étapes sont réalisées dans la fonction `change_hash` ci-dessous de la manière suivante :

```
 with change_hash (h: t) : unit
   requires { inv h }
   ensures  { inv h }
   ensures  { h.view = old h.view }
   ensures  { h.data1.length = (old h).data1.length }
   diverges
 = let g = create (Array.length h.data1) (h.vers1 + 1) (h.vers2 - 1)
   in insert_all h g;
   ( h.data1, h.data2, h.vers1, h.vers2, h.view ) ←
   ( g.data1, g.data2, g.vers1, g.vers2, g.view )
```

Le mot clé `with` indique que `change_hash` est mutuellement récursive avec la fonction `insert`. Remarquons d'abord que les numéros de version sont mis à jour, en respectant la convention que `h.vers1` et `h.vers2` restent respectivement strictement positif et strictement négatif. Remarquons également que dans l'affectation parallèle, il est possible que les numéros de version `g.vers1` et `g.vers2` ne correspondent plus à leurs valeurs initiales `(h.vers1 + 1)` et `(h.vers2 - 1)`, de la même manière que les tableaux stockés dans les champs `g.data1` et `g.data2` peuvent être différents de ceux qui ont été alloués lors de la création de l'enregistrement `g`. Pour comprendre comment cela peut arriver, regardons la manière dont la fonction `insert_all h g` est implémentée :

```
 with insert_all (h g: t) : unit
    requires { h.data1.length ≤ g.data1.length  }
    requires { g.view = empty }
    requires { inv h ∧ inv g }
    ensures  { inv h ∧ inv g }
    ensures  { h.view = g.view }
    ensures  { h.data1.length = (old h).data1.length }
    ensures  { g.data1.length = (old g).data1.length }
    diverges
 = let n = h.data1.length in
    for i = 0 to n - 1 do
      match h.data1[i] with
        | None   → ()
        | Some e → insert g e 0 true;
      end
    done;
    for i = 0 to n - 1 do
      match h.data2[i] with
        | None   → ()
        | Some e → insert g e 0 false
      end;
    done
```

Dans le code ci-dessus, on parcourt chacun des deux tableaux `h.data1` et `h.data2`. À la `i`-ième étape de chaque parcours, on regarde si la case du tableau en question est non vide, auquel cas on place l'élément `e` trouvé dans l'enregistrement `g`, en faisant appel à la fonction `insert`. Or, chacun des deux appels à `insert` peut à son tour appeler la fonction `change_hash` si la mise à jour des fonctions de hachage dans `g` ne permet pas de résoudre le problème survenu en premier lieu. En conséquence, il est possible qu'après le premier parcours, à l'intérieur même de la fonction `insert_all`, les numéros de version `g.vers1` et `g.vers2` ainsi que les tableaux stockés dans les champs `g.data1` et `g.data2` ne correspondent plus à leurs valeurs initiales.

Notons par ailleurs que dans le premier parcours, l'appel à la fonction `insert` a la forme `insert g e 0 true`, c'est-à-dire que dans le premier cas on insère l'élément `e` dans le tableau `g.data1`, alors que dans le second cas on l'insère dans le tableau `g.data2`. Cela est fait par symétrie, en supposant que si les fonctions de hachage sont bonnes, le premier parcours n'engendrera pas beaucoup de collisions, c'est-à-dire qu'au second parcours le tableau `g.data2` sera pratiquement vide et pourra être rempli efficacement. Néanmoins, il faut bien tenir en compte qu'il est tout à fait possible que des collisions se produisent lors du premier parcours et que, par conséquent, l'on ne peut pas supposer qu'au début du second parcours, le tableau `g.data2` est vide.

A.3.6 Redimensionnement

Lorsque la charge maximale autorisée est dépassée, c'est-à-dire lorsqu'on a l'inégalité stricte `h.data1.length < max_load * h.size`, il est nécessaire d'agrandir la taille des tableaux. Pour faire cela, nous utilisons la fonction `resize` spécifiée de la manière suivante :

```
let resize (h: t) (m: int) : unit
  requires { 2 ≤ m }
  requires { inv h }
  ensures  { inv h }
  ensures  { h.view = (old h).view }
  ensures  { h.data1.length = m * (old h).data1.length }
  diverges
= ...
```

Comme les autres fonctions, `resize` préserve donc l'invariant du type de la table `h`. Par ailleurs, elle ne doit pas changer le contenu de `h`, ce qui est exprimé par l'égalité entre le modèle logique de `h` avant et après l'exécution. On doit également garantir que la taille des deux tableaux est maintenant `m` fois plus grande que leur taille initiale. En revanche, on n'affirme pas que les numéros de version `h.vers1` et `h.vers2` restent les mêmes. Pour comprendre pourquoi nous ne pouvons pas garantir cela, regardons la définition de `resize` :

```
let g = create (m * h.data1.length) h.vers1 h.vers2 in
```

```
    insert_all h g;
    ( h.data1, h.data2, h.vers1, h.vers2, h.view ) ←
    ( g.data1, g.data2, g.vers1, g.vers2, g.view )
```

Comme on le voit, le code `resize` est très similaire à celui de `change_hash`, à ceci près que dans l'introduction de la table auxiliaire `g` on change la taille, au lieu de changer les numéros de version. Néanmoins, puisque l'on fait appel à la fonction `insert_all`, cela peut résulter en un appel à la fonction `change_hash`. Par conséquent, non seulement on ne peut pas supposer que les numéros de versions n'ont pas changé, mais, comme on le voit dans l'affectation parallèle ci-dessus, on doit remplacer les valeurs des champs `h.vers1` et `h.vers2` respectivement par les numéros de version `g.vers1` et `g.vers2`. Par ailleurs, cela explique aussi pourquoi le contrat de la fonction `resize` comporte la clause `diverges` (alors qu'elle n'est pas récursive). En effet, puisque `resize` fait appel à la fonction `insert_all`, qui peut ne pas terminer, `resize` peut ne pas terminer non plus.

A.3.7 Ajout d'un élément : cas général

Nous pouvons maintenant traiter le cas général de l'ajout d'un élément, qui est spécifié et implémenté de la façon suivante :

```
let add (e: elt) (h: t) : unit
  requires { inv h }
  ensures  { inv h }
  ensures  { h.view = S.add e (old h).view }
  diverges
= if not (mem h e) then begin
    if h.data1.length < max_load * (h.size + 1) then begin
      let m = if h.size = 0 then max_load else 2 in
      resize h m;
    end;
    insert h e 0 true
  end
```

Comme on le voit, on commence par vérifier si l'élément `e` est déjà dans la table `h`, auquel cas il n'y rien à faire. Sinon, on vérifie s'il est nécessaire d'agrandir la taille des tableaux, puis, dans les deux cas, on ajoute `e` dans `h` à l'aide de la fonction `insert`. Remarquons que l'on agrandit la taille de la table `h` toujours par un facteur 2, sauf si la table `h` est vide auquel cas le facteur est égal à la charge. Cela est dû au fait que lorsque la table est vide, la précondition `max_load * (h.size + 1)` $\leq$ `h.data1.length` dans le contrat de la fonction `insert` peut ne pas être satisfaite si la charge `max_load` est strictement plus grande que deux fois la taille du tableau `h.data1`.

A.3.8 Suppression d'un élément

Expliquons également la manière dont nous avons réalisée l'opération de suppression d'un élément dans la table de hachage à l'aide d'une fonction `remove`. Commençons par donner sa spécification :

```
let remove (e: elt) (h: t) : unit
  requires { inv h }
  ensures  { inv h }
  ensures  { h.view = S.remove e (old h).view }
= ...
```

À l'instar des autres opérations, `remove` doit préserver l'invariant de type de `h`. La post-condition `h.view = S.remove e (old h).view` exprime le fait que `remove` est en effet une opération de suppression de l'élément `e` dans `h`. Notons que, lorsque `e` n'appartient pas à la table `h`, cette post-condition va être trivialement vraie. Donnons maintenant le corps de la fonction `remove`. On commence par regarder si l'élément `e` se trouve dans le tableau `h.data1` :

```
= let i = index_of h e true in
    match h.data1[i] with
      | None  →
        let j = index_of h e false in
        match h.data2[j] with
          | None → ()
          | Some x → if eq x e then begin
                       h.size ← h.size - 1;
                       h.view ← S.remove e h.view;
                       h.data2[j] ← None
                     end
        end
      | Some x → ...
    end
```

On voit que, lorsque `e` n'appartient pas au `h.data1`, si la case `h.data2[j]` est vide ou bien si elle est occupée par un élément différent de `e` (au sens de l'égalité `eq`), il n'y rien à faire. Sinon, lorsque la case `h.data2[j]` contient `e`, on le supprime, en remplaçant la valeur `Some x` par `None`. Notons que dans ce cas-là il ne faut pas oublier de mettre à jour les champs `h.size` et `h.view`, sans quoi l'invariant de la table `h` ne sera pas préservé.

Sinon, lorsque la case `h.data1[i]` n'est pas vide, on commence par regarder si l'élément qui occupe la case `h.data1[i]` est égal à `e` auquel cas le raisonnement est similaire :

```
      | Some x →
```

```
if eq x e then begin
  h.size ← h.size - 1;
  h.view ← S.remove e h.view;
  h.data1[i] ← None;
  end
else ...
```

Sinon, on doit de nouveau regarder si l'élément `e` se trouve dans le deuxième tableau, ce qui reprend exactement les lignes du test `match h.data2[j] with ... end` ci-dessus (et que l'on pourrait donc factoriser en introduisant une fonction de suppression asymétrique retirant un élément `e` uniquement du tableau `h.data2`).

A.3.9 Preuve

L'implémentation que nous avons présentée ici utilise une version de Why3 en développement [2]. Pour prouver notre implémentation, nous avons dû instrumenter le code par des invariants de boucle et un certain nombre d'assertions. En revanche, toutes les obligations de preuve sont déchargées à l'aide des démonstrateurs automatiques. Au total, pour 142 lignes de code, l'implémentation contient 136 lignes du contenu logique dont 24 assertions et 11 invariants de boucle. Le temps total de la preuve est environ de 6 minutes et demie. [3]

A.4 Typage

Il est intéressant d'analyser notre implémentation du point de vue du système de types avec régions que l'on a présenté dans cette thèse.

Premièrement, remarquons que d'après la définition du type `t`, chaque table de hachage est reflétée statiquement par une région imbriquée de la forme

$$\rho = \{\, \mathtt{data}_1 : \rho_1, \mathtt{data}_2 : \rho_2, \mathtt{view} : \mathtt{set\ elt}, \mathtt{size} : \mathtt{Int}, \mathtt{vers}_1 : \mathtt{Int}, \mathtt{vers}_2 : \mathtt{Int} \,\}_r$$

où l'indice r, rappelons-le, permet de distinguer les régions de tables de hachage distinctes. On suppose dans la suite que les sous-régions ρ_1 et ρ_2, qui correspondent au type `array (option elt)`, sont distinctes. Par ailleurs, introduisons la notation

$$\rho\,\langle \rho_1, \rho_2 \rangle$$

2. Voir la branche "new_system", dans le dépôt git de Why3,
https://scm.gforge.inria.fr/anonscm/git/why3/why3.git

3. L'information détaillée sur les démonstrateurs utilisés et le temps de vérification pour chaque obligation de preuve peut être trouvée sur le lien suivant :
https://www.lri.fr/~gondelman/why3session.html

pour désigner que dans une région ρ, habitée par une certaine table de hachage, les champs $\texttt{data}_1$ et $\texttt{data}_2$ sont typés respectivement par les régions ρ_1 et ρ_2.

Deuxièmement, rappelons que dans notre système de types, toutes les régions fraîches faisant partie du résultat d'une fonction sont invalidées. Ainsi la fonction `create` possède la signature suivante :

$$\texttt{create} : \texttt{Int} \times \texttt{Int} \times \texttt{Int} \to \rho\,\langle\rho_1, \rho_2\rangle \cdot (\varnothing \cdot \{\rho, \rho_1, \rho_2\})$$

En ce qui concerne les signatures des fonctions `insert`, `change_hash` et `insert_all`, données ci-dessous, pour comprendre comment leurs effets sont calculés, il faut tenir compte du fait que leurs définitions sont mutuellement récursives :

$$\begin{array}{lll}
\texttt{insert} : & \rho\,\langle\rho_1, \rho_2\rangle \times \texttt{elt} \times \texttt{Int} \times \texttt{Bool} & \to \texttt{Unit} \cdot (\{\rho\} \cdot \{\rho_1, \rho_2\}) \\
\texttt{change_hash} : & \rho\,\langle\rho_1, \rho_2\rangle & \to \texttt{Unit} \cdot (\{\rho\} \cdot \{\rho_1, \rho_2\}) \\
\texttt{insert_all} : & \rho\,\langle\rho_1, \rho_2\rangle \times \rho'\,\langle\rho'_1, \rho'_2\rangle & \to \texttt{Unit} \cdot (\{\rho'\} \cdot \{\rho'_1, \rho'_2\})
\end{array}$$

En particulier, les ensembles des régions invalidées dans la signature de `insert` et de `insert_all` ne sont pas vides, car ces fonctions dépendent de `change_hash` dont le corps contient un conflit d'aliasing engendré par l'affectation parallèle

```
( h.data1, h.data2, h.vers1, h.vers2, h.view ) ←
( g.data1, g.data2, g.vers1, g.vers2, g.view )
```

Notons par ailleurs que le corps de `change_hash` contient également l'opération de création

```
create (Array.length h.data1) (h.vers1 + 1) (h.vers2 - 1)
```

qui engendre un effet d'invalidation des nouvelles régions ρ', ρ'_1, ρ'_2 (qui sont nécessairement distinctes des régions ρ, ρ_1, ρ_2, car sinon, on ne pourrait pas typer les deux lignes qui suivent cet appel à `create`). Cependant, puisque le type du résultat de `change_hash` est `Unit`, ces trois régions ne sont pas exposées dans la signature de la fonction. Le fait que la région ρ' apparaît dans les régions écrites de `insert_all` est important car il permet d'utiliser les champs `g.data1` et `g.data2` dans l'affectation parallèle.

Le typage des autres fonctions (`resize`, `add` et `remove`) suit le même raisonnement et nous ne le donnons pas ici.

Bibliographie

[1] Rasmus Pagh and Flemming Friche Rodler, *Cuckoo hashing*, Algorithms - ESA 2001, 9th Annual European Symposium, Aarhus, Denmark, August 28-31, 2001, Proceedings, 2001, pp. 121–133.

Résumé

Cette thèse se place dans le contexte de la vérification déductive des programmes et a pour objectif de formaliser un certain nombre de concepts qui sont mis en œuvre dans l'outil de vérification Why3. L'idée générale est d'explorer des solutions qu'une approche à base de systèmes de types peut apporter à la vérification.

Nous commençons par nous intéresser à la notion du *code fantôme*, une technique implantée dans de nombreux outils de vérification modernes, qui consiste à donner à des éléments de la spécification les apparences d'un code opérationnel. L'utilisation correcte du code fantôme requiert maintes précautions puisqu'il ne doit jamais interférer avec le reste du code. Le premier chapitre est consacré à une formalisation du code fantôme, en illustrant comment un système de types avec effets en permet une utilisation à la fois correcte et expressive.

Puis nous nous intéressons à la vérification des programmes manipulant des pointeurs. En présence d'*aliasing*, c'est-à-dire lorsque plusieurs pointeurs manipulés dans un programme dénotent une même case mémoire, la spécification et la vérification deviennent non triviales. Plutôt que de nous diriger vers des approches existantes qui abordent le problème d'aliasing dans toute sa complexité, mais sortent du cadre de la logique de Hoare, nous présentons un système de types avec effets et *régions* singletons qui permet d'effectuer un contrôle statique des alias avant même de générer les obligations de preuve. Bien que ce système de types nous limite à des pointeurs dont l'identité peut être connue statiquement, notre observation est qu'il convient à une grande majorité des programmes que l'on souhaite vérifier.

Enfin, nous abordons les questions liées à la vérification de programmes conçus de façon *modulaire*. Concrètement, nous nous intéressons à une situation où il existe une *barrière d'abstraction* entre le code de l'utilisateur et celui des bibliothèques dont il dépend. Cela signifie que les bibliothèques fournissent à l'utilisateur une énumération de fonctions et de structures de données manipulées, sans révéler les détails de leur implémentation. Le code de l'utilisateur ne peut alors exploiter ces données qu'à travers un ensemble de fonctions fournies. Dans une telle situation, la vérification peut elle-même être *modulaire*. Du côté de l'utilisateur, la vérification ne doit alors s'appuyer que sur des invariants de type et des contrats de fonctions exposés par les bibliothèques. Du côté de ces dernières, la vérification doit garantir que la représentation concrète *raffine* correctement les entités exposées, c'est-à-dire en préservant les invariants de types et les contrats de fonctions. Dans le troisième chapitre nous explorons comment un système de types permettant le contrôle statique des alias peut être adapté à la vérification modulaire et le raffinement des structures de données.

Nous terminons notre travail sur une étude de cas où nous illustrons comment les solutions proposées dans les trois chapitres de la thèse peuvent être combinées pour vérifier une implémentation non triviale de table de hachage appelée *cuckoo hashing*.

Abstract

This thesis is conducted in the framework of deductive software verification. is aims to formalize some concepts that are implemented in the verification tool Why3. The main idea is to explore solutions that a type system based approach can bring to deductive verification.

First, we focus our attention on the notion of *ghost code*, a technique that is used in most of modern verification tools and which consists in giving to some parts of specification the appearance of operational code. Using ghost code correctly requires various precautions since the ghost code must never interfere with the operational code. The first chapter presents a type system with effects illustrating how ghost code can be used in a way which is both correct and expressive.

The second chapter addresses some questions related to verification of programs with pointers in the presence of *aliasing*, *i.e.* when several pointers handled by a program denote a same memory cell. Rather than moving towards to approaches that address the problem in all its complexity to the costs of abandoning the framework of Hoare logic, we present a type system with effects and singleton regions which resolves aliasing issues by performing a static control of aliases even before the proof obligations are generated. Although our system is limited to pointers whose identity must be known statically, we observe that it fits for most of the code we want to verify.

Finally, we focus our attention on a situation where there exists an *abstraction barrier* between the user's code and the one of the libraries which it depends on. That means that libraries provide the user a set of functions and of data structures, without revealing details of their implementation. When programs are developed in a such modular way, verification must be modular itself. It means that the verification of user's code must take into account only function contracts supplied by libraries while the verification of libraries must ensure that their implementations *refine* correctly the exposed entities. The third chapter extends the system presented in the previous chapter with these concepts of modularity and data refinement.

We conclude our work by a case study in which we illustrate how solutions exposed in this thesis can be combined for verifying a non trivial implementation of hash tables called "cuckoo hashing".