THÈSE

présentée à

l'Université Paris 7 – Denis Diderot
École doctorale Sciences Mathématiques de Paris Centre

pour obtenir le titre de

Docteur en Informatique

Auteur

ZAYNAH DARGAYE

Vérification formelle d'un compilateur optimisant pour langages fonctionnels

Thèse dirigée par

Xavier Leroy

Soutenue le

06 Juillet 2009

JURY

Rapporteurs	Catherine Dubois Andrew Tolmach
Examinateurs	Pierre Crégut Roberto Di Cosmo Marc Pantel Christine Paulin-Mohring
Directeur	Xavier Leroy

Table des matières

1 Introduction

La programmation, d'après le *Nouveau Petit Robert*, recouvre « l'élaboration et la codification de la suite d'opérations formant un programme » ; un programme, aussi appelé un logiciel, étant « l'ensemble des instructions, rédigé dans un langage de programmation, permettant à un système informatique d'exécuter une tâche donnée ». Cette définition souligne le rôle essentiel du langage de programmation comme vecteur de communication entre le programmeur et l'ordinateur. Cependant, elle ne fait pas apparaître la grande diversité des nombreux langages de programmation existants. Un seul de ces langages est directement exécutable par les circuits de l'ordinateur : il s'agit du *langage machine* lorsqu'il se présente sous forme de nombres (les «0 et 1» bien connus du grand public), ou encore du *langage assembleur* lorsqu'il se présente sous forme textuelle. Ces langages machine ou assembleur sont dits «de bas niveau», car les ordres exprimés dans ces langages manipulent directement les composants de l'ordinateur que sont les registres, la mémoire et les périphériques.

Pour faciliter la programmation, de nombreux langages de programmation de plus haut niveau ont été inventés. Ces langages offrent au programmeur des constructions plus expressives et plus abstraites que celles du langage machine : les structures de contrôle (conditionnelles, itération, récursion, ...) ; les structures de données (tableaux, enregistrements, listes, arbres, graphes, ...) ; et des notions de composants logiciels (bibliothèques, modules, classes, interfaces, ...). Par exemple, un langage de niveau intermédiaire comme le langage C manipule explicitement la mémoire presque comme le langage machine (via les pointeurs et l'allocation et la libération de mémoire explicites). Cependant, il ne permet pas de manipuler les registres et offre des structures de contrôle d'assez haut niveau. Des langages de plus haut niveau comme Java présentent au programmeur une vision plus abstraite de la mémoire, sans pointeurs ni libération explicite.

De plus, chaque langage encourage une certaine manière de décrire et de penser les programmes : on parle de *paradigme de programmation*. Ainsi, le paradigme impératif met l'accent sur l'évolution d'un état mémoire au cours du calcul. Le paradigme par objets (*object-oriented programming*) repose sur l'encapsulation des états et l'organisation hiérarchique des structures de données. Le paradigme logique et par contraintes présente l'exécution des programmes comme la recherche de solutions à un ensemble de règles logiques et de contraintes. Le paradigme dans lequel s'inscrit cette thèse est celui de la *programmation fonctionnelle*, centré sur la notion de fonction.

Les langages fonctionnels présentent les programmes comme des ensembles de fonctions, souvent définies par cas et de manière récursive, et l'exécution des programmes comme le calcul des valeurs de ces fonctions. Ils se situent à la frontière entre les systèmes formels et les langages de programmation. A l'origine des langages fonctionnels on trouve un système

formel, le λ-calcul. L'ancêtre des langages fonctionnels est le langage Lisp [80]. Scheme [109], Haskell [57], SML [84] et Caml [73] sont d'autres représentants de cette famille de langages. On retrouve également l'approche fonctionnelle dans des langages de spécification comme celui de l'assistant de preuve Coq [20].

Les langages fonctionnels se caractérisent par un haut niveau d'expressivité et d'abstraction : ils manipulent des objets mathématiques comme les algèbres de termes (structures arborescentes de données) et les fonctions elles-mêmes, qui sont des données de première classe. On peut naturellement définir des fonctions qui opèrent sur d'autres fonctions : on parle alors de fonctions *d'ordre supérieur*. Pour ces raisons, les langages fonctionnels sont particulièrement adaptés à la programmation de calculs symboliques complexes : interprétation, compilation, analyse statique et transformation de programmes, mais aussi démonstration automatique, déduction logique, et réécriture.

La compilation Quel que soit le langage dans lequel un programmeur écrit ses logiciels, il faut savoir passer de ce langage de plus ou moins haut niveau au langage de la machine. *La compilation* est le processus qui traduit un langage de programmation vers le langage machine. Cette traduction est effectuée automatiquement par un programme appelé *compilateur*, qui produit du code machine directement exécutable à partir des programmes «source» fournis par le programmeur.

Les grandes différences d'expressivité entre un langage de programmation de haut niveau et un langage machine ou assembleur rendent très difficile la transformation directe entre les deux langages. C'est d'autant plus vrai pour la compilation *optimisante* qui vise à améliorer les performances du code produit en éliminant des inefficacités présentes dans le programme source. On procède souvent par une chaîne de compilation atténuant pas à pas les différences par des transformations successives, appelées *passes de compilation*, passant par des langages intermédiaires. Par exemple, ces langages intermédiaires vont petit à petit expliciter les ordres à effectuer sur la mémoire et les registres, tandis que le langage source n'y faisait aucune allusion. Le développement d'un compilateur ne se résume donc pas à écrire un programme traduisant les constructions du langage source en des constructions du langage cible : ce développement impose aussi des choix judicieux de langages intermédiaires et un découpage astucieux en passes. Les langages intermédiaires ne sont généralement pas destinés à être manipulés par des utilisateurs, aussi peuvent-ils être moins ergonomiques que les langages de programmation. Certains de ces langages intermédiaires servent à expliciter des structures de l'environnement d'exécution (notion de pile d'appels, par exemple), d'autre ont une grammaire particulière permettant d'opérer plus facilement des optimisations (par exemple, le style «par passage de continuation» étudié au chapitre 4). Le choix de ces langages intermédiaires dépend des optimisations et des choix de représentations des données du langage source dans le langage cible. Une fois ces choix faits, le compilateur se présente comme une chaîne de transformations successives. Chaque transformation effectue un certain nombre d'optimisations ou de rapprochements avec le langage cible.

Pour exécuter un programme écrit dans un langage de haut niveau, il ne suffit pas en général de le compiler vers du code machine : il faut aussi lui adjoindre les services d'un *environnement d'exécution*. Par exemple, dans les langages de bas niveau, le programmeur peut lui-même gérer l'utilisation de la mémoire en désallouant explicitement les données qui ne lui sont plus utiles. Dans les langages de haut niveau comme les langages fonctionnels, cette tâche est confiée à un *gestionnaire de mémoire automatique*, qui est la composée d'un allocateur de mémoire et d'un glaneur de cellule (GC, *garbage collector* en Anglais). Le GC

est chargé de nettoyer la mémoire afin de libérer l'espace qui n'est plus utile à l'exécution du programme, lorsque l'espace mémoire vient à manquer. La gestion de mémoire automatique est le principal composant d'un environnement d'exécution pour un langage fonctionnel.

Les méthodes formelles désignent un ensemble de techniques fondées sur les mathématiques et la logique et permettant de raisonner sur le logiciel et le matériel informatiques afin de vérifier qu'ils sont conformes à leurs spécifications. On parle alors de programme ou de système *vérifié formellement*. Parmi ces méthodes, on distingue plusieurs familles telles que :

La vérification de modèles (*model checking*). Le système que l'on veut vérifier est modélisé de telle sorte que toutes les exécutions possibles puissent être validées.

L'analyse statique par interprétation abstraite. Il s'agit d'une sur-approximation des exécutions possibles par calcul symbolique.

La vérification déductive, aussi appelée preuve de programme. La vérification se fait par démonstration, automatique ou assistée par l'utilisateur, de formules logiques combinant les spécifications et les exécutions possibles du système.

Les méthodes formelles permettent d'obtenir des garanties très fortes de sûreté sur un système informatique, mais sont difficiles et coûteuses à mettre en œuvre. C'est pourquoi elles sont principalement employées pour le logiciel *critique*. Il s'agit de programmes utilisés dans des domaines mettant des vies humaines en jeu, comme les transports (commandes électroniques de vol d'avions, métro sans conducteurs, ...), la médecine (pace-makers, équipements de radiothérapie, ...), ou l'énergie (contrôle de centrales nucléaires).

Les méthodes formelles ont progressé de manière significative ces 10 dernières années : on parvient maintenant à vérifier formellement des logiciels complets directement sur leur code source, et non plus sur un modèle abstrait de ce code. Par ailleurs, le *hardware* et tout particulièrement les microprocesseurs sont de plus en plus vérifiés par méthodes formelles également. Cependant, il reste un élément non vérifié dans ce schéma. En effet, le logiciel est écrit dans un langage de programmation et va être compilé vers le langage machine. Or, le compilateur est un programme comme un autre : il peut contenir des erreurs. Certaines de ces erreurs pourraient faire produire du code machine incorrect à partir d'un code source correct, invalidant ainsi les garanties obtenues par vérification formelle du code source.

La vérification de compilateurs a pour objectif d'assurer que le scénario précédent ne puisse pas se produire. Un compilateur est dit formellement vérifié s'il a la propriété de *préserver la sémantique* des programmes qu'il compile :

> *Pour tout programme source, si la compilation ne signale pas d'erreur, alors le code exécutable produit a le même comportement observable que le programme source.*

L'utilisation d'un compilateur formellement vérifié pour compiler un programme dont le code source a été lui-même vérifié garantit donc que l'exécutable compilé a les mêmes propriétés que le programme source. Cela apporte donc un degré supplémentaire de confiance dans la sûreté du programme.

Plusieurs méthodes sont envisageables pour vérifier formellement un compilateur :

Prouver correcte l'implantation du compilateur. Le compilateur est un programme comme un autre : on peut donc lui appliquer des méthodes formelles, notamment la

preuve de programme, en prenant comme spécification la propriété de préservation sémantique énoncée ci-dessus. Cette preuve est écrite une fois pour toutes.

Valider *a posteriori* le résultat de la compilation. De manière générale, une alternative à prouver correct un programme consiste à écrire un *validateur* pour les résultats de ce programme et à vérifier formellement ce validateur. Le validateur est une procédure de semi-décision qui prend en paramètre le résultat d'une exécution du programme et vérifie que ce résultat est conforme à la spécification du programme.

Cette approche, appliquée à la vérification de compilateurs, porte le nom de *validation de transformations* [89, 97]. Le validateur prend en paramètres le code source et le code machine produit par le compilateur, et vérifie que ces deux codes sont sémantiquement équivalents. La validation s'effectue donc à chaque appel du compilateur. Dans cette approche, il suffit de vérifier formellement la correction du validateur : le compilateur lui-même n'a pas besoin d'être vérifié.

Produire du code auto-certifiant. Cette approche est connue sous le nom de *Proof-Carriyng Code* [88]. Elle repose sur l'utilisation de compilateurs *certifiants*, qui produisent non seulement du code exécutable, mais également un *certificat* qui prouve que ce code vérifie une certaine spécification (comme par exemple la sûreté vis-à-vis de la mémoire et des types). Ce certificat constitue une preuve de la correction du code exécutable, preuve qui peut être vérifiée indépendamment du programme source.

La vérification formelle de compilateurs ou de passes de compilation est depuis longtemps un sujet de recherche actif : la première publication sur ce sujet remonte à 1967 [78]. La bibliographie de Dave [30] répertorie une bonne partie des travaux dans ce domaine, qui vont de la preuve de compilateurs simplifiés de langages «jouets» vers des machines abstraites à la vérification d'optimisations ambitieuses.

Le projet CompCert [69] s'attelle à la vérification formelle de compilateurs à la fois réalistes et optimisants. Le but est de développer et vérifier formellement des compilateurs utilisables notamment dans le domaine de l'embarqué critique (avionique, transports, ...).

C'est dans ce cadre qu'a été développé et vérifié un compilateur pour un sous-ensemble significatif du langage C [11, 12], le sous-ensemble utilisé dans l'embarqué critique. Ce compilateur produit du code assembleur pour le processeur PowerPC, processeur également très prisé dans l'embarqué critique. Ce compilateur est composé d'une chaîne de compilation en amont, le *front-end*, traduisant le sous-ensemble Clight de C en code intermédiaire Cminor, qui se combine à une chaîne en aval, le *back-end*, qui produit du code assembleur à partir de Cminor en effectuant quelques optimisations [70]. Le *back-end* de CompCert est réutilisable : son langage d'entrée Cminor a une expressivité permettant d'envisager la compilation d'autres langages via des *front-ends* ayant pour cible Cminor. (La section 2.3 décrit Cminor plus en détails.)

Un compilateur vérifié formellement pour langage purement fonctionnel. Notre thèse a pour but le développement et la preuve de correction d'un compilateur pour un langage purement fonctionnel, produisant du code Cminor et s'intégrant donc dans la chaîne de compilation CompCert.

Plusieurs raisons motivent l'intérêt de ce travail. D'une part, en raison du haut niveau d'abstraction des langages fonctionnels, leur compilation met en jeu des transformations ambitieuses et très différentes de celles que l'on trouve dans les compilateurs C. La preuve

de correction de ces transformations est donc un problème intéressant. D'autre part, même si les langages fonctionnels ne sont pas, aujourd'hui, utilisés pour la programmation de logiciels embarqués critiques, ils sont beaucoup utilisés pour écrire des outils de production et de vérification pour le logiciel critique. Par exemple, le générateur de code Scade KCG 6, l'analyseur statique Astrée, le prouveur de programmes Caduceus, la plateforme Frama-C et l'assistant de preuves Coq sont programmés dans le langage Caml. Un compilateur formellement vérifié pour un sous-ensemble de Caml peut donc contribuer à augmenter la confiance que l'on peut accorder à ces outils.

Une motivation supplémentaire pour notre travail est le lien étroit qui existe entre programmation fonctionnelle et preuve sur machine dans des systèmes comme Coq. Comme le montre l'*isomorphisme de Curry-Howard* [53], qui relie systèmes de types et logiques constructives, l'activité de programmation dans un langage fonctionnel et l'activité de preuve dans une logique comme celle de Coq ont beaucoup en commun. Ainsi, le langage de spécification de Coq, *Gallina*, contient un λ-calcul fortement typé d'ordre supérieur, utilisant le Calcul des Constructions Inductives comme système de types. L'assistant de preuve Coq offre le moyen d'écrire des programmes purement fonctionnels directement dans son langage de spécification, et de les prouver dans le même langage. À partir de ces spécifications, le mécanisme d'*extraction* de Coq produit automatiquement du code Caml, Haskell ou bien encore Scheme, code exécutable après compilation [74, 75].

Considérons un instant un programme écrit directement en Gallina puis vérifié formellement dans l'assistant de preuves Coq. (C'est le cas du compilateur CompCert et du *front-end* que nous avons développé dans cette thèse.) Un tel programme n'est pas directement exécutable : il faut l'extraire vers un langage fonctionnel, puis compiler le code extrait. Ceci met en jeu deux transformations de code (l'extraction puis la compilation) qui pourraient introduire des erreurs dans le programme vérifié en Coq. Pour éliminer ce risque, il est naturel de vérifier formellement ces deux transformations. La vérification du mécanisme d'extraction de Coq est en cours dans le cadre de la thèse de Stéphane Glondu [41]. Quant à la seconde transformation (compilation du code extrait), le compilateur formellement vérifié que nous avons développé dans le cadre de cette thèse répond exactement au besoin : le langage source que nous avons choisi est suffisamment riche pour inclure les programmes issus de l'extraction Coq[1]. La combinaison des travaux de Stéphane Glondu, de nos travaux, et du *back-end* CompCert fournit donc une chaîne d'exécution sûre pour les programmes écrits directement en Coq.

Contributions. Le présent manuscrit décrit le développement et la vérification formelle d'un compilateur pour langage fonctionnel, dont voici les principales caractéristiques.

- Le langage source est le fragment purement fonctionnel du langage ML. En plus du λ-calcul (fonctions d'ordre supérieur), il traite des liaisons locales, des fonctions récursives et un des mécanismes les plus riches de ML : les types concrets et le filtrage.
- Nous avons développé une chaîne de compilation de type *front-end*, compilant vers le langage intermédiaire Cminor. En appelant ensuite le *back-end* CompCert, nous produisons du code en assembleur PowerPC.
- A l'image des compilateurs modernes pour langages fonctionnels, notre compilateur présente une interface avec un gestionnaire de mémoire automatique à glaneur de cellules (GC), implanté en Cminor.

[1] Notons que nous n'avons pas traité les fonctions mutuellement récursives

- Notre compilateur implante plusieurs optimisations propres à la compilation de langages fonctionnels : la décurryfication des fonctions à plusieurs arguments, l'optimisation des appels en position terminale, ainsi qu'une transformation en style par passage de continuation. En prenant en compte les optimisations déjà présentes dans le back-end CompCert, nous aboutissons ainsi à un compilateur réaliste.
- Le développement a été mené dans l'assistant de preuve Coq et chaque passe du compilateur a été formellement vérifié en Coq.

Les enjeux de ce développement nous ont mené à bien plus que l'implantation d'algorithmes de compilation dans l'assistant de preuve Coq. La vérification formelle d'un compilateur invite à la formalisation de sémantiques propices à la preuve de préservation sémantique. De plus, la définition des langages intermédiaires de notre chaîne de compilation, ainsi que l'implantation des algorithmes de compilation, ont été fortement influencés par les besoins de cette preuve ; nous pensons qu'ils y ont gagné en clarté. Cette étude offre donc un nouveau regard sur les langages fonctionnels et leur compilation.

Travaux connexes Comme nous l'avons dit plus haut, la vérification formelle de compilation est un domaine à part entière depuis le milieu des années 60 [78]. Maulik A. Dave répertorie dans [30] l'ensemble de ces travaux jusqu'en 2003. Nous nous intéressons plus particulièrement à la vérification formelle de la compilation de langage fonctionnel.

Les premiers travaux dédiés à la vérification formelle de la compilation d'un langage fonctionnel sur machine [50] concerne la compilation formellement vérifiée du λ-calcul non typé vers une variante de la machine abstraite CAM [23] en utilisant le prouveur Elf.

Des travaux concernant la compilation d'un langage plus expressif apparaissent dans [32]. Le but original est le développement d'un compilateur vérifié formellement d'un sous-ensemble de Common Lisp (ANSI) vers de l'assembleur. En fait, il prouve la compilation vers un langage intermédiaire avec pile explicite, à l'aide du prouveur PVS.

Samuel Boutin a étudié la vérification formelle de compilation de miniML non typé sans type concret (et donc sans filtrage) vers la machine abstraite CAM. Cette certification est partiellement mécanisée dans l'assistant de preuve Coq, version 5.2.

Plus récemment, Benjamin Gregoire dans sa thèse [45], vérifie formellement, dans l'assistant de preuve Coq, la compilation du λ-calcul avec type concret avec une version fonctionnelle, mais tout aussi expressive, de la machine abstraite ZAM [64].

Ces deux derniers travaux sont les plus proches des nôtres. Cependant, la compilation est faite vers une machine abstraite et non pas vers un « vrai »microprocesseur.

Adam Chlipala [17] produit, dans l'assistant de preuves Coq, un compilateur préservant le typage et la sémantique pour le $\lambda-$calcul simplement typé vers un petit langage assembleur idéaliste, contenant une infinité de registres et de cellules mémoire ainsi que des instructions spécifiques à la gestion automatique de mémoire. La préservation de typage est assurée par l'utilisation des types dépendants de Coq, qui permettent de représenter un terme par une combinaison d'un terme et d'une dérivation de typage. La stratégie de compilation est dirigée par les types. La compilation se fait en 6 transformations. Les deux premières étapes mettent le programme en style par passage de continuations (CPS). La première produit des redex administratifs (ajouté par la transformation) et la seconde les élimine. Dans cette thèse, la transformation CPS se fait en une transformation optimisante qui ne produit pas de redex administratif. Clhipala esplicite les fermetures de la manière que

nous dans la troisième étape. Cependant, il compile vers les propres fonctions de Coq. La quatrième étape est la construction explicite des constructions de données, les informations de typage disparaissent et laissent place à des "tags" qui différencient les objets alloués des entiers. En contrepartie, la non-terminaison ou le blocage d'une évaluation par l'allocateur sont traités de manière traditionnelle au travers des types co-inductifs de Coq. L'évaluation des termes issue de cette étape nécessite une abstraction de notion de tas qui est une liste d'enregistrements "taggés". Ces "tags" prépare le terrain pour le traitement des racines. Dans nos travaux, le calcul des racines et leur enregistrement sont explicites, ce qui nous permet d'avoir une meilleur approximation de l'ensemble des racines réelles à l'appel d'un GC. La cinquième étape transforme les variables en registres. Les registres sont alors aussi "taggés", ce qui prépare la construction des vecteurs de racines pour la dernière étape, c'est-à-dire la production de code assembleur. Ces travaux forment un tour de force du point de vue de l'automatisation des preuves. Cependant, le langage source restent beaucoup plus restreint et moins expressif que le nôtre. De même, le compilateur ne contient aucune optimisation et le calcul des racines est beaucoup plus sur-approximé que celui que nous présentons ici. Enfin, bénéficiant du back-end de CompCert, nos problèmatique se portent sur un modèle mémoire moins abstrait et nous produisons de l'assembleur PowerPC.

On trouve aussi dans la littérature des travaux concernant la vérification formelle d'une passe de compilation propre aux langages fonctionnels. Ainsi, des travaux concernant la vérification formelle de transformation CPS ont aussi été menés, d'une part en Isabelle/HOL par Minamide et Okuma [86] ; d'autre part en Twelf par Tian [110]. Nous revenons sur leurs travaux au chapitre 4 où nous présentons notre transformation CPS formellement vérifiée. De même, d'autres optimisations de compilation de langages fonctionnels ont été formellement vérifiés. Hannan et Hicks ont vérifié formellement, partiellement sur machine, la décurryfication d'ordre supérieur [49] et l' *arity raising* [47]. Plus récemment, Hannan et Fischbach [39] ont spécifié et vérifié une transformation de λ-*lifting*.

Bien entendu, des travaux concernant la vérification formelle de compilateurs pour langage fonctionnel ont été menés sur papier. Guttman et *al* [46] ont vérifié formellement, sur papier l'intégralité d'un compilateur réaliste pour Scheme. Hardin et *al* [51] ont vérifié formellement, sur papier, la compilation du λ-calcul avec substitution explicite vers la machine abstraite de Krivine et la machine abstraite fonctionnelle FAM [15].

Description du manuscrit Ayant développé et vérifié formellement le front-end pour miniML dans l'assistant de preuves Coq, nous ne présentons pas exhaustivement les preuves. Nous avons préféré développer les différentes spécifications et invariants à la base de ces preuves afin d'expliciter notre démarche. Cependant, nous avons décrit certaines preuves lorsque cela nous a semblé pertinent.

Le chapitre suivant de ce manuscrit, le chapitre 2, précise notre démarche en détaillant les langages source et cible de notre chaîne de compilation. Nous y avons aussi décrit plus formellement le principe des preuves de préservation sémantique et les étapes de la chaîne de compilation.

Le schéma de la figure 1.0.1 résume l'organisation de notre chaîne de compilation. Les chapitres 3 à 7 décrivent les étapes de la chaîne de compilation. Pour chaque étape de compilation, nous commençons par décrire le langage intermédiaire cible par sa syntaxe et sa sémantique. La transformation est alors décrite avant sa preuve de préservation sémantique.

Plus précisément, nous définissons les invariants et propriétés qui nous ont été nécessaires. Bien entendu, nous motivons chacun de nos choix de transformation et discutons des difficultés et perspectives qu'elles offrent dans chacun de ces chapitres.

Le chapitre 3 présente une optimisation courante de la compilation de langages fonctionnels, la décurryfication. Le chapitre 4 décrit une transformation CPS qui met les programmes dans une forme particulière, le style par passage de continuation est un style permettant d'effectuer de nombreuses optimisations de compilation des langages fonctionnels. Le chapitre 5 aborde la traduction vers le langage Cminor en passant de programmes avec abstractions à des programmes à fonctions globales closes via l'explicitation des fermetures. Le chapitre 6 est consacré à la mise en place et la vérification formelle de l'interaction avec un gestionnaire de mémoire à glaneur de cellules. La dernière transformation produisant du code Cminor est décrite au chapitre 7.

Les observations liées à l'expérimentation et à l'utilisation de notre compilateur vérifié sont décrits au chapitre 8. Enfin, le chapitre 9 expose les conclusions et perspectives de nos travaux.

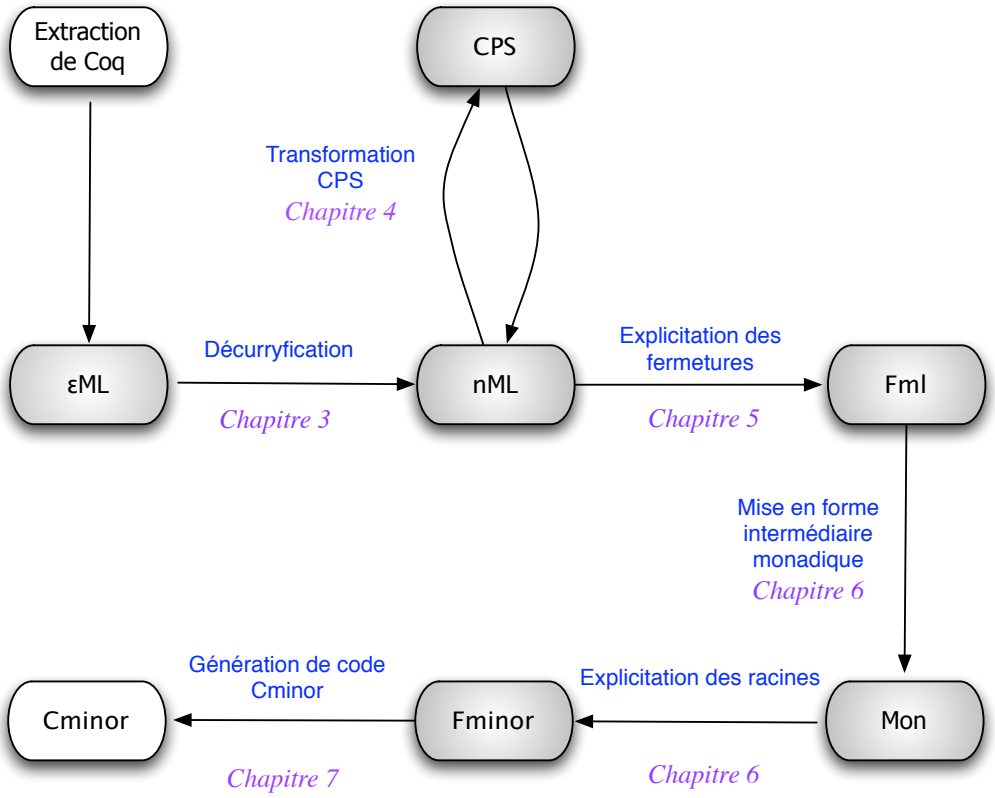


FIG. 1.0.1 – Le front-end miniML

2 Les principaux acteurs

Ce chapitre a pour objectif de préciser notre démarche en présentant l'ensemble des acteurs et concepts de nos travaux. La première section est un rappel sur les langages fonctionnels. La deuxième section est consacrée au langage source de la chaîne de compilation. Cette présentation se veut complète, nous y justifions nos choix et présentons les propriétés de notre langage d'entrée. La troisième section est une présentation succincte du langage cible Cminor, plus précisément du sous-ensemble que nous utilisons pour notre compilation. La quatrième section décrit le schéma général de nos preuves de préservation sémantique et nous l'illustrons d'un premier exemple : la transformation dite numérotation des constructeurs. La cinquième section décrit notre chaîne de compilation, en précisant les lignes directrices qui mènent à la conception et vérification d'un compilateur réaliste et optimisant pour un langage purement fonctionnel.

2.1 Rappels sur les langages fonctionnels

2.1.1 ML par l'exemple

Dans cette section nous rappelons les principales structures syntaxiques de ML en les illustrant par des exemples.

λ-calcul pur, ML purement fonctionnel

Le fragment purement fonctionnel de ML non typé, contient le λ-calcul. Les fonctions sont des objets de première classe, elles peuvent être passées en paramètres, et être retournée par une application : on parle *d'ordre supérieur*. Le λ-calcul se présente par trois constructions : l'abstraction $(\lambda x.\ t)$, l'application $t_1\ t_2$ et les variables x.

$$t\ ::=\ x\ \mid\ \lambda x.\ t\ \mid\ t\ t\ .$$

Exemple : La fonction identité,

$$\lambda x.\ x,$$

retourne son argument.

Exemple : La fonction `plus`,

$$\lambda x.\ \lambda y.\ x + y,$$

effectue l'addition de ses 2 paramètres. (Nous supposons que les entiers et la fonction + sont correctement encodés). La fonction plus est ici curryfiée, les paramètres sont passés un par un.

Exemple : La fonction `plus5`,

$$(\lambda x.\ \lambda y.\ x + y)\ 5,$$

est la fonction `plus` appliquée à un seul argument 5. La réduction de ce terme produit une abstraction : $\lambda y.\ 5 + y$. C'est une propriété caractéristique des langages fonctionnels d'ordre supérieur.

Exemple : La fonction `plus5-liste`

$$\texttt{map}\ l\ (\lambda y.\ 5 + y),$$

`map` est une fonction qui prend en paramètre une fonction f et une liste $x_1; \ldots; x_n$ et retourne la liste $f\ x_1; \ldots; f\ x_n$. La fonction plus5-liste illustre l'autre caractéristique des langage fonctionnels d'ordre supérieur : la capacité de prendre en argument une fonction. La fonction plus5-liste applique à chaque élément de la liste l la fonction plus5 et retourne la liste des résultats.

Liaisons locales

Le sous-ensemble purement fonctionnel de ML est syntaxiquement plus riche que le λ-calcul.
Une liaison locale permet de lier à une variable un terme dans un autre terme, grâce à la construction syntaxique suivante :

$$\texttt{let}\ x = t_1\ \texttt{in}\ t_2.$$

Le terme t_1 est lié à la variable x dans le terme t_2. Autrement dit la valeur dynamique de x dans t_2 est la valeur de t_1. Le terme t_2 délimite la *portée lexicale* de x. En dehors de t_2, la variable x n'est pas associée à t_1.

Exemple : La fonction `plus-double`,

$$\lambda x.\ \lambda y.\ (\lambda z.\ z + z)\ x + (\lambda z.\ z + z)\ y,$$

effectue l'addition des doubles de x et y. Dans cette exemple, la fonction $(\lambda z.\ z+z)$ est écrite deux fois et sera évaluée deux fois. Un moyen d'optimiser ce terme est de lier localement la fonction double :

$$\texttt{let}\ \mathit{double} = \lambda z.\ z + z\ \texttt{in}\ \lambda x.\ \lambda y.\ \mathit{double}\ x + \mathit{double}\ y.$$

Ainsi, *double* ne sera évalué qu'une fois.

Exemple : Le terme,

$$\texttt{let}\ x = (\texttt{let}\ x = t\ \texttt{in}\ 2 + x)\ \texttt{in}\ x,$$

est formé par une imbrication à gauche de liaison locale. Nous cherchons ici à montrer l'importance des portées lexicales. Dans $(\texttt{let}\ x = t\ \texttt{in}\ 2 + x)$, x désigne t dans $2 + x$. Ce x n'existe pas en dehors de ce terme, sa portée lexicale est $2 + x$. Le x le plus à droite dans notre exemple désigne le terme $(\texttt{let}\ x = t\ \texttt{in}\ 2 + x)$.

Les fonctions récursives

Une fonction récursive est une fonction qui dans son corps fait appel à elle-même. Il va s'agir de lier localement une abstraction dans un terme. La construction syntaxique qui définit une fonction récursive est :

$$\texttt{letrec}\ f\ x\ =\ t\ \texttt{in}\ t_1.$$

Ce terme correspond à la définition de la fonction récursive f de paramètre x et de corps t dans le terme t_1. Autrement dit, dans le terme t_1, la variable f désigne cette fonction récursive f. Dans le corps de l'abstraction, f est aussi une variable liée qui désigne cette même fonction, ce qui permet la récursivité. Enfin, le paramètre x n'est lié que dans le corps de l'abstraction, il n'est pas lié dans le terme t_1. Nous ne dérogerons pas à la règle et présentons quelques exemples classiques de fonctions récursives.

Exemple : La fonction factorielle

$$\texttt{letrec}\ \mathit{fact}\ x\ =\ \texttt{if}\ (x = 1)\ \texttt{then}\ 1\ \texttt{else}\ \mathit{fact}\ (x-1) \times x\ \texttt{in}\ \mathit{fact}\ 5.$$

On suppose la multiplication $\times$ et la conditionnelle définies de manière usuelle.

Exemple : La fonction de Fibonacci

$$\texttt{letrec}\ \mathit{fib}\ x\ =\ \texttt{if}\ (x <= 1)\ \texttt{then}\ x\ \texttt{else}\ \mathit{fib}\ (x-1) + \mathit{fib}\ (x-2)\ \texttt{in}\ \mathit{fib}\ 5.$$

Une extension naturelle des fonctions récursives est la définition de *fonctions mutuellement récursives*. Deux fonctions sont mutuellement récursives si leurs définitions sont mutuelles.

Exemple : Les fonctions `odd` et `even` sont deux fonctions mutuellement récursives.

$$\texttt{letrec even}\ x\ =\ \texttt{if}\ x = 0\ \texttt{then true else odd}\ (x-1)$$

$$\texttt{and odd}\ x =\ \texttt{if}\ x = 0\ \texttt{then false else even}\ (x-1).$$

`odd` teste si un entier est impair, tandis que `even` teste si un entier est pair.

Types concrets et filtrage

Les types concrets sont la principale structure de données des langages fonctionnels. Ils permettent de représenter des termes algébriques sous forme d'arbres étiquetés par des constructeurs. Les langages non typés comme Lisp[80] ou Scheme[109] offrent un seul type concret universel : les S-expressions. Dans les langages statiquement typés, comme Haskell[57], OCaml [73] ou Coq [20], le programmeur déclare explicitement les formes des types concrets préalablement à leur utilisation. La déclaration de type suivante (en OCaml) va nous permettre d'exhiber les caractéristiques que l'on exige pour définir des types concrets :

```
type ensemble =
| Vide
| Singleton of nat
| Union of ensemble*ensemble
```

Cette déclaration permet de définir un objet de type `ensemble` comme pouvant être soit l'ensemble `Vide`, soit un ensemble à un élément `Singleton` ou encore une `Union` de deux ensembles. `ensemble` est le nom du type défini, `Vide`, `Singleton` et `Union` sont les constructeurs du type `ensemble`. Un constructeur peut porter des paramètres, comme pour `Singleton` ou `Union`, ou aucun comme pour `Vide`. Chaque information de type représente le type du paramètre attendu. Un type concret peut être récursif, en effet, comme pour le constructeur `Union`, un constructeur de type concret peut attendre en paramètre un objet du même type. On remarquera que les constructeurs de type sont décurryfiés. Un constructeur, en fait, prend en paramètre un n-uplet qui contient tous les objets attendus. Dans d'autres langages fonctionnels, comme le langage de spécification de Coq, les constructeurs sont curryfiés. Ainsi, `Union (Singleton 3)` est une expression correcte. Considérons la déclaration du type concret `ensemble` en Coq :

```
Inductive ensemble : Set :=
  | Vide : ensemble
  | Singleton : nat -> ensemble
  | Union : ensemble -> ensemble -> ensemble.
```

Chaque constructeur est défini comme une fonction qui attend aucun, un ou plusieurs argument(s) pour produire un objet de type `ensemble`. L'expression `Union (Singleton 0)` est correcte en Coq.

L'utilisation d'un type concret permet de discriminer sur un objet du type selon le constructeur qui a servi à sa création et de choisir la suite du calcul : c'est le *filtrage de motif*. Par exemple, on peut vouloir compter les éléments d'un ensemble, c'est-à-dire son cardinal. On définit alors une fonction récursive, puisque pour le cas d'une union, son cardinal est la somme des cardinaux de ses fils.

```
letrec cardinal e=
    match e with
   | Union(e1,e2) -> (cardinal e1) + (cardinal e2)
   | Singleton n -> 1
   | Vide -> 0
```

Le filtrage dans la fonction `cardinal` est dit *exhaustif*, car tous les constructeurs du type `ensemble` y sont traités. Une *clause* est constituée d'un *motif* (un constructeur et les noms des paramètres) et d'une *action* dans laquelle les variables du motif sont liées. Dans tous les langages fonctionnels, le filtrage ne concerne qu'un seul type concret. Il n'est pas obligatoirement exhaustif. Il est aussi possible d'indiquer une suite de calculs commune à plusieurs motifs. Enfin, le filtrage peut être plus profond : on peut discriminer récursivement sur les paramètres formels des motifs. Par exemple, en OCaml, on peut définir la fonction qui simplifie les unions d'un ensemble e avec l'ensemble vide en l'ensemble e, `simplif`. La clause introduite par `_` filtre tous les motifs qui n'ont pas été filtrés par les clauses précédentes.

```
letrec simplif e =
 match e with
 | Union(e1,Vide) -> simplif e1
 | Union(Vide,e1)-> simplif e1
 | Union(e1,e2) -> Union(simplif e1,simplif e2)
 | _ -> e
```

Les filtrages plus profonds sont plus efficaces aussi bien au niveau de la concision de code que pour leur compilation.

Dans le langage source de cette étude appelé εML, la gestion des types concrets, s'inspire de ce qui existe déjà et se veut être la plus simple à compiler possible. Les types concrets sont donc déclarés. Comme en Coq, les noms des constructeurs d'un même type sont disjoints et les noms de tous les constructeurs d'un programme (nous ne traitons pas les modules) sont disjoints deux à deux. Cependant, les constructeurs ne portent aucune information, ni d'arité ni de typage, dans les déclarations de types. Notre langage de motifs est plus simple que celui de ML ou Haskell : nous pouvons seulement filtrer sur le constructeur de tête d'une valeur, mais pas récursivement sur les valeurs des arguments de ce constructeur. Néanmoins, le filtrage en profondeur de ML ou Haskell peut se compiler en une cascade de filtrages superficiels comme ceux fournis par εML, le langage source de notre chaîne de compilation. De plus les filtrages sont exhaustifs. Voici quelques exemples de types concrets et de deux fonctions les manipulant issus de la bibliothèque standard de Coq, qui nous est familière.

Exemple : Le type des entiers naturels,

$$\texttt{type nat = O | S of nat} \text{ (en OCaml).}$$

$$nat := \texttt{O | S} \text{ (en } \varepsilon\text{ML)}$$

Le type `nat` a deux constructeurs, le premier constructeur `O` désigne l'entier naturel 0, le second constructeur construit le successeur d'un entier. En εML, les informations d'arité des constructeurs et les types de leurs éventuels paramètres ont disparu dans la déclaration. La fonction `plus` qui additionne deux entiers naturels s'écrit comme suit en εML

```
letrec plus x = λ y.
  match x with
  | O → y
  | S m → S ((plus m) y)
in t
```

Il s'agit d'une fonction récursive curryfiée, le paramètre sur lequel on fait les appels récursifs est x. Le filtrage est exhaustif parce qu'il y a une clause pour chacun des constructeurs. Cette fonction a pour portée lexicale t.

Exemple : La liste polymorphe

$$\texttt{type 'a list = Nil | Cons of 'a * 'a list} \text{ (en OCaml)}$$

`list = Nil | Cons` (en εML)

Comme pour les entiers naturels aucun renseignement n'est donné sur les éventuels paramètres des constructeurs, le polymorphisme s'implante alors facilement (il est implicite). La fonction `map` se définit somme suit :

```
letrec map x = λ f.
     match x with
     | Nil →  Nil
     | Cons m l → Cons (f m) (map l f)
in ...
```

Cette fonction prend en paramètre une liste sur laquelle les appels récursifs se feront et une fonction. Ici encore, le filtrage est exhaustif.

2.1.2 Un bref historique des langages fonctionnels

Le λ-calcul est un système de calcul ayant comme objets principaux les fonctions (ou abstractions) et applications. Introduit par Alonzo Church dans les années 1930, le λ-calcul a toujours été considéré comme un langage de programmation et un système formel permettant d'étudier d'autres langages (métalangage) ou d'autres systèmes formels. Le λ-calcul trouve non seulement son essence comme système formel pour les fonctions mathématiques [18] ; mais aussi dans la logique combinatoire [24]. Une étude approfondie du λ-calcul est produite dans [8].

La famille des langages de programmation inspirés par le λ-calcul sont les langages fonctionnels. Les langages fonctionnels contemporains les plus répandus sont Scheme [109, 59], le langage purement fonctionnel Haskell [57] et les langages de la famille ML tels que OCaml [73] et Standard ML [84]. Erlang est une langage fonctionnel concurrent. Tous ont pour ancêtre commun le langage Lisp [80], apparu dans les années 60, le dialect actuel de lisp est CommonLisp.

En plus d'être des langages pour la programmation, les langages fonctionnels sont aussi des langages adaptés aux systèmes logiques et se prêtent au développement d'assistants de preuves. Les assistants de preuves sont des outils implantant des systèmes logiques dans le but de permettre d'automatiser le développement de preuves. L'utilisation de langages fonctionnels dans un assistant de preuve remonte à LCF [82]. LCF est un assistant de preuves pour la logique PCF de Scott (non publié avant 1993 [103]). Cette première version de LCF est Stanford LCF. Elle présente deux faiblesses : le coût des scripts de preuves en mémoire et la difficulté à étendre les tactiques. La solution que propose Milner dans le développement de Edimburg LCF est ML (Meta-Language) [83]. On peut décrire ML comme étant un langage fonctionnel, avec des traits impératifs et fortement typé. Par la suite de grandes améliorations sont apportées au langage ML notamment sous l'impulsion de Gérard Huet. Plus précisément, au début des années 1980, Huet et Cousineau cherchent à rendre ML compatible avec les différents compilateurs Lisp. Dans cette voie, un compilateur est ajouté et l'expressivité de ML augmente aussi via l'ajout des types concrets et du filtrage. Cette implantation de ML fut brièvement nommée LE_ML, elle est incluse dans LCF par Paulson dans Cambridge LCF et est à la base de la première version de HOL (voir [44]).

En 1984, la machine abstraite fonctionnelle FAM [15], la définition de Standard ML par Robin Milner et *al* [84] et le calcul des combinateurs catégoriques de Pierre Louis Curien dont une correspondance avec le λ-calcul peut être vue comme une méthode de compilation de ML, convergent vers le développement de la machine abstraite catégorique [23] et d'une nouvelle implantation de ML : Caml.

Caml a pour première utilité le développement de l'assistant de preuve Coq. Le développement de Coq et de Caml sont donc intimement liés depuis leur création.

La première implantation de Caml apparaît en 1987 (Alesander Suarez, Pierre Weis, Michel Mauny). Au début des années 1990, l'implantation de Caml change et gagne en portabilité. Xavier Leroy propose une compilation de Caml vers du bytecode [72, 64]. En 1995, l'introduction des modules à la standard ML [66] et l'ajout d'un compilateur produisant du code natif, augmente la performance et la popularité de Caml. En 1996, une couche orientée objet est ajoutée à Caml [100, 101] et l'implantation devient alors Objective Caml [73].

Coq est un assistant de preuve basé sur le Calcul des Constructions [21], une puissante logique constructive issue de la théorie des types de Per Martin-Löf. Cette logique a ensuite été enrichie par des types inductifs, qui sont la contrepartie logique des types concrets de ML, obtenant ainsi le Calcul des Constructions Inductives [22]. Le langage de spécification et de preuve de Coq ressemble donc beaucoup à un langage fonctionnel. Nous reviendrons sur cette similarité dans la section suivante.

2.1.3 Les langages fonctionnels et les systèmes de types

Typage statique

Le langage ML à son origine [83] apporte une plus grande sûreté au langage de tactiques de LCF en particulier parce qu'il est fortement typé. Ce typage n'apparaît pas syntaxiquement. Le typage statique d'un programme permet d'éliminer, dans une chaîne de compilation les programmes absurdes, par exemple l'addition d'une chaîne de caractères à un flottant. Le typage statique consiste à attribuer un type à chaque structure syntaxique, il faut alors vérifier la cohérence des types de ce programme. On dit alors que le programme est bien typé.

Les langages fonctionnels de la famille de ML, tout comme le λ-calcul, se munissent facilement d'un système de types.

Exemple : le λ-calcul simplement typé, avec comme type de base les entiers ι. Le langage peut porter syntaxiquement les types, on parle alors de présentation à la Church,

$$N, M ::= i \mid x \mid \lambda x : \tau.\ M \mid MN.$$

Les types sont composés du type de base entier ι et du type "flèche", qui est le type des abstractions. $\tau_1 \to \tau_2$ est le type d'une abstraction qui prend un paramètre de type τ_1 et retourne un terme de type τ_2.

$$\tau ::= \iota \mid \tau \to \tau.$$

Le jugement de typage

$$\Gamma \vdash M : \tau$$

se lit : " sous les hypothèses de typage Γ, le terme M est de type τ." Le système formel définissant le typage de ce langage est défini par le jeu de règles d'inférence suivant :

$$\Gamma \vdash i : \iota \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma + x : \tau \vdash M : \tau'}{\Gamma \vdash \lambda x : \tau.\ M : \tau \to \tau'} \qquad \frac{\Gamma \vdash M : \tau \to \tau' \quad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \tau'}$$

Si l'on trouve un type τ via l'application de ces règles à un terme M, alors on dit que M est bien typé.

Une autre présentation où le typage est implicite, c'est-à-dire où la syntaxe ne porte pas les types, est la présentation à la Curry. Déterminer si un terme est bien typé devient alors plus difficile : il faut deviner le type du paramètre d'une fonction à partir de ses utilisations dans le corps de la fonction. C'est ce qu'on appelle l'inférence de types.

Par construction, un terme bien typé ne produit pas d'erreur [83]. Autrement dit, l'évaluation d'un terme bien typé conduit soit à une valeur soit à une divergence soit à un terme bloqué (qui n'a pas de réduction).

Les langages fonctionnels de la famille ML supportent le polymorphisme paramétrique, par exemple la fonction identité $\lambda x.\ x$ est polymorphe : elle est correcte quel que soit le type de x. De même, des structures de données peuvent être polymorphes comme le type des listes

```
type 'a list = Nil | Cons of 'a * 'a list
```

On peut alors définir des modules autour de telles structures, comme le module des listes de Caml ou bien encore de Coq.

L'inférence de types en présence de polymorphisme est plus complexe que dans le cas des types simples, mais reste possible avec l'algorithme W de Damas et Milner [25].

Isomorphisme de Curry-Howard

Dès l'origine du λ-calcul, le rapprochement entre le système de types et le système logique propositionnel est fait. Il y a un isomorphisme entre les programmes du λ-calcul et les preuves dans le système logique propositionnel, basé sur une similitude entre les arbres d'inférence de type et les arbres de déduction logique. L'isomorphisme de Curry-Howard [24, 53] met en correspondance le type des fonctions, $a \to b$ et l'implication $a \Rightarrow b$. De même, l'isomorphisme s'étend des termes aux démonstrations : il y a une correspondance entre la preuve constructive de $a \Rightarrow b$ (méthode pour passer d'une preuve de a à une preuve de b) et un terme de type $a \to b$ (une fonction qui prend un objet de type a et retourne un objet de type b). L'isomorphisme de Curry-Howard se caractérise par l'expression : "Programmer c'est prouver".

Le calcul des constructions [21], à la base du langage de programmation de Coq, est un λ-calcul typé d'ordre supérieur, les types sont des valeurs de première classe. Le système de types de ce langage est fortement normalisant, les programmes qui ne terminent pas ne sont pas typables et sont donc rejetés. Actuellement, le langage Coq se base sur un enrichissement du calcul des constructions avec types de données inductifs [22], le calcul des constructions inductifs.

Le calcul des constructions est une extension de l'isomorphisme de Curry-Howard : les termes de preuve Coq sont des λ-termes. L'assistant de preuve Coq est une concrétisation de l'isomorphisme de Curry-Howard. En particulier, le *mécanisme d'extraction* permet d'obtenir un programme à partir d'une preuve.

Le mécanisme d'extraction

Le mécanisme d'extraction de Coq est le processus qui permet d'obtenir un programme fonctionnel pur exprimé dans un langage cible (Haskell, SML ou OCaml) à partir d'un terme Coq. Ce mécanisme est intégré à Coq dès le début. Le premier mécanisme d'extraction mis en place est le fruit de la thèse de C .Paulin [91]. Dans la version actuelle, le mécanisme d'extraction de preuve est celui mis en place par P .Letouzey [75], qui apporte une adaptation vis-à-vis de l'évolution de Coq, une garantie que l'exécution du terme extrait est correcte, ainsi que son bon typage dans le langage hôte. La formalisation de la vérification du mécanisme d'extraction de Coq est un travail en cours réalisé actuellement par S. Glondu [41], dans le cadre de sa thèse doctorale.

2.1.4 Syntaxe de miniML

Nous isolons, ici, formellement le fragment purement fonctionnel de ML que nous traitons. Toutefois, il ne s'agit pas du langage source du compilateur formellement vérifié pour miniML. Ce dernier, εML, sera présenté à la section 2.2, seule la représentation des variables diffère.

Nous avons fait le choix de considérer miniML non typé. Nous reposer sur le seul typage dynamique (typage implicite des valeurs sémantiques à l'évaluation), nous permet de garder une grande expressivité. De plus, des travaux de vérification d'inférence de types ont été menés, en Coq [35, 34] ainsi qu'en Isabelle/HOL [87]. Nous pourrions imaginer utiliser l'un de ces travaux en amont de notre chaîne de compilation. Enfin, dans notre scénario idéal, nous désirons compiler les programmes issus du mécanisme d'extraction de Coq [75]. Il s'agit donc de programmes ayant passé l'inférence et la vérification de typage de Coq [20] ; ils sont donc bien typés.

Nous définissons formellement le fragment purement fonctionnel de ML non typé, avec variables nommées. Nous commençons par présenter la syntaxe de miniML par sa grammaire dans la forme BNF :

Terme :	$t ::= x$		variable
		$\mid \lambda x.\ t$	abstraction
		$\mid t_1\ t_2$	application
		$\mid \texttt{let}\ x\ =\ t_1\ \texttt{in}\ t_2$	définition locale
		$\mid \texttt{letrec}\ x_1\ x_2\ =\ t_1\ \texttt{in}\ t_2$	définition récursive
		$\mid C\ (t_1; \ldots; t_n)$	constructeur appliqué
		$\mid \texttt{match}\ t_1\ \texttt{with}\ \pi_1; \ldots; \pi_n$	filtrage

Motif : $\pi ::= C\ x_1; \ldots; x_n \rightarrow t$

Type concret : $\tau ::= C_0 \ldots C_n$

Programme : $prog ::= \{\texttt{type}\ :\ \tau_1; \ldots; \tau_n\ ;\ \texttt{terme}\ :\ t\}$

Un programme miniML n'est pas un simple terme, mais un terme (`terme`) accompagné de la liste (`types`) des déclarations de types concrets. miniML n'étant pas typé, la déclaration d'un type concret est plus simple qu'en ML : elle liste simplement tous les constructeurs du type, mais ne spécifie ni leur arité ni les types de leurs arguments.
Les termes miniML contiennent les termes du λ-calcul : une variable x représentée par son nom, l'abstraction unaire $(\lambda x.\ t)$ de paramètre formel x et de corps t et l'application unaire $(t_1\ t_2)$ du terme t_1 au terme t_2.

En plus du λ-calcul, les termes miniML offrent la possibilité de lier localement une variable et de définir des fonctions récursives. Le terme `let` $x\ =\ t_1$ `in` t_2, lie la variable x au terme t_1 dans le terme t_2. `letrec` $x_1\ x_2\ =\ t_1$ `in` t_2 définit la fonction récursive locale x_1, de paramètre formel x_2 et de corps t_1 dans le terme t_2.

Enfin, la présence des types concrets se reflète dans la syntaxe de miniML via deux sortes de termes. Le terme $C\ (t_1; \ldots ; t_n)$ désigne l'application du constructeur C à ses n arguments.
Le terme `match` t `with` $\pi_1; \ldots ; \pi_n$ effectue un filtrage de la valeur du terme t. Chacun des cas du filtrage est composé d'un motif $C\ (x_1; \ldots ; x_n)$ et d'une action t.

L'ensemble des exemples présentés dans la section 2.1.1, s'implante naturellement et sous la même forme en εML. Cependant, notons que nous ne traitons pas les fonctions mutuellement récursives.

Exemple : Le programme p suivant est un exemple de code miniML :

```
{ type : (( 0, S )) ;
  terme :
   letrec plus n =
           \m. match n with
               | S k -> S (plus k m)
               | 0 -> m
   in
    let y = S (S 0) in
        (\x. plus x (S 0))y }
```

Dans la définition de type, nous reconnaissons le type `nat` à deux constructeurs. Dans le corps de programme, on commence par définir la fonction `plus`, comme présentée dans la section 2. La fonction `plus` est une fonction récursive effectuant un filtrage exhaustif sur son premier paramètre. La portée lexicale de la fonction `plus` est le terme `let y =S (S 0) in (\x. plus x (S 0)) y` . Il s'agit de la liaison locale de l'entier naturel 2 à la variable y. Enfin, l'abstraction anonyme `(\x. plus x (S 0))` est appliquée à l'argument y.

2.1.5 Sémantique de miniML

La syntaxe d'un langage de programmation ne suffit pas à définir ce dernier. La sémantique d'un langage définit le lien entre la syntaxe et l'effet des "ordres" décrits par la syntaxe. La sémantique d'un langage est un système formel définissant le comportement, le sens, de chacune des constructions syntaxiques du langage par rapport aux environnements dans lequel il se trouve. Concernant les langages fonctionnels, leurs sémantiques peuvent être définies de différentes manières. À leur base se trouve toujours un ensemble de règles

de réduction, que l'on peut voir comme une réécriture du terme. Pour le λ-calcul la règle de réduction de base est la β-réduction :

$$(\lambda x.\ M)\ N \xrightarrow{\beta} M\{x \leftarrow N\}.$$

Un terme de la forme $(\lambda x.\ M)\ N$ est appelé β-redex et on note $M\{x \leftarrow N\}$ la substitution dans le terme M de la variable x par le terme N, sans capture de variable libre.

Dans cette section, nous rappelons les propriétés des séquences de réductions dans le λ-calcul, avant de présenter les différentes stratégies d'évaluation.

Les propriétés de l'évaluation dans le λ-calcul

Le λ-calcul présente des propriétés d'évaluation. Notons $\rightarrow$ une réduction par β-réduction, $\rightarrow^*$ 0,1 ou plusieurs β-réduction(s) et $\leftrightarrow^*$ la relation de β-équivalence entre termes.

La préservation du typage par réduction (ce théorème est connu sous le nom de *subject reduction*).

$$\text{Si } \Gamma \vdash M : \tau \text{ et } M \rightarrow N \text{ alors } \Gamma \vdash N : \tau.$$

La confluence Dans un λ-terme, il peut y avoir plusieurs β-redex. On peut donc avoir plusieurs séquences de réductions pour un même terme. La confluence est la propriété qui établit l'indifférence du choix de la prochaine réduction à effectuer. En effet, quelle que soit la séquence de réduction effectuée sur un terme, ces séquences convergent vers un même terme.

$$\text{Si } M \rightarrow N_1 \text{ et } M \rightarrow N_2 \text{ alors il existe } N \text{ tel que } N_1 \rightarrow^* N \text{ et } N_2 \rightarrow^* N.$$

Church-Rosser On dit que deux termes sont β-équivalents s'ils sont dans la relation $\leftrightarrow^*$. La propriété de Church-Rosser est la propriété qui énonce la confluence des réductions de deux termes β-équivalents.

$$\text{Si } M_1 \leftrightarrow^* M_2 \text{ alors il existe } M \text{ tel que } M_1 \rightarrow^* M \text{ et } M_2 \rightarrow^* M.$$

Il est donc facile d'emprunter une séquence de β-réductions de taille plus ou moins importante en termes de nombre d'étapes de réduction. De plus, le choix du prochain β-redex à réduire, dans une séquence de réduction, est non déterministe. Certains choix peuvent mener à la divergence (suite infinie de réductions) ; d'autres débouchent sur la même valeur finale, mais par des chemins plus ou moins longs en terme de nombre de réductions.

Ce non-déterminisme ne facilite pas la tâche du programmeur, qui ne contrôle pas la terminaison ou la complexité en temps de ses programmes. Pour cette raison, il est souhaitable de déterminiser les calculs en spécifiant des *stratégies d'évaluation*.

Les stratégies d'évaluation

L'évaluation d'un langage est régie par un ensemble de règles de réduction. Il y a différentes manières d'appliquer ces règles de réduction, il s'agit des stratégies d'évaluation.

Réduction forte *versus* réduction faible : Une stratégie de *réduction forte* est une stratégie où les règles de réduction peuvent être appliquées dans le corps des fonctions, c'est-à-dire sous les λ. A l'inverse, une stratégie de *réduction faible* est une stratégie où les réductions ne traversent pas les λ. Les stratégies de réduction forte sont nécessaires dans des systèmes de preuves comme Coq, et plus généralement, pour effectuer des calculs symboliques. Les stratégies d'évaluation faible se prêtent plus facilement aux langages de programmation et à l'exécution de programmes. Nous intéressant à un langage de programmation, nous optons pour une stratégie d'évaluation faible.

Considérons l'application $(\lambda x.\ M)\ N$.

Appel par valeur *versus* appel par nom : La stratégie d'*appel par nom* (aussi appelée *normal order*). Dans une stratégie d'appel par nom, on n'évalue pas N, on le passe tel quel :

$$(\lambda x.\ M)\ N \xrightarrow{\beta} M\{x \leftarrow N\}.$$

C'est une stratégie *normalisante* : si le λ-terme a une forme normale, alors la stratégie d'appel par nom retrouve cette forme.

Le dual de la stratégie d'appel par nom est la stratégie d'*appel par valeur* (aussi appelée *applicative order*). Dans cette stratégie, on évalue d'abord N en une valeur v avant d'effectuer la β-réduction :

$$(\lambda x.\ M)\ v \xrightarrow{\beta} M\{x \leftarrow v\} \text{ où } v \text{ est une valeur.}$$

Ce n'est pas une stratégie normalisante. Considérons $(\lambda x.\ 0)\ M$. Ce terme a pour forme normale 0. Si M diverge alors en appel par valeur le terme entier diverge bien qu'il ait une forme normale.

Considérons à présent le terme $(\lambda x.\ x + x)\ M$, en appel par valeur, le terme M n'est évalué qu'une fois avant l'application de la β-réduction. En appel par nom, M sera évalué deux fois, après la β-réduction. Bien qu'étant une stratégie normalisante, l'appel par nom est en pratique inutilisable pour des raisons de performances car des calculs seraient dupliqués.

La stratégie d'*appel par nécessité* évite cette duplication en évaluant un terme M au moment de sa première utilisation, en mémorisant la valeur ainsi obtenue, et en réutilisant cette valeur lorsque M est utilisé plus tard. Cette stratégie, comme l'appel par nom, est une stratégie normalisante, mais garantit qu'un terme sera évalué au plus une fois.

Dans les faits, la stratégie d'appel par valeur reste la plus simple à implanter efficacement. C'est donc la stratégie que nous traitons dans notre chaîne de compilation.

Formalisation de l'évaluation : sémantiques opérationnelles

La description formelle du comportement calculatoire d'un langage est donnée par sa *sémantique opérationnelle*. Il s'agit d'un système formel qui décrit les sites d'application des règles de réduction ce qui permet de définir formellement les stratégies d'évaluation choisies. On utilise aussi le terme de *sémantique dynamique*. Il y a deux manières de formaliser une sémantique opérationnelle selon le degré d'observation que l'on désire préciser et/ou observer.

La sémantique à réductions ou bien encore "à petit pas" (*small-step* dans la littérature anglophone). Il s'agit de décrire une évaluation par réduction. On commence donc par définir formellement une étape de réduction. Il y a deux présentations possibles.

La *sémantique structurelle* (S.O.S.) est présentée par Plotkin [96]. Le sémantique est alors décrite au travers d'un système de règles d'inférence définissant un jugement de la forme

$$M \to N.$$

Parmi ces règles, on distingue :

- Les règles de réduction où M est un redex. Par exemple, en appel par valeur, $(\lambda x.N)\ v$ est un βv-redex (où v est une valeur). On lui applique alors la βv-réduction.
- Les règles de contexte qui s'appliquent lorsque le terme n'est pas un redex. Ces règles contrôlent l'ordre d'évaluation des sous-termes et l'aspect "fort" ou "faible" de la stratégie (voir la section 2.2.3).

Une autre présentation de la sémantique à réduction est la *sémantique contextuelle*, présentée par Felleisen et Wright [119]. Les règles de réduction sont définies comme des règles de réécriture. La stratégie d'évaluation est formalisée au travers de contextes à un trou. Par exemple, pour le λ-calcul pur par appel par valeur de gauche à droite, la règle de réduction est la βv-réduction et les contextes sont définis comme suit :

$$\mathbb{C} ::= [] \ \mid \ \mathbb{C}\ N \ \mid \ (\lambda x.\ M)\ \mathbb{C}.$$

Quelle que soit sa présentation, après avoir formalisé un système décrivant une étape de réduction, on décrit l'exécution d'un programme comme une séquence de réductions successives. Trois cas se présentent :

- Terminaison : une séquence finie de réductions qui se termine sur une valeur

$$M \to M_1 \to \cdots \to v$$

- Divergence : une séquence infinie de réductions

$$M \to M_1 \to \cdots \to \cdots$$

- Erreur : une séquence finie de réductions qui se termine sur un terme qui n'est pas une valeur mais ne se réduit pas, comme par exemple $C()\ C()$ (un constructeur appliqué comme si c'était une fonction). On dit alors que le terme est bloqué.

$$M \to M_1 \to \cdots \to M_n \not\to$$

La sémantique opérationnelle décrite sous forme d'une sémantique à réduction permet donc une observation et une description fine de l'évaluation des termes, puisque cette évaluation est décrite pas à pas. La sémantique à réduction est particulièrement adéquate à la formalisation d'évaluation de langages pour lesquels on désire observer la non terminaison.

La sémantique naturelle ou bien encore "à grand pas" (*big step* dans la littérature anglophone) est une autre manière de décrire formellement l'évaluation d'un langage. Elle est présentée par G.Kahn [58]. Le système est sous forme d'un jeu de règles d'inférence dirigées par la syntaxe décrivant le jugement suivant :

$$M \Rightarrow v,$$

qui se lit : "le terme M s'évalue en la valeur v". L'observation et la description sont beaucoup moins fines qu'en sémantique à réductions, les règles sont définies de telle manière que chacune décrit en une règle la séquence de réductions du terme entier jusqu'à sa valeur. Plus intuitive, car plus proche de la syntaxe, la sémantique naturelle ne permet pas d'observer la non terminaison.

2.1.6 Des variables qui veulent s'échapper : noms versus indices

Nous discutons ici des variables, de leur liaison et de leur portée lexicale.

Dans notre langage source, les variables sont définies localement par des *lieurs* tels que :

La liaison locale (`let`) Considérons le terme : `let` $x = t$ `in` $x + (f\ x)$, la variable x est liée à la valeur de t dans $x + (f\ x)$.

Les paramètres formels des fonctions Le passage des arguments d'une application se fait via des variables, les paramètres formels. Par exemple dans l'abstraction : $\lambda\, x.\ x + 2 * x$, la variable x est liée dans le corps de l'abstraction $x + 2 * x$.

La définition d'abstraction récursive (`letrec`) Considérons le terme `letrec` $f\ x = x + f\ (x - 1)$ `in` $f\ 3$, la variable f désigne le nom de la fonction récursive et est liée dans les deux sous-termes. La variable x désigne le paramètre formel de l'abstraction et est liée uniquement dans le corps de l'abstraction.

Ces lieurs ont pour rôle de définir le domaine de validité des variables qu'ils lient, c'est-à-dire leur *portée lexicale*. La portée lexicale d'une variable est le sous-terme dans lequel elle est liée, c'est-à-dire qu'elle y a un sens du point de vue de la sémantique dynamique. Une variable considérée dans sa portée lexicale est dite *liée*, dans le cas contraire, elle est dite *libre*. Dans le terme $\lambda\, x.x + y$, la variable x est liée tandis que la variable y est libre.

De nombreuses preuves formelles de transformations de programme rencontrent des problèmes avec les variables et leurs liaisons. Lors d'une substitution de la variable x dans le terme t par le terme t' : $t\{x \leftarrow t'\}$, il faut éviter de *capturer* une variable libre de t'. Par exemple, si on substitue x par $y + 1$ dans $\lambda\ y.\ y\ x$ on obtient le terme $\lambda\ y.\ y\ (y + 1)$ et le y de $y + 1$ a été capturé par la liaison de l'abstraction. Pour éviter cela, il faut *renommer* les variables liées (α-conversion). Ainsi, dans le terme $\lambda\ y.\ y\ x$, on peut effectuer un *renommage* en remplaçant toutes les occurrences de y par z. On obtient alors le terme *α-équivalent* $\lambda\ z.\ z\ x$. La substitution de x par $y + 1$ dans ce terme ne produit plus de capture de la variable y.

Ces mécanismes de renommage et de substitution sans capture sont simples à utiliser manuellement dans des preuves "sur papier", mais très difficiles à formaliser dans des preuves "sur machine". Ces problèmes sont récurrents dans les formalisations et la vérification formelle de sémantiques de langages, de systèmes de types ou bien encore de transformations de programmes. Des efforts importants sont engagés, afin de résoudre cette difficulté. Le "POPLmark challenge" [7] donne une vue d'ensemble des principales solutions connues, telles que la syntaxe abstraite d'ordre supérieur [94], l'approche *locally nameless* [79, 43], la logique nominale [113], ou encore les indices de de Bruijn, l'approche que nous allons maintenant détailler.

Les *indices de de Bruijn* [31] sont un formalisme dans lequel chaque variable est représentée non plus par un nom, mais par un indice, qui n'est autre qu'un entier. Cet indice représente la position relative de la variable par rapport au lieur l'ayant déclarée dans l'arbre de syntaxe abstraite. Dans ce formalisme, la notion d'α-conversion est caduque. Un terme

a une unique représentation. Les termes $\lambda\ x.\ x$ et $\lambda\ y.\ y$ deviennent $\lambda.\ 0!$ dans le formalisme de de Bruijn. (On note $n!$ la variable d'indice n.)

Les variables libres se repèrent arithmétiquement. La *profondeur* d'un sous-terme dans un terme correspond au nombre de lieurs au-dessus du sous-terme dans l'arbre représentant le terme. Une variable libre dans un terme t est une variable dont l'indice est supérieur à sa profondeur dans t. Par exemple, considérons le terme $(\lambda.\ 0! + 1!)$ Dans le corps de cette abstraction, $0!$ désigne le paramètre formel lié par λ et $1!$ est une variable libre. En effet, la profondeur est ici de 0 et $1 > 0$. Si maintenant ce terme est placé sous un `let`, comme dans `let` t `in` $(\lambda.\ 0! + 1!)$, la variable $1!$ est la variable qui est liée par le `let`.

Pour garantir la non capture de variables libres lors d'une substitution, il suffit d'incrémenter tous les indices des variables libres du substituant par la profondeur du site de substitution.

Dans notre compilateur, nous avons choisi d'utiliser les indices de de Bruijn aussi bien pour le langage source que pour le premier groupe de nos langages intermédiaires. Le second groupe de langages intermédiaires utilise des variables nommées. Les noms de ces variables sont générés au cours de la compilation. Nous spécifions cette génération de noms que nous voulons unique. Ayant une garantie sur l'unicité des noms générés, les problèmes liés aux noms de variables et à leurs lieurs sont gérés plus facilement dans notre spécification.

2.2 Le langage source de notre compilateur : εML

Nous présentons maintenant le langage source : εML, soit miniML (voir la section 2.1.4) dans le formalisme de de Bruijn. Nous présenterons aussi une variante de εML qui se prête mieux à la substitution.

2.2.1 Syntaxe avec définition locale de fonctions récursives

La grammaire suivante décrit εML :

Termes : $t ::= n!$ — variable d'indice n
$\mid \lambda.\ t$
$\mid t_1\ t_2$
$\mid$ `let` t_1 `in` t_2
$\mid$ `letrec` t_1 `in` t_2
$\mid C\ (t_1; \ldots; t_n)$
$\mid$ `match` t_1 `with` $\pi_1; \ldots; \pi_n$

Motif : $\pi ::= C\ n \rightarrow t$

Type concret : $\tau ::= C_0 \ldots C_n$

Programme : $prog ::= \{$`type` $:\ \tau_1; \ldots; \tau_n\ ;$ `terme` $:\ t\}$

Comme attendu les variations apparaissent au niveau des variables et de leurs lieurs. $n!$ désigne la variable d'indice n ; c'est-à-dire la variable liée $(n + 1)$ lieurs au-dessus dans

l'arbre de syntaxe abstraite. $\lambda.\ t$ est l'abstraction unaire de corps t dans lequel le paramètre formel est désigné par la variable 0!.
De même, la liaison locale `let` t_1 `in` t_2 lie la variable 0! à t_1 dans t_2.
Considérons la définition récursive `letrec` t_1 `in` t_2 et la représentation des variables nouvellement liées. La variable désignant la fonction récursive est désignée par l'indice 1! dans le corps de la fonction t_1 et par l'indice 0! dans le sous-terme droite t_2. Le paramètre formel est quant à lui désigné par la variable 0! dans t_1.
Enfin, dans une clause $C\ n \rightarrow t$ les n paramètres formels sont désignés par les indices $(n-1)! \ldots 0!$ dans l'action t.

Le programme, précédemment présenté (voir page 24), devient dans le formalisme de de Bruijn :

```
{ type : ( 0, S ) ;
  terme :
   letrec
           \. match 1 ! with
               | S 1 -> S (3 ! 0 ! 1 !)
               | 0 0 -> 0 !
   in
    let (S (S 0)) in
        (\. 2 ! 0 ! (S 0)) 1 ! }
```

2.2.2 Une syntaxe plus appropriée à la substitution

Pour certaines transformations, telles que la mise en style par passage de continuations (voir le chapitre 4), il est parfois nécessaire de considérer une variante syntaxique de nos langages plus appropriée à la substitution. Dans cette variante, la définition locale de fonctions récursives et la construction syntaxique `letrec` laissent place à l'abstraction récursive μ. Nous présentons ici une variante de εML avec abstraction récursive. Ce langage peut être vu comme une variante du miniML présenté dans le premier chapitre de [65]. Cette présentation est plus proche du λ-calcul.

Syntaxe εML avec abstraction récursive

Cette variante de εML ne diffère que par l'absence de la définition de fonctions récursives. La structure syntaxique `letrec` disparaît. La récursivité est définissable par les abstractions récursives

$$\mu.\ t \text{ avec variables nommées } \mu\ f\ x.\ t$$

qui désigne une abstraction récursive de corps t. Dans ce corps, le paramètre récursif, désignant la fonction elle-même est la variable d'indice 1 (f), 1! et le paramètre formel est la variable d'indice 0 (x), 0!.

Le passage de εML avec définition locale de fonctions récursives à εML avec abstractions récursives consiste à transformer les termes de la forme `letrec` t_1 `in` t_2 en `let` $\mu.\ t'_1$ `in` t'_2, où t'_1 et t'_2 sont des transformations de t_1 et t_2. Symétriquement, l'abstraction récursive $\mu.\ t$ s'implante dans εML avec définition locale de fonction récursive comme suit : `letrec` t `in` 0!.

Nous avons montré la préservation sémantique d'une transformation d'une variante de langage à la εML avec définitions locales récursives vers une variante avec abstractions récursives 3.4.2.

Substitution simultanée

Nous sommes maintenant en mesure de définir la *substitution simultanée* dans un formalisme de de Bruijn. Par substitution simultanée nous entendons substituer n variables par n termes $t_1; \ldots; t_n$ dans un même terme t de manière simultanée.

Dans un formalisme à la de Bruijn, la non capture de variables libres lors de la substitution nécessite un mécanisme d'actualisation des indices des variables (*lifting*) comme nous l'avons illustré plus haut (voir la section 2.1.6).

Le tableau suivant récapitule les notations des opérations de substitutions et d'actualisation avec indices de de Bruijn.

Acutalisation d'indice	
de 1 de tous les indices libres dans le terme t	$\Uparrow t$
de n de tous les indices libres dans le terme t	$\Uparrow_n t$
de n de tous les indices supérieurs ou égaux à i libres dans le terme t	$\Uparrow_n^i t$
Substitution dans le terme t	
de tous les indices libres de 0 à n par les termes $t_0; \ldots; t_n$	$t\{t_0; \ldots; t_n\}$
des variables d'indice de i à $i+n$ libres par les termes $t_0; \ldots; t_n$	$t\{t_0; \ldots; t_n\}_i$

Avec pour définition de $\Uparrow_n^i t$ la fonction suivante :

$$
\begin{aligned}
\Uparrow_n^i x! &= \begin{cases} x! & \text{si } x < i\,; \\ (x+n) & \text{sinon} \end{cases} \\
\Uparrow_n^i \lambda.\, t &= \lambda.\ \Uparrow_n^{i+1} t \\
\Uparrow_n^i \mu.\, t &= \mu.\ \Uparrow_n^{i+2} t \\
\Uparrow_n^i t_1\ t_2 &= \Uparrow_n^i t_1\ \Uparrow_n^i t_2 \\
\Uparrow_n^i \texttt{let}\ t_1\ \texttt{in}\ t_2 &= \texttt{let}\ \Uparrow_n^i t_1\ \texttt{in}\ \Uparrow_n^{i+1} t_2 \\
\Uparrow_n^i C\ (t_1; \ldots; t_m) &= C\ (\Uparrow_n^i t_1; \ldots; t_m) \\
\Uparrow_n^i \texttt{match}\ t\ \texttt{with}\ \pi_1; \ldots; \pi_m &= \texttt{match}\ \Uparrow_n^i t\ \texttt{with}\ \Uparrow_n^i \pi_1; \ldots; \pi_m
\end{aligned}
$$

$$
\Uparrow_n^i C\ m\ t = C\ m\ \Uparrow_n^{i+m} t
$$

Et pour définition de $t[t_0; \ldots; t_n\}_i$ la fonction suivante :

$$
x!\{t_0; \ldots; t_n\}_i = \begin{cases} x! & \text{si } x < i\,; \\ t_{n-x} & i \leq x < i+n+1\,; \\ (x-n-1)! & \text{sinon} \end{cases}
$$

$$
\begin{array}{rcl}
(\lambda.\ t)\{t_0;\ldots;t_n\}_i & = & \lambda.\ (t\{\Uparrow t_0;\ldots;t_n\}_{(i+1)}) \\
(\mu.\ t)\{t_0;\ldots;t_n\}_i & = & \mu.\ (t\{\Uparrow_2 t_0;\ldots;t_n\}_{(i+2)}) \\
(t_1\ t_2)\{t_0;\ldots;t_n\}_i & = & t_1\{t_0;\ldots;t_n\}_i\ t_2\{t_0;\ldots;t_n\}_i \\
(\texttt{let}\ t_1\ \texttt{in}\ t_2)\{t_0;\ldots;t_n\}_i & = & \texttt{let}\ t_1\{t_0;\ldots;t_n\}_i\ \texttt{in}\ t_2\{\Uparrow t_0;\ldots;t_n\}_{i+1} \\
(C\ (u_1;\ldots;u_n))\{t_0;\ldots;t_n\}_i & = & C\ (u_1;\ldots;u_n)\{t_0;\ldots;t_n\}_i \\
(\texttt{match}\ t\ \texttt{with}\ \pi_1;\ldots;\pi_m)\{t_0;\ldots;t_n\}_i & = & \texttt{match}\ t\{t_0;\ldots;t_n\}_i\ \texttt{with}\ \pi_1;\ldots;\pi_m\{t_0;\ldots;t_n\}_i \\
\\
(C\ m\ t)\{t_0;\ldots;t_n\}_i & = & C\ m\ t\{t_0;\ldots;t_n\}_{i+m}
\end{array}
$$

2.2.3 Sémantiques opérationnelles

La sémantique d'un langage est un système formel décrivant le comportement de chaque construction syntaxique du langage dans un environnement d'évaluation donné. Comme nous l'avons expliqué plus haut, il existe différentes manières de décrire la sémantique d'un langage (voir la section 2.1.5). La stratégie d'évaluation de εML est une stratégie de réduction faible par appel par valeur. Nous présentons ici la sémantique de εML de trois manières différentes et montrons l'équivalence entre ces trois présentations.

Sémantique à réduction

La sémantique à réductions de εML sous forme S.O.S., se définit par un jeu de règles d'inférence décrivant un pas de réduction. La sémantique à réductions nous permet d'observer pas à pas l'évaluation d'un terme, ce qui est parfaitement adéquat lorsque l'on veut observer la non terminaison. Une sémantique à réductions se définit autour des règles de réductions du langage.

On note

$$\Pi \vdash t \to t'$$

la réduction du terme t en un pas en le terme t' en considérant les définitions de types concrets Π. Pour pouvoir définir les réductions, nous avons besoin de différencier parmi les termes lesquels sont des valeurs. Une valeur se réduit en elle-même. Nous notons *Val* l'ensemble des termes qui sont des valeurs. Nous notons $\Pi \vdash t \in \mathit{Val}$ le fait que t est une valeur en accord avec l'ensemble de définitions de types concrets Π. Les termes valeurs de εML sont définis comme suit :

$$\Pi \vdash \lambda.\ t \in \mathit{Val} \qquad \Pi \vdash \mu.\ t \in \mathit{Val} \qquad \frac{\Pi(\tau) = l \quad C \in l \qquad \Pi \vdash v_i \in \mathit{Val} \text{ pour tout } i \in [1;n]}{\Pi \vdash C\ (v_1;\ldots;v_n) \in \mathit{Val}}$$

Les abstractions, aussi bien simples que récursives, sont des valeurs. Un constructeur appliqué à des valeurs est une valeur si le constructeur est défini dans Π.

Les valeurs sont des termes qui se réduisent en eux-mêmes :

$$\frac{\Pi(\tau) \;=\; l \quad C \in l \qquad \Pi \vdash t_i \in \mathit{Val} \text{ pour tout } i \in [1;n]}{\Pi \vdash C\ (t_1;\ldots;t_n) \rightarrow C\ (t_1;\ldots;t_n)}$$

Parmi les règles définissant la sémantique à réductions, nous pouvons distinguer les règles de réductions en tête de termes. Il s'agit des règles de réductions en tête de termes que nous retrouverions dans une présentation de la sémantique de réductions par contextes (Wright et Felleisen) :

$$\frac{\Pi \vdash t_0 \in \mathit{Val}}{\Pi \vdash \texttt{let}\ t_0\ \texttt{in}\ t \rightarrow t\{t_0\}} \qquad \frac{\Pi \vdash t_0 \in \mathit{Val}}{\Pi \vdash (\lambda.\ t)\ t_0 \rightarrow t\{t_0\}} \qquad \frac{\Pi \vdash t_0 \in \mathit{Val}}{\Pi \vdash (\mu.\ t)\ t_0 \rightarrow t\{t_0;\mu.\ t\}}$$

$$\frac{\tau \in \Pi \quad \texttt{exhaustive}\ (\pi_1;\ldots;\pi_n)\ \tau \qquad \Pi \vdash t_j \in \mathit{Val} \text{ pour tout } j \in [1,k] \quad (C)\ k \rightarrow t \in \pi_1;\ldots;\pi_n}{\Pi \vdash \texttt{match}\ C\ (t_1;\ldots;t_k)\ \texttt{with}\ \pi_1;\ldots;\pi_n \rightarrow t\{t_k;\ldots;t_1\}}$$

La liaison locale d'une valeur dans un terme, `let` t_0 `in` t se réduit en un pas en le terme t où la variable liée par le `let`, 0!, est substituée par t_0. L'application d'une abstraction simple à une valeur $(\lambda.\ t)\ t_0$ se (βv-)réduit en le corps de l'abstraction où le paramètre formel 0! est substitué par la valeur t_0. L'application d'une abstraction récursive à une valeur $(\mu.\ t)\ t_0$ se réduit en le corps de l'abstraction où le paramètre formel 0! est substitué par la valeur t_0 et le paramètre formel désignant l'abstraction récursive 1! est substitué par $\mu.\ t$. Le prédicat `exhaustive` détermine pour une liste de clauses et la définition d'un type concret si tous les constructeurs de ce type sont filtrés par cette liste de clauses. Enfin, un filtrage exhaustif dont le sujet est une valeur constructeur $C\ (t_1;\ldots;t_k)$ se réduit comme le corps de la clause t correspondant au motif $(C)\ k$ où les variables $(k-1)!;\ldots;0!$ ont simultanément été substituées par les k arguments du sujet.

Enfin, les règles de réduction de contexte permettent de réduire pas à pas les termes jusqu'à obtenir un terme valeur ou bien qu'une règle de réduction s'applique. Ce sont ces règles qui définissent la stratégie d'évaluation. Chacune des règles correspond à un contexte à trou dans une présentation à la Wright et Felleisen. Commençons par considérer la réduction d'une liste de termes :

$$\frac{\Pi \vdash t \rightarrow t'}{\Pi \vdash (t;l) \rightarrow (t';l)} \qquad \frac{\Pi \vdash t \in \mathit{Val} \quad \Pi \vdash l \rightarrow l'}{\Pi \vdash (t;l) \rightarrow (t;l')}$$

Comme attendu la liste vide de termes se réduit en la liste vide de termes. Les deux règles suivantes induisent l'évaluation des sous-termes de gauche à droite. On ne réduit un élément de la liste que si tous les éléments à sa gauche sont déjà des valeurs.

Les autres règles de réduction de contexte explicitent tout autant la stratégie d'évaluation :

$$\frac{\Pi(\tau) \;=\; l \quad C \in l \quad \Pi \vdash \vec{t} \rightarrow \vec{t'}}{\Pi \vdash C\ (\vec{t}) \rightarrow C\ (\vec{t'})} \qquad \frac{\Pi \vdash t_1 \rightarrow t}{\mathit{let}\ \texttt{t}_1\ \texttt{in}\ \texttt{t}_2 \rightarrow\ \texttt{let t in}\ \texttt{t}_2} \qquad \frac{\Pi \vdash t_1 \rightarrow t}{\Pi \vdash t_1\ t_2 \rightarrow t\ t_2}$$

$$\frac{\Pi \vdash t_2 \to t'}{\Pi \vdash (\lambda.\ t)\ t_2 \to (\lambda.\ t)\ t'} \qquad \frac{\Pi \vdash t_2 \to t'}{\Pi \vdash (\mu.\ t)\ t_2 \to (\mu.\ t)\ t'}$$

$$\frac{\tau \in \Pi \quad \texttt{exhaustive}\ (\pi_1; \ldots; \pi_n)\ \tau \quad \Pi \vdash t \to t'}{\Pi \vdash \texttt{match}\ t\ \texttt{with}\ \pi_1; \ldots; \pi_n \to\ \texttt{match}\ t'\ \texttt{with}\ \pi_1; \ldots; \pi_n}$$

L'absence de règle de réduction du sous-terme droit d'une liaison locale impose la réduction du sous-terme gauche avant l'application de la règle de réduction du `let`. De même, le sous-terme droit d'une application n'est évalué que lorsque le sous-terme gauche a déjà été évalué en une abstraction.

L'ensemble des règles de réduction est collecté sur la figure 2.2.1.

$$\frac{\Pi(\tau) \ = \ l \quad C \in l \qquad \Pi \vdash v_i \in \mathit{Val} \text{ pour tout } i \in [1; n]}{\Pi \vdash C\ (v_1; \ldots; v_n) \to C\ (v_1; \ldots; v_n)}$$

$$\frac{\Pi \vdash t_0 \in \mathit{Val}}{\Pi \vdash \texttt{let}\ t_0\ \texttt{in}\ t \to t\{t_0\}} \qquad \frac{\Pi \vdash t_0 \in \mathit{Val}}{\Pi \vdash (\lambda.\ t)\ t_0 \to t\{t_0\}} \qquad \frac{\Pi \vdash t_0 \in \mathit{Val}}{\Pi \vdash (\mu.\ t)\ t_0 \to t\{t_0; \mu.\ t\}}$$

$$\frac{\tau \in \Pi \quad \texttt{exhaustive}\ (\pi_1; \ldots; \pi_n)\ \tau \qquad \Pi \vdash t_j \in \mathit{Val} \text{ pour tout } j \in [1, k] \quad (C)\ k \to t \in \pi_1; \ldots; \pi_n}{\Pi \vdash \texttt{match}\ C\ (t_1; \ldots; t_k)\ \texttt{with}\ \pi_1; \ldots; \pi_n \to t\{t_k; \ldots; t_1\}}$$

$$\frac{\Pi(\tau) \ = \ l \quad C \in l \quad \Pi \vdash \vec{t} \to \vec{t'}}{\Pi \vdash C\ (\vec{t}) \to C\ (\vec{t'})} \qquad \frac{\Pi \vdash t_1 \to t}{\mathit{let}\ \texttt{t}_1\ \texttt{in}\ \texttt{t}_2 \to\ \texttt{let t in}\ \texttt{t}_2}$$

$$\frac{\Pi \vdash t_1 \to t}{\Pi \vdash t_1\ t_2 \to t\ t_2} \qquad \frac{\Pi \vdash t_2 \to t'}{\Pi \vdash (\lambda.\ t)\ t_2 \to (\lambda.\ t)\ t'} \qquad \frac{\Pi \vdash t_2 \to t'}{\Pi \vdash (\mu.\ t)\ t_2 \to (\mu.\ t)\ t'}$$

$$\frac{\tau \in \Pi \quad \texttt{exhaustive}\ (\pi_1; \ldots; \pi_n)\ \tau \quad \Pi \vdash t \to t'}{\Pi \vdash \texttt{match}\ t\ \texttt{with}\ \pi_1; \ldots; \pi_n \to\ \texttt{match}\ t'\ \texttt{with}\ \pi_1; \ldots; \pi_n}$$

$$\frac{\Pi \vdash t \to t'}{(t; l) \to (t'; l)} \qquad \frac{\Pi \vdash t \in \mathit{Val} \quad \Pi \vdash l \to l'}{\Pi \vdash (t; l) \to (t; l')}$$

FIG. 2.2.1 – Sémantique à réductions de εML

L'évaluation d'un terme dans son intégralité se définit par une séquence d'applications de règles de réduction. On note $\to^*$ la séquence de zéro, une ou plusieurs étapes de réduction (fermeture réflexive transitive de $\to$).

$$\Pi \vdash \lambda.\ t \Rightarrow \lambda.\ t \qquad \Pi \vdash \mu.\ t \Rightarrow \mu.\ t$$

$$\frac{\Pi(\tau) \ = \ l \quad C \in l \quad \Pi \vdash t_i \Rightarrow v_i \quad \text{pour tout } i \in [1;n]}{\Pi \vdash C\ (t_1;\ldots;t_n) \Rightarrow C\ (v_1;\ldots;v_n)}$$

$$\frac{\Pi \vdash t_1 \Rightarrow v_1 \quad \Pi \vdash t_2\{v_1\} \Rightarrow v}{\Pi \vdash \texttt{let}\ t_1\ \texttt{in}\ t_2 \Rightarrow v} \qquad \frac{\Pi \vdash t_1 \Rightarrow (\lambda.\ t) \quad \Pi \vdash t_2 \Rightarrow v_2 \quad \Pi \vdash t\{v_2\} \Rightarrow v}{\Pi \vdash t_1\ t_2 \Rightarrow v}$$

$$\frac{\Pi \vdash t_1 \Rightarrow (\mu.\ t) \quad \Pi \vdash t_2 \Rightarrow v_2 \quad \Pi \vdash t\{v_2; \mu.\ t\} \Rightarrow v}{\Pi \vdash t_1\ t_2 \Rightarrow v}$$

$$\frac{\begin{array}{c}\Pi \vdash t \Rightarrow C\ (v_1;\ldots;v_k) \quad \tau \in \Pi \quad \texttt{exhaustive}\ (\pi_1;\ldots;\pi_n)\ \tau \\ \pi_i \ = \ C\ k \rightarrow t_i \quad \Pi \vdash t_i\{v_k;\ldots;v_1\} \Rightarrow v\end{array}}{\Pi \vdash \texttt{match}\ t\ \texttt{with}\ \pi_1;\ldots;\pi_n \Rightarrow v}$$

FIG. 2.2.2 – Sémantique naturelle par substitution de εML

Sémantique naturelle par substitution

La sémantique naturelle d'un langage permet une observation moins fine de l'évaluation d'un terme. Sa définition est plus intuitive et proche de la définition syntaxique du langage. Les règles définissant une sémantique naturelle sont dites dirigées par la syntaxe. Les règles d'évaluation sont moins nombreuses que dans une sémantique à réduction. L'idée est de décrire la sémantique d'une structure syntaxique en une règle tout en décrivant l'évaluation des sous-termes vers des termes valeurs. Il n'est plus nécessaire d'avoir des règles de réduction et des règles de contexte, puisque le contexte se retrouve à travers la structure syntaxique. Le jugement d'évaluation d'un terme εML dans la sémantique naturelle par substitution s'énonce comme suit :

$$\Pi \vdash t \Rightarrow v$$

Le terme t s'évalue en le terme valeur v en considérant les définitions de types concrets Π.

Le système de règles d'inférence définissant la sémantique naturelle de εML est donné sur la figure 2.2.2. Tout comme dans la sémantique à réductions, les abstractions s'évaluent en elles-mêmes. S'il existe dans Π, un constructeur C appliqué à $t_1;\ldots;t_n$ s'évalue en le même constructeur appliqué à $v_1;\ldots;v_n$, où $v_1;\ldots;v_n$ sont les valeurs issues de l'évaluation des termes $t_1;\ldots;t_n$.

L'évaluation d'une liaison locale $\texttt{let}\ t_1\ \texttt{in}\ t_2$ s'évalue en v si le terme t_1 s'évalue en v_1 et le terme $t_2\{v_1\}$ s'évalue en v. Nous voyons clairement les combinaisons de séquences de réduction menant à cette règle d'évaluation. On réduit le sous-terme gauche tant qu'il ne s'agit pas d'une valeur, soit n_1 séquence de :

$$\frac{\Pi \vdash t_1 \rightarrow t_1'}{\Pi \vdash \texttt{let}\ t_1\ \texttt{in}\ t_2 \rightarrow \texttt{let}\ t_1'\ \texttt{in}\ t_2}$$

Soit : $\texttt{let}\ t_1\ \texttt{in}\ t_2 \rightarrow^* \texttt{let}\ v_1\ \texttt{in}\ t_2$.

On peut alors appliquer la règle de réduction du `let` :

$$\texttt{let } v_1 \texttt{ in } t_2 \rightarrow t_2\{v_1\}$$

Enfin, une séquence de n_2 réductions telle que : $t_2\{v_1\} \rightarrow^* v$.

On peut observer des découpages similaires des règles d'évaluation en séquences de réduction pour chaque règle définissant la sémantique à grands pas. Ces observations sont à la base de la preuve d'équivalence entre les deux sémantiques que nous expliciterons plus bas (voir la section 2.2.3).

Concernant l'évaluation d'une application, nous avons deux règles d'évaluation, selon que l'abstraction mise en jeu est simple ou récursive. Si t_1 s'évalue en l'abstraction simple $\lambda.\ t$, que t_2 s'évalue en v_2 alors $t_1\ t_2$ s'évalue en le résultat de l'évaluation de $t\{v_2\}$. Dans le cas où t_1 s'évalue en $\mu.\ t$, $t_1\ t_2$ s'évalue en le résultat de l'évaluation $t\{v_2; \mu.\ t\}$.

Enfin, pour évaluer un filtrage $\texttt{match } t \texttt{ with } \pi_1; \ldots; \pi_n$, on commence par évaluer le sujet en un constructeur appliqué à des valeurs $(C\ (v_1; \ldots; v_k))$. Si ce constructeur existe dans Π et que le filtrage est exhaustif selon un des types de Π, alors on sélectionne la clause correspondant au constructeur $\pi_i = C\ k \rightarrow t_i$. Le filtrage s'évalue en le résultat de l'évaluation de $t_i\{v_k; \ldots; v_1\}$.

Équivalence entre sémantique à réductions et sémantique naturelle par substitution

Nous avons montré dans l'assistant de preuve Coq, l'équivalence entre la sémantique à réduction et sémantique naturelle par substitution du langage εML.

Théorème 2.2.1 (Équivalence entre sémantique à réductions et naturelle)

$$\Pi \vdash t \rightarrow^* v \textit{ et } \Pi \vdash v \in \textit{Val} \textit{ si et seulement si } \Pi \vdash t \Rightarrow v.$$

La démonstration se décompose en deux lemmes :

Lemme 2.2.2 (De sémantique à réductions à sémantique naturelle)

$$\textit{Si } \Pi \vdash t \rightarrow^* v \textit{ et } \Pi \vdash v \in \textit{Val} \textit{ alors } \Pi \vdash t \Rightarrow v.$$

La preuve se fait par induction sur la séquence de réductions $\Pi \vdash t \rightarrow^* v$. Dans le cas où il y a eu 0 pas de réduction, $\Pi \vdash t \rightarrow^* t$, on a par hypothèse t est une valeur. Nous montrons qu'une valeur s'évalue en elle-même (le lemme 2.2.3) pour conclure.

Lemme 2.2.3

$$\textit{Si } \Pi \vdash v \in \textit{Val} \textit{ alors } \Pi \vdash v \Rightarrow v.$$

Dans le cas transitif, $\Pi \vdash t \rightarrow t'$ et $\Pi \vdash t' \rightarrow^* v$, l'hypothèse d'induction nous donne :

$$\Pi \vdash t' \Rightarrow v$$

Le lemme 2.2.4 nous permet de conclure.

Lemme 2.2.4

$$\textit{Si } \Pi \vdash t \rightarrow t' \textit{ et } \Pi \vdash t' \Rightarrow v \textit{ alors } \Pi \vdash t \Rightarrow v.$$

$$\frac{\Pi(\tau) = l \quad C \in l \quad \Pi \vdash \vec{t} \rightarrow^* \vec{t'}}{\Pi \vdash C(\vec{t}) \rightarrow^* C(\vec{t'})} \qquad \frac{\Pi \vdash t \rightarrow^* t'}{\Pi \vdash \texttt{let}\ t\ \texttt{in}\ t_2 \rightarrow^* \texttt{let}\ t'\ \texttt{in}\ t_2}$$

$$\frac{\Pi \vdash t \rightarrow^* t'}{\Pi \vdash t\ t_2 \rightarrow^* t'\ t_2} \qquad \frac{\Pi \vdash t \rightarrow^* t'}{\Pi \vdash t_1\ t \rightarrow^* t_1\ t'}$$

$$\frac{\tau \in \Pi \quad \texttt{exhaustive}\ (\vec{\pi})\ \tau \quad \Pi \vdash t \rightarrow^* t'}{\Pi \vdash \texttt{match}\ t\ \texttt{with}\ \vec{\pi} \rightarrow^* \texttt{match}\ t'\ \texttt{with}\ \vec{\pi}}$$

$$\frac{\Pi \vdash t \rightarrow^* t'}{\Pi \vdash (t; l) \rightarrow^* (t'; l)} \qquad \frac{\Pi \vdash t \in \mathit{Val} \quad \Pi \vdash l \rightarrow^* l'}{\Pi \vdash (t; l) \rightarrow^* (t; l')}$$

FIG. 2.2.3 – Lemmes d'injection syntaxique de la séquence de réductions

Lemme 2.2.5 (De sémantique naturelle à sémantique à réduction)

$$\Pi \vdash t \Rightarrow v \textit{ alors } \Pi \vdash t \rightarrow^* v \textit{ et } \Pi \vdash v \in \mathit{Val}.$$

La preuve se fait par induction sur la dérivation d'évaluation en sémantique naturelle. La sémantique naturelle est définie par un système de règles d'inférence dirigées par la syntaxe, tandis que la sémantique à réductions se développe pas à pas. En appliquant l'induction sur l'évaluation naturelle, nous obtenons les séquences de réductions des sous-termes. Il est alors nécessaire de définir un système dirigé par la syntaxe pour les séquences de réductions. Autrement dit, si on considère le terme $T[t]$ où t est un sous-terme dans T alors, si $\Pi \vdash t \rightarrow^* t'$ alors $\Pi \vdash T[t] \rightarrow^* T[t']$.

Nous avons défini un tel système au travers de lemmes intermédiaires que nous avons prouvés. Nous présentons ces lemmes sous la forme des règles d'inférence de la figure 2.2.3.

La preuve du lemme 2.2.5 se fait pour chaque règle de sémantique naturelle par reconstruction de la séquence de réductions en utilisant les lemmes de la figure 2.2.3 et la transitivité de la séquence de réduction.

Sémantique naturelle avec environnement

L'approche de la sémantique naturelle avec environnement peut se comprendre comme l'explicitation des substitutions [1]. Au lieu de réécrire les termes modifiés par les substitutions lors des évaluations, on place dans un environnement les associations entre variables et valeurs. Comme nous le verrons au chapitre 5, cette vision nous rapproche d'une implantation efficace d'un évaluateur. Dans le formalisme de de Bruijn, un environnement se résume en une simple liste de valeurs de telle manière que la $n$ième valeur de cette liste soit associée à la variable d'indice n. Ces valeurs ne sont pas des termes mais des valeurs sémantiques définies comme suit :

Valeurs :	$v ::= (t, e)$	fermeture simple
	$\mid (t, e)_{rec}$	fermeture récursive
	$\mid C\ (v_1; \ldots; v_n)$	valeur constructeur appliqué

$$\Pi, e \vdash \lambda.\ t \Rightarrow (t, e) \qquad \Pi, e \vdash \mu.\ t \Rightarrow (t, e)_{rec} \qquad \frac{e(n) = v}{\Pi, e \vdash n! \Rightarrow v}$$

$$\frac{\Pi, e \vdash t_1 \Rightarrow v_1 \quad \Pi, (v_1; e) \vdash t_2 \Rightarrow v}{\Pi, e \vdash \texttt{let}\ t_1\ \texttt{in}\ t_2 \Rightarrow v}$$

$$\frac{\Pi, e \vdash t_1 \Rightarrow (t, e_1) \quad \Pi, e \vdash t_2 \Rightarrow v_2 \quad \Pi, (v_2; e_1) \vdash t \Rightarrow v}{\Pi, e \vdash t_1\ t_2 \Rightarrow v}$$

$$\frac{\Pi, e \vdash t_1 \Rightarrow (t, e_1)_{rec} \quad \Pi, e \vdash t_2 \Rightarrow v_2 \quad \Pi, (v_2; (t, e_1)_{rec}; e_1) \vdash t \Rightarrow v}{\Pi, e \vdash t_1\ t_2 \Rightarrow v}$$

$$\frac{\Pi(\tau) = l \quad C \in l \quad \Pi, e \vdash t_i \Rightarrow v_i \quad (\text{pour tout } i \in [1; n])}{\Pi, e \vdash C\ (t_1; \ldots; t_n) \Rightarrow C\ (v_1; \ldots; v_n)}$$

$$\frac{\begin{array}{c}\Pi, e \vdash t \Rightarrow C\ (v_1; \ldots; v_k)) \quad \tau \in \Pi \quad \texttt{exhaustive}\ (\pi_1; \ldots; \pi_n)\ \tau \\ \pi_i = C\ k \rightarrow t_i \quad \Pi, (v_k; \ldots; v_1; e) \vdash t_i \Rightarrow v\end{array}}{\Pi, e \vdash \texttt{match}\ t\ \texttt{with}\ \pi_1; \ldots; \pi_n \Rightarrow v}$$

FIG. 2.2.4 – Sémantique naturelle de εML avec environnement

Environnement : $e = \varepsilon \mid v; e$

La valeur sémantique associée à un constructeur appliqué est formée du nom du constructeur et de la liste des valeurs de ses arguments.

Les valeurs sémantiques associées aux abstractions sont des *fermetures* [63]. Une fermeture est composée du corps de l'abstraction et d'une copie de l'environnement qui contribue à l'évaluation de l'abstraction concernée. Cette copie de l'environnement courant à l'évaluation de l'abstraction permet de fermer le corps. En effet, celui-ci peut contenir des variables libres dont les valeurs dynamiques sont présentes au moment de l'évaluation de l'abstraction.

Le jugement d'évaluation d'un terme εML en sémantique naturelle avec évaluation se note :

$$\Pi, e \vdash t \Rightarrow v$$

et signifie : " Dans l'environnement e et la liste de définitions de types concrets Π, le terme t s'évalue en la valeur v."

Le système de règles d'inférence définissant la sémantique naturelle avec environnement est, comme celui de la sémantique naturelle par substitution, dirigé par la syntaxe. Ces règles sont présentées sur la figure 2.2.4.

La variable d'indice n, $n!$ s'évalue comme le n-ième élément de e.

Les abstractions s'évaluent comme les fermetures correspondantes avec une copie de l'environnement dynamique.

Les substitutions ont été remplacées par des concaténations d'environnements. Par exemple, dans le cas de l'évaluation d'une liaison locale, `let` t_1 `in` t_2 dans l'environnement

e, si t_1 s'évalue en v_1, on n'évalue pas $t_2\{v_1\}$ mais t_2 dans l'environnement étendu $(v_1; e)$. Ainsi la variable 0! de t_2 est bien associée à la valeur v_1 dans l'environnement.

De même, lors de l'évaluation d'une application simple, la valeur d'évaluation de l'argument est placée en tête d'environnement de fermeture avant l'évaluation du corps de l'abstraction. Ainsi, les variables libres du corps des fermetures retrouvent leurs valeurs dynamiques dans l'environnement de fermeture et le paramètre formel 0! est bien associé à la valeur d'évaluation de l'argument.

Dans le cas d'une application de fermeture récursive, le corps de l'abstraction est évalué dans l'environnement de fermeture auquel on ajoute en tête, d'abord, la fermeture récursive elle-même, puis, la valeur de l'argument.

Enfin, lors de l'évaluation d'un filtrage, on n'évalue plus le corps de la clause sélectionnée dans lequel les paramètres ont été substitués par les arguments : on évalue ce corps dans l'environnement courant en tête duquel on a placé les arguments du sujets en ordre inverse afin de respecter les associations entre indices de de Bruijn et valeurs dynamiques.

Équivalence entre sémantique naturelle par substitution et avec environnement

Nous avons montré le théorème d'équivalence suivant :

Théorème 2.2.6 *Si l'environnement e correspond à la liste de termes e' alors :*

Si $\Pi, e \vdash t \Rightarrow v$ *alors il existe une valeur* v' *telle que* $\Pi \vdash t\{e'\} \Rightarrow v'$ *et* v' *correspond à* v.

De plus, $\Pi \vdash t\{e'\} \Rightarrow v'$ *alors il existe une valeur* v *telle que* $\Pi, e \vdash t \Rightarrow v$ *et* v *correspond à* v'.

Avant d'expliciter la preuve de ce théorème, nous présentons plus en détail la notion de correspondance entre environnement et liste de termes valeurs. Pour cela, nous définissons la relation de correspondance entre une valeur sémantique et un terme valeur. On note

$$\Pi \vdash t \sim v$$

l'équivalence entre la valeur terme t et la valeur sémantique v. Cette relation est définie par les trois règles suivantes :

$$\frac{\Pi(\tau) \;=\; l \qquad C \in l \qquad \Pi \vdash t_i \sim v_i \text{ pour tout } i \in [1;n]}{\Pi \vdash C(t_1; \ldots; t_n) \sim (C, v_1; \ldots; v_n)}$$

$$\frac{\Pi \vdash \vec{t} \sim \vec{v}}{\Pi \vdash \lambda.\ (t_0\ \{\Uparrow \vec{t}\}_1) \sim (t_0, \vec{v})} \qquad \frac{\Pi \vdash \vec{t} \sim \vec{v}}{\Pi \vdash \mu.\ (t_0\ \{\Uparrow_2 \vec{t}\}_2) \sim (t_0, \vec{v})_{rec}}$$

L'idée est toujours ce rapprochement entre un environnement et une substitution explicite. Il faut bien entendu passer de termes à des valeurs dynamiques. La correspondance s'étend à une liste de termes et une liste de valeurs. Bien entendu, nous avons montré que les termes issus de cette relation sont des termes valeurs.

Lemme 2.2.7 *Si* $\Pi \vdash t \sim v$ *alors* $\Pi \vdash t \in Val$.

Le théorème 2.2.6 se décompose naturellement en les deux lemmes suivants :

Lemme 2.2.8 (De sémantique naturelle avec environnement à par substitution) *Si* $\Pi, e \vdash t \Rightarrow v$ *et* $\Pi \vdash e' \sim e$ *alors il existe* v' *telle que :*

$$\Pi \vdash t\{e'\} \Rightarrow v' \text{ et } \Pi \vdash v \sim v'.$$

Lemme 2.2.9 (De sémantique naturelle par substitution à avec environnement) *Si* $\Pi \vdash t \Rightarrow v$ *tel que* $t = u\{e'\}$ *et* $\Pi \vdash e' \sim e$ *alors il existe* v' *telle que :*

$$\Pi, e \vdash u \Rightarrow v' \text{ et } \Pi \vdash v' \sim v.$$

Le lemme 2.2.8 se prouve par induction sur l'évaluation $\Pi, e \vdash t \Rightarrow v$. Le lemme 2.2.9 se prouve par induction sur l'évaluation $\Pi \vdash t \Rightarrow v$. Les deux lemmes nécessitent l'équivalent du lemme de substitution pour la substitution simultanée, que nous appelons le théorème de composition de substitutions :

Théorème 2.2.10 (Théorème de composition de substitutions)

$$t\{\vec{t}; \vec{u}\}_m = (t\{\Uparrow_{|\vec{t}|} \vec{u}\}_{m+|\vec{t}|})\{\vec{t}\}_m.$$

L'auteur vous épargne les résultats intermédiaires, notamment concernant la composition de réactualisations d'indices. Cependant, quelques résultats similaires sont présentés dans le chapitre 4 (voir la section 4.2.3).

En alliant les théorèmes 2.2.1 et 2.2.6 nous avons montré l'équivalence des sémantiques opérationnelles pour une stratégie d'évaluation faible par appel par valeur, entre les trois formes suivantes :

- à petits pas par substitution,
- à grands pas par substitution et
- à grands pas avec environnement .

Dans notre étude, nous ne cherchons pas à observer la non terminaison. En effet, nous nous intéressons aux programmes issus du mécanisme d'extraction de Coq et par conséquent bien typés et normalisant. De plus nos langages intermédiaires sont très proches, du point de vue des constructions syntaxiques (constructeur appliqué, filtrage, liaison locale, etc sont présents sous des formes plus ou moins similaires dans chacun d'entre eux). Enfin, les transformations préservent le plus souvent les structures syntaxiques. Pour toutes ces raisons nous définissons les sémantiques opérationnelles de nos langages par grands pas, le plus souvent avec environnement (dans le chapitre 4, la sémantique du langage intermédiaire CPS est donnée par substitution).

2.3 Cminor

Nous donnons ici un aperçu du premier langage intermédiaire du back-end du compilateur `Compcert` : le langage Cminor [68, 70]. Dans le cadre de notre étude, il s'agit du langage cible de notre front-end. Cminor est un langage impératif de bas niveau, inspiré du langage `C` [61] et de `C-` [93]. Cminor est doté d'un système de types très simple qui discrimine entre entiers/pointeurs et flottants. La figure 2.3.1 présente la fonction `average` qui calcule la moyenne d'un tableau d'entiers. Cette présentation nous permet d'observer les différences et similitudes entre Cminor (droite) et `C` (gauche).

```
double average(double tbl[], int sz)
{
  double s; int i;
  for (i = 0, s = 0; i < sz; i++)
    s += tbl[i];
  return s / sz;
}
```

```
"average"(tbl, sz) : int, int -> float
{
   vars s, i;  stacksize 0;
   s = 0.0; i = 0;
   block { loop {
     if (i >= sz) exit(0);
     s = s +f
          floatofint(int32[tbl + i*4]);
     i = i + 1;
   } }
   return s /f floatofint(sz);
}
```

FIG. 2.3.1 – Un exemple de fonction Cminor (droit) et la fonction `C` correspondante (gauche).

Le système de types de Cminor est plus faible que celui de `C`. En Cminor, un tableau est représenté par un pointeur. Les pointeurs Cminor sont de type `int`, car ils ont le même type mémoire. Le type des variables locales n'est pas indiqué lors de leur déclaration en Cminor. Le langage Cminor ne dispose pas des boucles de `C`, cependant les structures de contrôle de Cminor permettent de les encoder. Parmi ces structures de contrôle, on compte une boucle infinie `loop` dont le corps se répète sauf s'il est interrompu. Cminor dispose d'un système de blocs de code `block` et de sortie de blocs de code `exit`. Une sortie de bloc prend en paramètre un entier, cet entier indique le nombre de blocs englobants dont l'on veut sortir. Ici, l'encodage de la boucle `for` commence par les initialisations des variables locales `i` et `s`. Puis on met en place la boucle d'itération, pour cela on place une boucle infinie dans un bloc. Le corps de cette boucle commence par une conditionnelle qui permet de mettre en place la condition de sortie de boucle. Si la condition de sortie est vraie, on sort d'un bloc, ce qui interrompt l'exécution de la boucle infinie et le code est repris à la sortie du bloc, c'est-à-dire, dans notre exemple, après l'encodage de la boucle `for`. L'incrémentation est ajoutée dans le corps de la boucle. Les opérateurs arithmétiques de Cminor ne sont pas surchargés : l'opérateur d'addition `+` de deux entiers est distinct de l'opérateur d'addition $+_f$ de deux flottants. De plus on ne peut additionner à un flottant un entier. Ainsi `s+=tbl[i]` ajoute au flottant `s` l'entier `tbl[i]` et l'affecte à ce flottant. En Cminor, l'opération devient une addition de flottants, l'entier est converti en flottant.

Les fonctions Cminor disposent d'une structure de stockage de données dans la pile d'appels. Il s'agit d'un bloc mémoire particulier, directement manipulable dans la syntaxe. Il se manipule par son adresse, comme un pointeur. A chaque appel de fonction, un nouveau bloc de pile est alloué. Chaque fonction indique dans sa définition la taille du bloc de pile dont elle a besoin.

Sur la figure 2.3.2, se trouve un exemple de l'utilisation de la pile Cminor. La fonction à gauche est la fonction `C`, et à droite la fonction Cminor correspondante. Il s'agit d'un test de la fonction `average` de la figure 2.3.1. La fonction déclare un tableau local de trois entiers `(1,2,3)` et retourne le résultat `average` sur ce tableau.

Une possibilité de traduction en Cminor est de placer le tableau local sur la pile courante de la fonction test. Pour cela, la fonction commence par indiquer la taille de pile nécessaire

```
double test(void)
{
  int t[3] = {1,2,3};
  return average(t,3)
}
```

```
"test"() : -> float
{
   stacksize 12;
  int32[addrstack(0)]=1;
 int32[addrstack(1)]=2;
  int32[addrstack(2)]=3;
  return "average"(addrstack(0),3);
}
```

FIG. 2.3.2 – Un exemple d'utilisation de la pile Cminor (droit) et la fonction `C` correspondante (gauche).

à l'exécution de son corps. Ici, la pile doit pouvoir stocker 3 entiers, soit 12 octets mémoire. L'initialisation du tableau se fait alors par écriture dans les emplacements concernés. Par exemple, pour le premier élément du tableau on écrit au premier emplacement sur la pile `addrstack(0)` l'entier de 32 bits 1. Enfin, on appelle la fonction `average` avec l'adresse de la pile.

Nous nous intéressons ici au sous-ensemble de Cminor significatif pour la compilation de εML.

2.3.1 Syntaxe abstraite

Cminor est organisé de manière classique en quatre niveaux : expressions, instructions, fonctions globales et programmes.

Expressions

Les expressions de Cminor sont *pures* : elles ne produisent pas d'effets de bord et ne modifient pas la mémoire. La grammaire des expressions Cminor est la suivante :

Expressions :

a	$::= id$	lecture d'une variable locale
	$\mid cst$	constante
	$\mid \texttt{add}(a_1, a_2)$	addition entre entiers ou entier + pointeurs
	$\mid \texttt{mult}(a_1, a_2)$	multiplication entre entiers
	$\mid \texttt{int32}[a]$	lecture en mémoire du mot à l'adresse a

Constantes :

cst	$::= n$	constante entière
	$\mid \texttt{addrsymbol}(id)$	adresse d'un symbole global
	$\mid \texttt{addrstack}(n)$	adresse d'un emplacement de pile

Le sous-ensemble des expressions de Cminor que nous présentons ici se caractérise par la consultation explicite de la mémoire. Il est possible de récupérer l'adresse mémoire d'un

symbole id grâce à la constante `addrsymbol`(id). Nous nous en servirons pour récupérer l'adresse de pointeur de code des fonctions. On peut consulter explicitement le contenu du bloc de pile situé à l'emplacement n par la constante `addrstack`(n). Il est aussi possible de consulter le contenu de la mémoire à une certaine adresse : `int32`[a], lit un entier de 32 bits ou un pointeur à l'adresse a. Notons qu'il est possible de lire d'autres quantités de mémoire, cependant, nous ne travaillerons qu'avec cette quantité. Pour ces raisons, nous noterons la lecture d'un entier de 32 bits ou d'un pointeur à l'adresse a simplement : [a].

En plus de la consultation explicite de la mémoire, nous disposons de variables locales. Elles servent à stocker des valeurs dans un environnement local qui n'ont pas besoin d'être en mémoire. Enfin, nous avons aussi des constantes entières et parmi les nombreux opérateurs de Cminor (pratiquement tout ceux de `C`) nous n'utiliserons que l'addition entre entiers ou entre un pointeur et un entier ainsi que la multiplication entre entiers.

Instructions (*statements*)

Les instructions Cminor sont aussi très proches des instructions de `C`. Néanmoins, elles sont de plus bas niveau : par exemple, les écritures en mémoire indiquent la taille du mot à écrire. Ici encore, nous nous sommes concentrés sur le sous-ensemble de Cminor utilisé dans la génération de code Cminor.

Instructions :

s	::= `skip`	pas d'opération
	$\mid id \leftarrow a$	affectation d'une variable locale
	$\mid$ `int32`$[a_1] = a_2$	écriture en mémoire à l'adresse a_1
	$\mid a(a_1, \ldots, a_n) : sig$	appel de procédure
	$\mid id \leftarrow a(a_1, \ldots, a_n) : sig$	appel de fonction
	$\mid$ `tailcall` $a(a_1, \ldots, a_n) : sig$	appel terminal de fonction
	$\mid s_1; s_2$	séquence
	$\mid$ `block`$\{s_1\}$	bloc pour délimiter les `exit`
	$\mid$ `exit`(k)	sort du $(k+1)^{\text{ième}}$ bloc englobant
	$\mid$ `switch`$(a)\{\ldots$ `case` n_i : `exit`$(k_i)\ldots$ `default` : `exit`$(k)\}$	discrimination sur la valeur de a
	$\mid$ `return`	retour de procédure
	$\mid$ `return`(a)	retour de fonction avec le résultat a

Signatures de fonctions :

sig	::= $\tau_1, \ldots, \tau_n \rightarrow \tau$	signature de fonction
	$\mid \tau_1, \ldots, \tau_n \rightarrow$ `void`	signature de procédure

Type de valeur :

τ	::= `int`	nombre entier ou pointeur
	$\mid$ `float`	nombre flottant

Il y a deux sortes d'appels de fonction en Cminor. Les appels normaux continuent dans la fonction courante au retour de la fonction appelée. Les appels terminaux (`tailcall`) retournent immédiatement le résultat de la fonction appelée comme résultat de la fonction courante. Nous retrouverons cette notion d'appel terminal dans le chapitre 7, section 7.3.4.

Fonctions

Les fonctions de Cminor sont globales. Cela signifie que le corps d'une fonction est clos et que par conséquent, il n'y a pas d'ordre de visibilité entre l'ensemble des fonctions d'un programme : elles peuvent toutes s'utiliser entre elles.

Une fonction est composée d'un corps (`body`) qui se trouve être une instruction. La définition d'une fonction est munie d'une liste de variables locales `vars` et de la liste des paramètres `params`. Les fonctions Cminor peuvent avoir plusieurs arguments. La définition d'une fonction indique aussi la taille du bloc de pile qu'elle utilisera (`stacksize`), afin d'allouer un bloc de pile de taille suffisante. Enfin, cette définition contient une signature (`sig`) indiquant le type de retour et le type des paramètres formels de la fonction.

Fonction :

$fn ::= \{$	$\texttt{sig} = sig;$	signature de la fonction
	$\texttt{params} = \vec{id}$	paramètres de la fonction
	$\texttt{vars} = \vec{id}$	variables locales de la fonction
	$\texttt{stacksize} = n$	taille du bloc de pile
	$\texttt{body} = s\ \}$	corps de la fonction

Programmes

Pour nous, un programme Cminor est une liste de fonctions `functs` accompagné du nom de la fonction de point d'entrée `main`.

Programmes :

$prog ::= \{$	$\texttt{functs} = \ldots\ id = fd\ \ldots$	les fonctions du programme
	$\texttt{main} = id\ \}$	le point d'entrée du programme

2.3.2 Sémantique opérationnelle

Nous décrivons ici la sémantique opérationnelle du langage Cminor. La sémantique de Cminor est une sémantique à grands pas, avec environnement. Les sémantiques avec environnement requièrent la définition de valeurs sémantiques.

Valeurs et environnements

Une valeur sémantique est un objet formel décrivant la valeur en laquelle un calcul du langage s'évalue selon sa sémantique formelle. Les valeurs associées au calcul en Cminor sont les valeurs correspondant au système de types de Cminor : entiers, pointeurs et flottants. De plus, nous considérons la valeur spéciale indéfinie `undef`. Cette valeur nous permet de refléter tous les calculs dont le comportement est indéfini. On note b l'identifiant d'un bloc mémoire (adresse) et M un état mémoire.

Valeurs :

v ::=	$\texttt{int}(n)$	entier
|	$\texttt{ptr}(b, \delta)$	pointeur sur le δième octet du bloc mémoire b
|	`Vundef`	valeur indéfinie

Environnements locaux :

$E ::= id \to v$

Environnements globaux :

$G ::= (id \to b) \times (b \to fd)$

Les environnements globaux associent des blocs mémoire aux identificateurs des variables globales et des fonctions, et des définitions de fonction aux blocs mémoire représentant des fonctions. Les opérations sur les environnements globaux sont :

$\texttt{find\_symbol}(G, id) = b$
: Renvoie le bloc mémoire b associé à l'identificateur global id dans l'environnement global G.

$\texttt{find\_funct}(G, b) = fd$
: Renvoie la définition de fonction fd stockée dans le bloc mémoire b dans l'environnement globale G.

$\texttt{init\_genv}(prog) = (G, M)$
: Construit l'environnement global G et l'état mémoire initial M correspondant aux déclarations de fonctions et de variables globales du programme $prog$.

États mémoire

Une description plus formelle et complète du modèle mémoire de Cminor est présentée dans [71]. Nous nous contentons ici de décrire les fonctions de ce modèle mémoire utiles à la description de la sémantique du sous-ensemble de Cminor qui nous est utile.

Les opérations sur l'état mémoire sont les suivantes :

$\texttt{alloc}(M, n) = (b, M')$
: Alloue un nouveau bloc de taille n. b est l'adresse de ce bloc, M' la nouvelle mémoire.

$\texttt{load}(M, \kappa, b, \delta) = v$
: Lit une quantité de mémoire κ à l'offset δ dans le bloc b de la mémoire M. Renvoie la valeur v correspondante. Nous noterons $M(b, \delta)$ la lecture dans la configuration M à l'offset δ du bloc b de la quantité `int32` (mots de 32 octets).

$\texttt{store}(M, \kappa, b, \delta, v) = M'$
: Stocke la valeur v suivant la quantité de mémoire κ à l'offset δ dans le bloc b de la mémoire M. Renvoie la nouvelle mémoire M'.

$\texttt{free}(M, b) = M'$
: Désalloue le bloc b. M' est la nouvelle mémoire après désallocation.

Évaluation des expressions

Nous rappelons que les expressions de Cminor sont pures, elles ne produisent pas d'effet de bord. Autrement dit, l'évaluation d'une expression Cminor ne modifie pas les environ-

nements d'évaluation. Le jugement d'évaluation d'une expression Cminor s'énonce comme suit :

$$G, sp, E, M \vdash a \Rightarrow v.$$

Il se lit : "Dans l'environnement global G, le pointeur de pile sp, l'environnement local E et l'état mémoire M, l'expression a s'évalue en la valeur v."

$$\frac{E(id) = v}{G, sp, E, M \vdash id \Rightarrow v} \qquad \frac{\texttt{eval\_constant}(G, sp, cst) = v}{G, sp, E, M \vdash cst \Rightarrow v}$$

$$\frac{G, sp, E, M \vdash a_1 \Rightarrow v_1 \quad G, sp, E, M \vdash a_2 \Rightarrow v_2 \quad v_1 + v_2 = v}{G, sp, E, M \vdash \texttt{add}(a_1, a_2) \Rightarrow v}$$

$$\frac{G, sp, E, M \vdash a_1 \Rightarrow \texttt{ptr}(b, \delta) \quad \texttt{load}(M, \texttt{int32}, b, \delta) = v}{G, sp, E, M \vdash [a_1] \Rightarrow v}$$

Calcul de la valeur d'une constante :

$$\begin{aligned} \texttt{eval\_constant}(G, sp, n) &= \texttt{int}(n) \\ \texttt{eval\_constant}(G, sp, \texttt{addrsymbol}(id)) &= \texttt{ptr}(b, 0) \text{ si } \texttt{find\_symbol}(G, id) = b \\ \texttt{eval\_constant}(G, sp, \texttt{addrstack}(n)) &= \texttt{ptr}(sp, n) \end{aligned}$$

Le jugement de l'évaluation d'une liste d'expressions s'énonce comme suit :

$$G, sp, E, M \vdash \vec{a} \Rightarrow \vec{v}$$

Il se lit : "Dans l'environnement global G, le pointeur de pile sp, l'environnement local E et l'état mémoire M, la liste d'expressions $\vec{a}$ s'évalue en la liste de valeurs $\vec{v}$."

$$G, sp, E, M \vdash \varepsilon \Rightarrow \varepsilon \qquad \frac{G, sp, E, M \vdash a \Rightarrow v \quad G, sp, E, M \vdash \vec{a} \Rightarrow \vec{v}}{G, sp, E, M \vdash (a; \vec{a}) \Rightarrow (v; \vec{v})}$$

Évaluation des instructions

Contrairement aux expressions, les évaluations des instructions Cminor produisent des efforts de bord. Ces évaluations modifient l'état (environnement local et configuration mémoire). De plus, les structures de contrôle guident l'exécution d'une instruction. Afin de caractériser un déroulement d'exécution, nous munissons la sémantique formelle d'informations caractérisant le mode de fin d'exécution d'une exécution. La fin d'exécution d'une instruction produit un *outcome*, qui peut être :

out ::=	`Out_continue`	continuer en séquence
	| `Out_exit`(k)	sortir du $(k+1)^{\text{ième}}$ bloc englobant
	| `Out_return`	quitter la fonction
	| `Out_return`(v)	quitter la fonction avec la valeur v
	| `Out_normal`	quitter normalement
	| `Out_tailcall_return`(v)	quitter la fonction avec la valeur v, sachant que son bloc de pile a déjà été libéré

Le jugement d'exécution d'une instruction se définit comme suit :

$$G, sp \vdash s, E, M \Rightarrow out, E', M'$$

Il se lit : "dans l'environnement global G, le pointeur de pile sp, l'environnement local E et l'état mémoire M, l'instruction s s'évalue en le résultat out. L'environnement local à la fin de l'évaluation est E' et l'état mémoire à la fin de l'évaluation est M'." La figure 2.3.3 présente les règles de sémantique des instructions de Cminor.

Il y a deux sortes d'appels de fonction en Cminor, en position terminale ou pas. La sémantique de Cminor optimise les appels en position terminale. L'exécution d'un appel normal alloue un bloc de pile propre à son corps de fonction et ne désalloue pas le précédent (bloc de pile de la fonction appelante). Dans le cas d'un appel en position terminale, on sait que la fonction appelée retourne un résultat que la fonction appelante retourne immédiatement. De ce fait, le bloc de pile de la fonction appelante devient superflu et est donc libéré avant l'appel.

Appel de fonctions

$$G \vdash fd(\vec{v}), M \Rightarrow v, M'$$

Dans l'environnement global G et l'état mémoire initial M, la fonction fd appliquée aux arguments $\vec{v}$ renvoie la valeur v provenant d'un retour d'appel de fonction avec pour outcome `Out_return`(v) ou bien `Out_tailcall_return`(v) que nous notons $out(v)$. L'état mémoire à la fin de l'évaluation est M'. Seul ce cas intéresse la compilation de εML en Cminor.

$$\frac{\begin{array}{c} alloc(M, fn.\texttt{stackspace}) = (sp, M_1) \\ fn.\texttt{params} = (p_1, \ldots, p_n) \qquad fn.\texttt{vars} = (q_1, \ldots, q_m) \\ E_1 = [p_1 \leftarrow v_1, \ldots, p_n \leftarrow v_n, q_1 \leftarrow \texttt{Vundef}, \ldots, q_m \leftarrow \texttt{Vundef}] \\ G, sp \vdash fn.\texttt{body}, E_1, M_1 \Rightarrow out(v), E_2, M_2 \end{array}}{G \vdash (\texttt{internal}\ fn)(v_1, \ldots, v_n), M \Rightarrow v, M_2}$$

Évaluation d'un programme complet

Enfin, l'exécution d'un programme Cminor :

$$\vdash prog \Rightarrow n.$$

$$G, sp \vdash \texttt{skip}, E, M \Rightarrow \texttt{Out\_normal}, E, M$$

$$\frac{G, sp, E, M \vdash a \Rightarrow v}{G, sp \vdash (id = a), E, M \Rightarrow \texttt{Out\_normal}, E\{id \leftarrow v\}, M}$$

$$\frac{G, sp, E, M \vdash a_1 \Rightarrow \texttt{ptr}(b, \delta) \quad G, sp, E, M \vdash a_2 \Rightarrow v \quad \texttt{store}(M, \texttt{int32}, b, \delta, v) = M'}{G, sp \vdash ([a_1] = a_2), E, M \Rightarrow \texttt{Out\_normal}, E, M'}$$

$$\frac{G, sp, E, M \vdash a \Rightarrow \texttt{ptr}(b, 0) \quad G, sp, E, M \vdash \vec{a} \Rightarrow \vec{v} \quad \texttt{find\_funct}(G, b) = fd \quad G \vdash fd(\vec{v}), M \Rightarrow v, M'}{G, sp \vdash (id = a(\vec{a})), E, M \Rightarrow \texttt{Out\_normal}, E\{id \leftarrow v\}, M'}$$

$$\frac{\begin{array}{c} G, sp, E, M \vdash a \Rightarrow \texttt{ptr}(b, 0) \quad G, sp, E, M \vdash \vec{a} \Rightarrow \vec{v} \\ \texttt{find\_funct}(G, b) = fd \quad G \vdash fd(\vec{v}), \texttt{free}(M, sp) \Rightarrow v, M' \end{array}}{G, sp \vdash (id = \texttt{tailcall}\ a(\vec{a})), E, M \Rightarrow \texttt{Out\_normal}, E\{id \leftarrow v\}, M'}$$

$$\frac{G, sp \vdash s_1, E, M \Rightarrow \texttt{Out\_normal}, E_1, M_1 \quad G, sp \vdash s_2, E_1, M_1 \Rightarrow out, E', M'}{G, sp \vdash (s_1; s_2), E, M \Rightarrow out, E', M'}$$

$$\frac{G, sp \vdash s_1, E, M \Rightarrow out, E', M' \quad out \neq \texttt{Out\_normal}}{G, sp \vdash (s_1; s_2), E, M \Rightarrow out, E', M'}$$

$$G, sp \vdash \texttt{exit}(k), E, M \Rightarrow \texttt{Out\_exit}(k), E, M$$

$$\frac{G, sp, E, M \vdash a \Rightarrow \texttt{int}(n) \quad k = \begin{cases} k_i & \text{si } n = n_i \\ k_{dfl} & \text{sinon} \end{cases}}{G, sp \vdash (\texttt{switch}(a)\{\ldots \texttt{case}\ n_i : \texttt{exit}(k_i) \ldots \texttt{default} : \texttt{exit}(k_{dfl})), E, M \Rightarrow \texttt{Out\_exit}(k), E, M}$$

$$G, sp \vdash \texttt{return}, E, M \Rightarrow \texttt{Out\_return}, E, M$$

$$\frac{G, sp, E, M \vdash a \Rightarrow v}{G, sp \vdash \texttt{return}(a), E, M \Rightarrow \texttt{Out\_return}(v), E, M}$$

FIG. 2.3.3 – Sémantique des instructions Cminor

Le programme *prog* s'évalue en le code de retour n.

$$\frac{\texttt{init\_genv}(prog) = (G, M) \quad \texttt{find\_symbol}(G, prog.\texttt{main}) = b \quad \texttt{find\_funct}(G, b) = fd \quad G \vdash fd(\varepsilon), M \Rightarrow \texttt{int}(n), M'}{\vdash prog \Rightarrow n}$$

2.4 De la preuve de préservation sémantique

Nous essayons, ici, de définir le concept de nos formalisations et preuves de préservation sémantique. Les définitions et relations sont abstraites.

2.4.1 Préservation sémantique

Une transformation ou compilation d'un langage source L_s vers un langage cible L_c est un algorithme $\mathcal{C}$ qui transforme un programme p du langage L_s en un programme p' (où $p' = \mathcal{C}(p)$) du langage L_c de telle manière que le comportement de p' soit celui de p. Le comportement d'un programme est donné par sa sémantique dynamique. La correction d'un compilateur consiste à vérifier que pour tout programme source pouvant être compilé, son comportement est préservé par le programme cible généré. On parle alors de *préservation sémantique*. Nous rappelons que, nous souciant du langage des programmes issus du mécanisme d'extraction, il n'est pas nécessaire d'observer la non terminaison. La comparaison de deux évaluations de langages différents se fait en comparant les valeurs observables d'évaluation de programme. Il s'agit de valeurs constantes ou de premier ordre. Dans le langage εML et ses variantes, les valeurs de premier ordre sont soit un constructeur constant (c'est-à-dire d'arité nulle) ou bien un constructeur appliqué à des constructeurs. On note v_1 les valeurs de premier ordre de εML et ses variantes que l'on définit comme suit :

$$v_1 \ ::= \ C(\vec{v_1}).$$

Considérons la compilation du langage L_s vers le langage L_c (dans notre chaîne de compilation deux langages différents de Cminor), le théorème de préservation sémantique s'exprime comme suit :

Théorème 2.4.1 (Préservation sémantique pour un programme)
Considérons le programme p du langage L_s, si

$$p \Rightarrow v_1 \text{ où } v_1 \text{ est une valeur de premier ordre,}$$

alors

$$\mathcal{C}(p) \Rightarrow v_1.$$

Généralement, un compilateur réalise une séquence de transformations de langages intermédiaires. Par exemple,

$$L_s \rightarrow^{\mathcal{C}_0} L_0 \rightarrow^{\mathcal{C}_1} L_1 \ldots \rightarrow^{\mathcal{C}_n} L_c.$$

Le compilateur $\mathcal{C}$ est alors la composée $\mathcal{C}_n \circ \ldots \circ \mathcal{C}_0$ des transformations successives.

La preuve du théorème 2.4.1, se compose alors des théorèmes de préservation sémantique de chaque transformation de la chaîne de compilation.

2.4.2 Équivalence observationnelle

En général, les valeurs produites par l'évaluation des termes constituant un programme ne sont pas des constantes. En particulier, les valeurs fermetures portent le corps de l'abstraction. Bien évidemment ce corps est un terme du langage et donc n'est pas directement comparable avec une fermeture d'un autre langage intermédiaire, aussi proche syntaxiquement que possible.

De même, lors de l'évaluation de termes constituant un programme, on consulte l'ensemble des environnements d'évaluation. En plus de contenir des valeurs non directement comparables, l'organisation, le type d'information des environnements d'évaluation de deux langages intermédiaires ne sont pas directement comparables. Ces considérations induisent la nécessité de définir une *équivalence observationnelle* afin de pouvoir comparer les évaluations lors d'une preuve de préservation sémantique. Ainsi, pour chaque transformation, nous avons défini une équivalence observationnelle entre les valeurs et environnements des sémantiques du langage source et du langage cible. La préservation sémantique d'une transformation, notée $[\![]\!]$, pour un sous-terme de programme, pour une notion d'équivalence observationnelle donnée, s'énonce par le théorème suivant :

Théorème 2.4.2 (Préservation sémantique pour un sous-terme de programme)
Considérons l'évaluation du terme t du langage L_i dans l'environnement d'évaluation $\mathbb{E}_i$ en la valeur v,

$$\mathbb{E}_i \vdash t \Rightarrow v,$$

si l'environnement $\mathbb{E}_i$ est observationnellement équivalent à l'environnement d'évaluation $\mathbb{E}_{i+1}$ du langage L_{i+1} ,

$$\mathbb{E}_i \sim \mathbb{E}_{i+1}$$

alors il existe une valeur v' du langage L_{i+1} telle que :

$$\mathbb{E}_{i+1} \vdash [\![t]\!] \Rightarrow v' \text{ où } v' \text{ est observationnellement équivalente à } v \ .$$

2.4.3 Lemme de simulation

La reconstitution du cadre d'observation lors de la preuve de préservation sémantique d'un sous-terme, incluse dans la preuve de préservation sémantique d'un terme peut nécessiter la maintenance d'invariants.

Enfin la préservation sémantique pour un sous-terme de programme se décrit au travers de lemmes de simulations. Ces lemmes de simulation peuvent s'exprimer au travers de diagrammes de simulation tels que celui décrit par la figure 2.4.1 ; où t est un terme du langage L_i et $[\![t]\!]$ est le terme du langage L_{i+1} issu de la traduction du terme t. Ce diagramme se lit comme suit :

Lemme 2.4.3 (Lemme de simulation)
Considérons l'évaluation du terme t dans l'environnement d'évaluation $\mathbb{E}_i$:

$$\mathbb{E}_i \vdash t \Rightarrow v,$$

si les invariants H_{inv} sont vérifiés, alors il existe une valeur v' telle que :

$$\mathbb{E}_{i+1} \vdash [\![t]\!] \Rightarrow v' \text{ et les conclusions } \mathsf{C}_{inv} \text{ sont vérifiées.}$$

Bien entendu, le cadre d'équivalence entre les environnements d'évaluation $\mathbb{E}_i$ et $\mathbb{E}_{i+1}$ est inclus dans H_{inv} et l'équivalence observationnelle entre les valeurs obtenues est incluse dans C_{inv}.

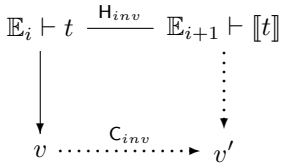


FIG. 2.4.1 – Diagramme de simulation

De tels lemmes de simulation se démontrent par induction sur le jugement d'évaluation du terme source. Les preuves sont donc constituées d'autant de cas qu'il y a de règles d'inférence définissant la sémantique du langage source.

Pour chaque transformation, nous avons spécifié de tels cadres d'observation et identifié les invariants nécessaires à leur maintien dans l'assistant de preuve Coq. Cela nous a permis de mener, dans cet assistant, les preuves des lemmes de simulation pour chacune de nos transformations ; et donc de prouver de manière formelle, la préservation sémantique de chacune d'elle.

Enfin, la composition des théorèmes de préservation sémantique de toutes les transformations de notre chaîne de compilation forme la vérification formelle de notre front-end pour miniML vers Cminor.

Composant cette preuve avec celle du back-end CompCert, nous obtenons un compilateur pour miniML, formellement vérifié dans l'assistant de preuve Coq.

2.4.4 Une illustration : la préservation sémantique de la numérotation

La numérotation des constructeurs est la première de nos transformations. Parmi les structures de données de εML, on trouve les types concrets. La définition d'un type concret répertorie la liste de ses constructeurs : $\tau := C(x_0), \ldots, C(x_n)$, où les x_i sont les noms identifiants les constructeurs. Par exemple,

`arbre` := `C(Feuille)` | `C(Bourgeon)` | `C(Noeud)` .

L'utilité des types concrets s'illustre à travers des filtrages. Un filtrage sur un type concret donné τ permet de sélectionner la suite du calcul selon un constructeur de τ. Ainsi,

$$\begin{array}{l}\texttt{match } t \texttt{ with}\\ \mid C(x_0)\ (y_0, \ldots, y_m) \rightarrow t_0\\ \mid \ldots\\ \mid C(x_n)\ (z_0, \ldots, z_r) \rightarrow t_n\end{array}$$

sélectionne selon la valeur de t, la clause qui sera la suite du calcul. Si t s'évalue en le constructeur $C(x_i)\ (v_0, \ldots, v_j)$ où $i \in [0, n]$ alors la clause $C(x_i)\ (a_0, \ldots, a_j) \rightarrow t_i$ est

sélectionnée et on évalue alors t_i où les paramètres du motif ont été instanciés par les v_i.

Il existe une autre manière de représenter les constructeurs. Un constructeur peut être représenté par sa position dans la définition de son type concret. Dans le code `Cminor` produit par le compilateur, nous souhaitons représenter les constructeurs par des petits entiers et non pas par leurs noms. Les constructeurs $C(0), \ldots, C(n)$ d'un même type concret τ seront représentés par les entiers machine $0, 1, \ldots, n$. Cette représentation est celle qui se prête le mieux à la production de code efficace pour le filtrage, notamment à l'utilisation de tables de sauts ou de recherche dichotomique pour discriminer sur un constructeur. Remarquons que dans cette représentation deux constructeurs de deux types concrets différents peuvent être représentés par le même entier. Ceci n'entraîne pas de risque de confusion entre constructeurs pendant un filtrage, car le bon typage du programme (statique ou dynamique) nous garantit que tous les constructeurs apparaissant dans un même filtrage appartiennent à un même type, et donc ont des numéros distincts. Ici, le bon typage dynamique sera obtenu par transformation. Nous transformons des programme εML, où les constructeurs sont nommés, en programme εML#, variante de εML où les constructeurs sont numérotés.

Dans cette section nous notons les constructeurs de εML $C(x)$ et et ceux de εML# $C(n)$ afin de distinguer les constructeurs nommés des constructeurs numérotés.

εML#

εML# est une variante de εML où les constructeurs sont numérotés. C'est le premier langage intermédiaire de notre chaîne de compilation.

Syntaxe La grammaire suivante définit εML# dans le formalisme de de Bruijn.

Termes : $t ::= n!$
$\quad\quad | \ \lambda.\ t$
$\quad\quad | \ t_1\ t_2$
$\quad\quad | \ \texttt{let}\ t_1\ \texttt{in}\ t_2$
$\quad\quad | \ \texttt{letrec}\ t_1\ \texttt{in}\ t_2$
$\quad\quad | \ C(m)\ (t_1; \ldots; t_n)$ constructeur numéroté
$\quad\quad | \ \texttt{match}\ t_1\ \texttt{with}\ \pi_1; \ldots; \pi_n$

Motif : $\pi ::= n \to t$ n est l'arité du filtré

Un programme εML# est un terme, il n'y a pas de liste de définitions de types concrets.

Un constructeur appliqué est représenté par son numéro m et sa liste d'arguments $(t_1; \ldots; t_n) : C(m)\ (t_1; \ldots; t_n)$.

Les clauses de filtrage ne portent plus de nom de constructeur. Une clause en $i$ème position d'un filtrage informe, dans son motif, de l'arité du $i$ème constructeur du type concerné par le filtrage. $m \to t$, est une clause de m paramètres formels. Dans son corps ces paramètres sont les variables d'indices $0!, \ldots, (m-1)!$, comme dans εML.

Le programme suivant est l'exemple exposé à la section 2.1 de la définition de la fonction *plus* sur les entiers de Peano :

```
{
  letrec f x =
     \ y. match x with
        | 0  -> y
        | (1) z -> (C(1)) (f z y)
   in
    let z = (C(1) (C(1) C(0))) in
        (\ x. f x (C(1) C(0))) z }
```

Nous avons nommé les variables pour plus de clarté dans l'exemple. Les nombres apparaissant dans les motifs du filtrage sont les arités des constructeurs. Pour profiter de la numérotation des constructeurs, les clauses de filtrage sont réordonnées selon le numéro du constructeur de leur motif. Comme ces constructeurs sont toujours dans cet ordre, le $i$ème motif est lié au $i$ème constructeur du type sur lequel on veut discriminer.

Sémantique naturelle avec environnement La sémantique de εML# diffère de la précédente au niveau de l'évaluation du filtrage et des constructeurs. L'absence de définition de type concret allège le jugement d'évaluation :

$$e \vdash t \Rightarrow v.$$

La valeur sémantique d'un constructeur ne porte plus d'information nominale mais le numéro lui correspondant.

Valeurs : $v ::= (t, e) \mid (t, e)_{rec} \mid (i, v_1; \ldots; v_n)$
Environnement : $e ::= \varepsilon \mid v; e$

$$e \vdash \lambda.\ t \Rightarrow (t, e) \qquad \frac{e(n) = v}{e \vdash n! \Rightarrow v} \qquad \frac{e \vdash t_1 \Rightarrow v_1 \quad v_1; e \vdash t_2 \Rightarrow v}{e \vdash \texttt{let}\ t_1\ \texttt{in}\ t_2 \Rightarrow v}$$

$$\frac{(t_1, e)_{rec}; e \vdash t_2 \Rightarrow v}{e \vdash \texttt{letrec}\ t_1\ \texttt{in}\ t_2 \Rightarrow v} \qquad \frac{e \vdash t_1 \Rightarrow (t, e_1) \quad e \vdash t_2 \Rightarrow v_2 \quad v_2; e_1 \vdash t \Rightarrow v}{e \vdash t_1\ t_2 \Rightarrow v}$$

$$\frac{e \vdash t_1 \Rightarrow (t, e_1)_{rec} \quad e \vdash t_2 \Rightarrow v_2 \quad v_2; (t, e_1)_{rec}; e_1 \vdash t \Rightarrow v}{e \vdash t_1\ t_2 \Rightarrow v}$$

$$\frac{e \vdash t_i \Rightarrow v_i \quad (\text{pour tout } i \in [1; n])}{e \vdash C(m)\ (t_1; \ldots; t_n) \Rightarrow (m, v_1; \ldots; v_n)}$$

$$\frac{e \vdash t \Rightarrow (i, (v_1; \ldots; v_k)) \quad \pi_i\ =\ k \rightarrow t_i \quad v_{(k-1)}; \ldots; v_0; e \vdash t_i \Rightarrow v}{e \vdash \texttt{match}\ t\ \texttt{with}\ \pi_1; \ldots; \pi_n \Rightarrow v}$$

Le constructeur appliqué $C(m)\ (v_1; \ldots; v_n)$ dans l'environnement e s'évalue en $(m, v_1; \ldots; v_n)$ si, dans e, $t_1; \ldots; t_n$ s'évalue en $v_1; \ldots; v_n$.
Le filtrage `match` t `with` $\pi_1; \ldots; \pi_n$ s'évalue dans e en v, si t dans e s'évalue en $(i, v_1; \ldots; v_k)$, et t_i, le corps de la $i$ème clause, dans $v_{(k-1)}; \ldots; v_0; e$ s'évalue en v.

Numérotation des constructeurs : des noms aux numéros

Cette transformation se fait au travers d'une numérotation Γ, telle que $\Gamma(x) = n$ si le constructeur εML $C(x)$ du type τ apparaît à la n-ième position dans la définition de τ, c'est-à-dire si $\tau(n) = x$.

La numérotation est construite au début de la transformation d'un programme p à partir de sa liste de définitions $p.\texttt{type}$.

Cette transformation réordonne aussi les clauses de filtrage. En effet, en εML les constructeurs n'apparaissent pas obligatoirement dans un filtrage dans l'ordre de la définition du type associé. En εML#, les motifs de filtrage sont ordonnés suivant la position des constructeurs dans la définition du type.

Dans les faits, la traduction peut échouer : un constructeur peut ne pas avoir d'image par la numérotation. Dans ce cas, la traduction échoue. Nous utilisons la monade d'erreur en Coq pour gérer le cas d'échec et sa propagation sur toute la transformation d'un programme. Avant de décrire la transformation dans la présentation monadique utilisée dans notre développement, nous décrivons l'algorithme.

Présentation de l'algorithme La numérotation est un paramètre global de la numérotation du terme d'un programme : elle reste inchangée au cours de cette numérotation. La numérotation d'un programme p dans son intégralité commence donc par la construction de la numérotation qui guidera la numérotation de son corps $p.\texttt{terme}$.

La numérotation associe à chaque nom de constructeur apparaissant dans $p.\texttt{type}$ le numéro correspondant à sa position dans la déclaration de son type. Pour chaque $\tau \in p.\texttt{type}$ où $\tau = C(x_0), \ldots, C(x_n)$, on associe à $C(x_i)$ pour chaque $i \in [0, n]$ le numéro i. En appliquant ce mécanisme à chaque $\tau \in p.\texttt{type}$, la fonction `make_num`, qui prend la liste de déclarations de types du programme, construit la numérotation dans laquelle $p.\texttt{terme}$ sera numéroté. Soit γ la numérotation `make_num` $p.\texttt{type}$. La construction de la numérotation effectue, lors de son parcours des constructeurs, une vérification de l'unicité de chaque constructeur de p.

La numérotation d'un terme t est définie récursivement sur la structure de t en considérant la numérotation Γ construite au niveau du programme. Nous notons $[\![t]\!]_\Gamma$ la numérotation du terme t par rapport à la numérotation Γ. La transformation d'un programme est :

$$\begin{aligned} [\![p]\!] \quad = \quad & \texttt{let } \Gamma = \texttt{make\_num } p.\texttt{type } \texttt{in} \\ & [\![p.\texttt{terme}]\!]_\Gamma \end{aligned}$$

Les cas intéressants de la numérotation sont les suivants :

Transformation d'un constructeur appliqué :
$C(x)\ (t_1; \ldots; t_n)$ se traduit en $C(\Gamma(x))\ ([\![t_1; \ldots; t_n]\!]_\Gamma)$ où $\Gamma(x)$ est le numéro d'apparition de $C(x)$ dans la définition du type correspondant.

Transformation d'une clause de filtrage :
Lors de la numérotation de la clause $C(x)\ n \to t$, nous récupérons le numéro associé à c dans Γ, $\Gamma(x)$ en plus de la clause traduite $n \to [\![t]\!]_\Gamma$. Soit,

$$[\![C(x)\ n \to t]\!]_\Gamma = (\Gamma(x), n \to [\![t]\!]_\Gamma)$$

.

$$
\begin{array}{rcl}
[\![n!]\!]_\Gamma & = & n! \\
[\![\lambda.\ t]\!]_\Gamma & = & \lambda.\ [\![t]\!]_\Gamma \\
[\![t_0\ t_1]\!]_\Gamma & = & [\![t_0]\!]_\Gamma\ [\![t_1]\!]_\Gamma \\
[\![\texttt{let}\ t_0\ \texttt{in}\ t_1]\!]_\Gamma & = & \texttt{let}\ [\![t_0]\!]_\Gamma\ \texttt{in}\ [\![t_1]\!]_\Gamma \\
[\![\texttt{letrec}\ t_0\ \texttt{in}\ t_1]\!]_\Gamma & = & \texttt{letrec}\ [\![t_0]\!]_\Gamma\ \texttt{in}\ [\![t_1]\!]_\Gamma \\
[\![C(x)\ (t_1;\ldots;t_n)]\!]_\Gamma & = & C(\Gamma(x))\ ([\![t_1;\ldots;t_n]\!]_\Gamma) \\
[\![\texttt{match}\ t\ \texttt{with}\ \pi_1;\ldots;\pi_n]\!]_\Gamma & = & \texttt{match}\ [\![t]\!]_\Gamma\ \texttt{with}\ \texttt{sort}([\![\pi_1;\ldots;\pi_n]\!]_\Gamma) \\
\\
[\![C(x)\ n \to t]\!]_\Gamma & = & (\Gamma(x), n \to [\![t]\!]_\Gamma) \\
\\
[\![p]\!] & = & [\![p.\texttt{terme}]\!]_{\texttt{make\_num}(p.\texttt{type})}
\end{array}
$$

FIG. 2.4.2 – La numérotation

Traduction d'un filtrage :

Ainsi, la traduction d'une liste de clauses est une liste de couples de numéros et de clauses. La numérotation de $\pi_0, \ldots, \pi_n$ produit $(m_1, [\![\pi_1]\!]_\Gamma), \ldots, (m_n, [\![\pi_n]\!]_\Gamma)$, où les m_i sont les indices dans Γ correspondant au nom du constructeur de chaque clause. Utilisant les numéros m_i, nous trions cette liste afin d'ordonner le filtrage en utilisant la fonction `sort`. `sort` prend en paramètre une liste de couples numéros et clauses, trie ces couples en ordre croissant des numéros. En plus de trier la liste, `sort` vérifie que les numéros forment une séquence continue d'entiers $0, 1, 2, \ldots, n$ sans doublons. Cette vérification assure l'exhaustivité du filtrage. Enfin, la fonction `sort` efface les numéros et ne renvoie que les clauses. En effet, les clauses de εML# ne portent pas les numéros des constructeurs.

On peut alors transformer un filtrage $\texttt{match}\ t\ \texttt{with}\ \pi_1;\ldots;\pi_n$, on traduit t dans Γ ainsi que la liste des clauses. Puis, on applique la fonction `sort` à cette liste de clauses pour obtenir les clauses εML# correctement ordonnée. Soit :

$$
\begin{array}{rcl}
[\![\texttt{match}\ t\ \texttt{with}\ \pi_1;\ldots;\pi_n]\!]_\Gamma & = & \texttt{let}\ \vec{\pi'} = \texttt{sort}\ ([\![\pi_1;\ldots;\pi_n]\!]_\Gamma)\ \texttt{in} \\
 & & \texttt{match}\ [\![t]\!]_\Gamma\ \texttt{with}\ \vec{\pi'}
\end{array}
$$

Pour les autres termes, il s'agit d'une propagation de la transformation aux sous-termes. La transformation dans son intégralité est résumée sur la figure 2.4.2.

Présentation monadique Notre transformation est partielle, elle peut échouer. En effet, lors de la transformation, un constructeur nommé $C(x)$ peut ne pas avoir d'image par la numérotation Γ. C'est le cas si le constructeur n'a pas été déclaré préalablement dans un type

concret. La recherche du numéro associé à un constructeur pouvant échouer, la transformation d'un terme peut donc échouer. Nous gérons la possibilité d'échec et la propagation d'un échec au travers du type `option` polymorphe de Coq :

$$\Gamma(x) \quad = \quad \begin{cases} \texttt{Some}\ n & \text{si } \exists \tau.\ \tau \in \Pi \text{ et } \tau(n) = x\,; \\ \texttt{None} & \text{sinon} \end{cases}$$

De même :

$$[\![t]\!]_\Gamma \quad = \quad \begin{cases} \texttt{Some}\ t' & \text{si tout constructeur dans } t \text{ est déclaré}\,; \\ \texttt{None} & \text{sinon} \end{cases}$$

La traduction d'un terme échoue si celle de l'un des sous-termes échoue. Au niveau de l'implantation, cela se traduit par une imbrication de filtrages sur les traductions des sous-termes. Par exemple, la traduction de la liaison locale `let` a `in` b dans Γ :

```
match [[a]]_Γ with
| None => None
| Some p =>
     match [[b]]_Γ with
     | None => None
     | Some r =>
          Some (let p in r)
     end
end
```

Les filtrages imbriqués alourdissent grandement le code. En utilisant la monade d'erreur nous gagnons en lisibilité. Le type de la monade est le type option polymorphe de Coq. L'encapsulation est naturelle, x devient `Some` x. Enfin, nous définissons la fonction de composition (*bind*) :

```
Definition bind (A B : Set) (f : option A) (g : A -> option B) :=
  match res with None => None | Some x => g x end.
```

`bind` permet d'appliquer une fonction `(g : A -> option B)` à un objet de type `option A` et retourne un objet de type `option B`. Nous utilisons la notation *do* pour récupérer le résultat intermédiaire manipulé dans le bind en lui attribuant un nom. *do* $x \leftarrow a; b$ se lit comme `bind a (fun x -> b)`. On remarquera la similitude entre `bind` et le constructeur `let` .

Elle permet de calculer a avec effet de bord et d'en stocker le résultat dans une variable qui sera liée dans la suite du calcul.

La transformation de la liaison locale dans notre code devient :

```
do p <- [[a]]_Γ ;
do r <- [[b]]_Γ ;
Some (let p r)
```

On notera que l'environnement de traduction Γ n'apparaît pas au niveau de la traduction d'un terme. Il s'agit d'un paramètre global de cette transformation.

Préservation sémantique

Nous montrons la préservation sémantique de la numérotation par équivalence observationnelle. Autrement dit, si un programme p de εML s'évalue en la valeur v, et si la numérotation de p n'échoue pas, $[\![p]\!]$ s'évalue en une valeur sémantique v' de εML# et v' est observationnellement équivalente à v. (Nous définissons plus bas cette équivalence.)

Théorème 2.4.4 (Préservation sémantique de la numérotation)
Si

$$\vdash p \Rightarrow v$$

alors il existe v' tel que,

$$\vdash [\![p]\!] \Rightarrow v' \text{ où } v' \text{ est observationnellement équivalent à } v.$$

La preuve de ce théorème repose sur une simulation de l'évaluation d'un terme εML à constructeurs nommés. Cette simulation met en relation une évaluation d'un terme t et l'évaluation de sa numérotation $[\![t]\!]_\Gamma$.

Les invariants de la simulation définissent le cadre d'observation. Un terme t s'évalue selon le jugement εML à constructeurs nommés :

$$\Pi, e \vdash t \Rightarrow v,$$

tandis que sa numérotation s'évalue selon :

$$e' \vdash [\![t]\!]_\Gamma \Rightarrow v'.$$

Nous avons besoin de mettre en relation les cadres d'évaluation que sont les environnements et les valeurs sémantiques afin de pouvoir comparer une évaluation εML et une évaluation εML#.

Cohérence de la numérotation Le terme t est évalué par rapport à une liste de définitions de types concrets Π, correspondant au programme entier. La numérotation Γ est aussi définie à partir de la liste de types concrets du programme entier. Dans le cadre d'une observation, l'assurance que Γ est bien issue de Π est nécessaire.

Définition 2.4.5 (Cohérence de la numérotation)
Γ *est cohérent par rapport à* Π *si pour tout* $\tau \in \Pi$ *et* $c \in \tau$, *il existe* n *tel que* $\Gamma(c) = n$ *et* $\tau(n) = c$. *On note la cohérence :* $\Pi \vdash \Gamma$.

La cohérence se montre au niveau de l'évaluation d'un programme et se démontre par construction de Γ.

Correspondance des valeurs La cohérence de la numérotation se propage au niveau des valeurs. En effet, la valeur associée à un constructeur appliqué de εML, indique le nom du constructeur, tandis qu'en εML# elle porte le numéro du constructeur. La valeur $(C(x), v_1; \ldots; v_n)$ correspond à $(C(j), w_1; \ldots; w_n)$ si $\Gamma(x) = j$ et on propage de telles correspondances aux valeurs des arguments : v_i correspond à w_i pour $1 \leq i \leq n$.
Une fermeture simple (t, e) (resp. récursive $(t, e)_{rec}$) correspond à la fermeture simple (resp.

récursive) dont le corps est la numérotation de t, $[\![t]\!]_\Gamma$ et dont les environnements se correspondent.
La correspondance entre environnements est la propagation de la correspondance entre valeurs. La relation de correspondance entre valeurs est notée :

$$\Gamma \vdash v \sim v',$$

signifiant que la valeur sémantique v de εML correspond à la valeur sémantique v' de εML# selon la numérotation Γ. Cette relation se définit par le jeu de règles suivant :

$$\frac{\Gamma(x) = m \quad \Gamma \vdash v_i \sim w_i \quad (\text{pour tout } i \in [1;n])}{\Gamma \vdash (C(x), v_1; \ldots; v_n) \sim (C(m), w_1; \ldots; w_n)} \qquad \frac{\Gamma \vdash e \sim e'}{\Gamma \vdash (t, e) \sim ([\![t]\!]_\Gamma, e')}$$

$$\frac{\Gamma \vdash e \sim e'}{\Gamma \vdash (t, e)_{rec} \sim ([\![t]\!]_\Gamma, e')_{rec}}$$

$$\Gamma \vdash \varepsilon \sim \varepsilon \qquad \frac{\Gamma \vdash v \sim w \quad \Gamma \vdash e \sim e'}{\Gamma \vdash (v; e) \sim (w; e')}$$

Simulation Nous pouvons maintenant énoncer le lemme de simulation sur l'évaluation d'un terme εML :

Lemme 2.4.6 (Simulation)
Supposons $\Pi \vdash \Gamma$ *et* $\Gamma \vdash e \sim e'$. *Si*

$$\Pi, e \vdash t \Rightarrow v$$

alors il existe v' *telle que :*

$$e' \vdash [\![t]\!]_\Gamma \Rightarrow v' \text{ et } \Gamma \vdash v \sim v'.$$

La preuve se fait par induction sur une évaluation de εML à constructeurs nommés. Les cas intéressants sont, bien entendu, l'évaluation d'un constructeur appliqué et l'évaluation d'un filtrage.

Simulation de l'évaluation d'un constructeur appliqué Nous appliquons la simulation à l'évaluation du constructeur nommé appliqué $C(x)\ (t_1; \ldots; t_n)$. Nous montrons, avec les hypothèse d'inductions :
Supposons $\Pi \vdash \Gamma$ *et* $\Gamma \vdash e \sim e'$. *Si*

$$\Pi, e \vdash C(x)\ (t_1; \ldots; t_n) \Rightarrow (x, v_1; \ldots; v_n)$$

alors, il existe v' *telle que*

$$e' \vdash [\![C(x)\ (t_1; \ldots; t_n)]\!]_\Gamma \Rightarrow v'$$

et $\Gamma \vdash (x, v_1; \ldots; v_n) \sim v'$.

La numérotation ayant réussi, nous avons :
$\Gamma(x) = j$ et $[\![C(x)\ (t_1; \ldots; t_n)]\!]_\Gamma = C(j)\ ([\![t_1; \ldots; t_n]\!]_\Gamma)$.

En appliquant l'hypothèse d'induction sur les évaluations des arguments :
Supposons $\Pi \vdash \Gamma$ *et* $\Gamma \vdash e \sim e'$. *Si*

$$\Pi, e \vdash t_i \Rightarrow v_i, \ \textit{(pour tout } i \in [1;n]\textit{)}$$

alors, il existe $w_1; \ldots; w_n$ *telles que :*

$$e' \vdash [\![t_i]\!]_\Gamma \Rightarrow w_i, \ \textit{(pour tout } i \in [1;n]\textit{)}$$

et $\Gamma \vdash v_i \sim w_i$, *(pour tout* $i \in [1;n]$*).*
Nous obtenons l'existence de $w_1; \ldots; w_n$ telles que :
– $e' \vdash [\![t_i]\!]_\Gamma \Rightarrow w_i$, (pour tout $i \in [1;n]$) et
– $\Gamma \vdash v_i \sim w_i$, (pour tout $i \in [1;n]$) .
On conclut alors l'existence de $(j, w_1; \ldots; w_n)$. En effet :

$$\frac{e' \vdash [\![t_i]\!]_\Gamma \Rightarrow w_i \quad \text{(pour tout } i \in [1;n])}{e' \vdash C(j)\ ([\![t_1; \ldots; t_n]\!]_\Gamma) \Rightarrow (j, w_1; \ldots; w_n)}$$

et

$$\frac{\Gamma(x) = j \quad \Gamma \vdash v_i \sim w_i \quad \text{(pour tout } i \in [1;n])}{\Gamma \vdash (x, v_1; \ldots; v_n) \sim (j, w_1; \ldots; w_n)}$$

Simulation de l'évaluation d'un filtrage L'application de la simulation à l'évaluation d'un filtrage, nous définit la propriété suivante à prouver avec les hypothèses d'induction :
Supposons Γ^Π *et* $\Gamma \vdash e \sim e'$. *Si*

$$\Pi, e \vdash \texttt{match}\ t\ \texttt{with}\ \pi_1; \ldots; \pi_n \Rightarrow v$$

alors il existe v' *telle que :*

$$e' \vdash [\![\texttt{match}\ t\ \texttt{with}\ \pi_1; \ldots; \pi_n]\!]_\Gamma \Rightarrow v'$$

et $\Gamma \vdash v \sim v'$.
La numérotation du filtrage n'ayant pas échoué :

$$[\![\texttt{match}\ t\ \texttt{with}\ \pi_1; \ldots; \pi_n]\!]_\Gamma \quad = \quad \texttt{match}\ [\![t]\!]_\Gamma\ \texttt{with}\ \texttt{sort}([\![\pi_1; \ldots; \pi_n]\!]_\Gamma)$$

L'application de l'hypothèse d'induction sur l'évaluation de t nous fournit l'existence de v'_1 telle que :
– $\Pi, e \vdash [\![t]\!]_\Gamma \Rightarrow v'_1$ et
– $\Gamma \vdash (x_i, v_1; \ldots; v_k) \sim v'_1$.
Or, $\Gamma \vdash (x_i, v_1; \ldots; v_k) \sim v'_1$ si et seulement si $v'_1 = (j, w_1; \ldots; w_k)$ telle que :
– $\Gamma(x_i) = j$ et
– $\Gamma \vdash v_i \sim w_i$, (pour tout $i \in [1;k]$) .

Afin d'appliquer l'hypothèse d'induction sur l'évaluation du corps de la clause mise en jeu, nous construisons la correspondance entre les environnements :

$\Gamma \vdash v_k; \ldots; v_1; e \sim w_k; \ldots; w_1; e'$.

Il nous reste à montrer que $\texttt{sort}([\![\pi_1; \ldots; \pi_n]\!]_\Gamma)(j) = k \to [\![t]\!]_\Gamma$.

Si $\pi_i = x_i\ k \to t_i$ et $\Gamma(x_i) = j$ alors, $[\![\pi_i]\!]_\Gamma = (j, k \to [\![t_i]\!]_\Gamma)$.

En utilisant les résultats intermédiaires définissant la correction de `sort` :

`sort` fournit une liste triée : `is_sorted(sort` l).

`sort` fournit une liste telle que `enum` i l avec `enum` i l si $l = (i, \_), \ldots, (i + |l| - 1, \_)$.

De plus,

Lemme 2.4.7 *Si* `is_sorted` l *et* `enum` 0 l *alors* $l(i) = (i, \_)$.

Nous obtenons $\texttt{sort}([\![\pi_1; \ldots; \pi_n]\!]_\Gamma)(j) = k \to [\![t_i]\!]_\Gamma$ et appliquons l'hypothèse d'induction sur l'évaluation du corps de la clause mise en jeu. La conclusion s'en suit.

2.5 Lignes directrices de la compilation

Nous décrivons ici les lignes directrices de la compilation d'un langage purement fonctionnel de type miniML (le langage εML présenté dans la section 2.2) vers un langage impératif de bas niveau, Cminor. Ces lignes directrices dirigent les choix de langages intermédiaires et de transformations présents dans notre chaîne de compilation. Une vue de l'ensemble est donnée par la figure 2.5.1.

2.5.1 Des fonctions locales aux fonctions closes et globales

Les termes fonctionnels de notre langage source sont anonymes et disséminés dans la syntaxe des termes, ce sont des valeurs de première classe. Un programme εML n'est pas organisé comme une suite de fonctions globales. Comme dans le λ-calcul les fonctions sont des abstractions apparaissant dans un terme comme n'importe quel sous-terme.

N'étant pas distinguée syntaxiquement, une abstraction apparaît au milieu d'un terme quelconque et son corps peut dépendre du contexte qui précède, car il peut contenir des variables libres. Le corps d'une abstraction n'est donc pas forcément un terme clos pouvant être transformé directement en une fonction globale.

Cminor est un langage stratifié où les fonctions sont des entités distinctes des expressions et instructions. De plus, une fonction Cminor est une fonction globale, son corps est donc clos. Il est aisé de comprendre qu'une fonction Cminor ne peut suffire à exprimer une abstraction εML.

Lors d'une évaluation avec environnement, nous avons vu que la valeur associée à une abstraction est une fermeture. Une valeur fermeture est la combinaison d'une abstraction et de la copie de son environnement courant. Cet environnement permet d'associer aux variables libres dans le corps de l'abstraction leur valeur courante. Le passage de fonctions locales à des fonctions globales fait l'objet du chapitre 5 de ce manuscrit. La méthode que nous employons, ici, est la construction de fermetures explicites, dans la littérature anglophone *closure conversion*. L'esprit est de représenter une abstraction par la combinaison d'une fonction globale et d'un environnement restreint aux seules variables libres dans le

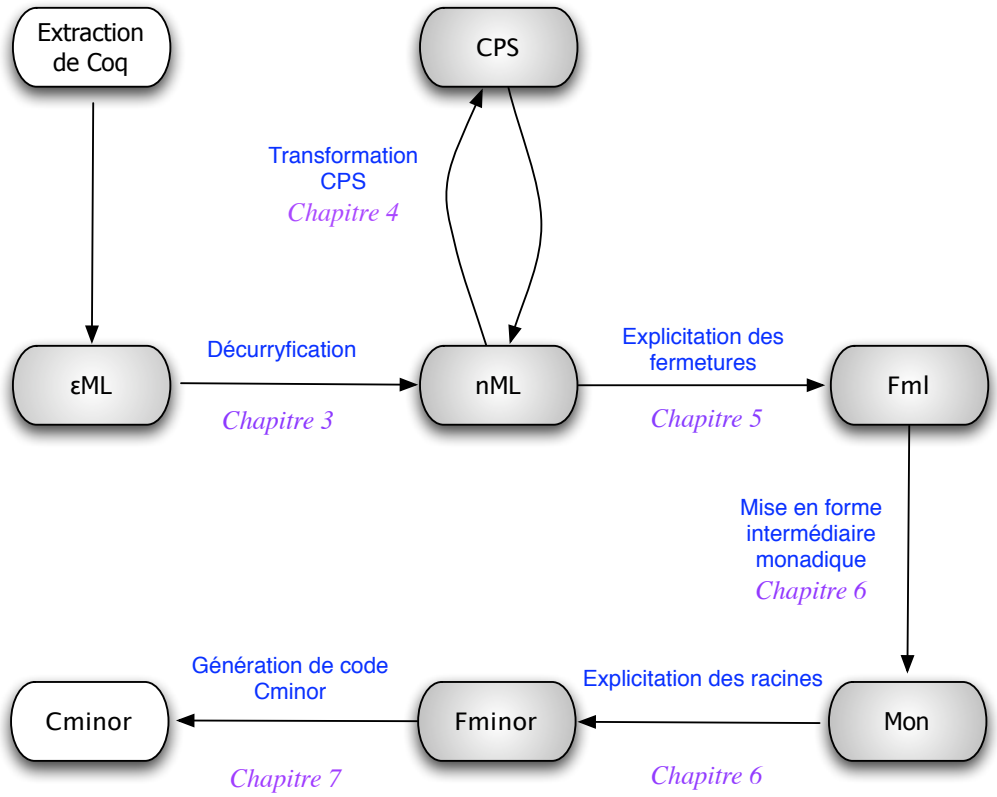


FIG. 2.5.1 – Le front-end miniML

corps de la fonction. Nous obtenons alors un programme organisé en fonctions globales et à l'ancienne position syntaxique d'une abstraction, on trouve des estampilles, une fermeture qui réfère la fonction et portent syntaxiquement les variables libres dans le corps de cette fonction, afin d'associer à ces variables leurs valeurs.

2.5.2 Représentation uniforme des données

Les structures de données de notre langage source εML sont des structures de données spécifiques aux langages fonctionnels de haut niveau.

Elles sont au nombre de deux. La première est la fermeture, qui comme nous l'avons dit plus haut à la section 2.5.1 ne peut clairement pas être représentée par une fonction Cminor. Les autres structures de données sont les constructeurs appliqués, qui, eux non plus, n'ont pas leur équivalent en Cminor. Cminor a un système de types de données organisé en flottants et entiers ou pointeurs.

De plus, nous nous plaçons dans le scénario où les programmes proviennent du mécanisme d'extraction de Coq, ils sont donc tous bien typés et normalisants, ce qui nous permet de considérer que notre langage source est non typé (bénéficiant des garanties du système de types de Coq).

Ces considérations convergent vers une représentation uniforme des données, dans la littérature anglophone *boxing*. Il s'agit de "mettre en bloc" les structures de données et de les manipuler de manière uniforme au travers des pointeurs sur les blocs contenant leurs données. Concernant une fermeture, un bloc représentant une fermeture contient un pointeur vers le code de sa fonction globale et les valeurs de ses variables libres. Cette mise en bloc des fermetures commence au chapitre 5 et est réalisée lors de la génération de code Cminor décrite au chapitre 7.

Un bloc représentant un constructeur appliqué contient une représentation identifiant (un entier) le constructeur mis en jeu et les valeurs de ses arguments. La mise en place de la représentation des constructeurs par leurs numéros a été présentée plus haut à la section 2.4.4. La mise en bloc des constructeurs est effective lors de la génération de code Cminor (voir le chapitre 7).

2.5.3 Optimisation des fonctions curryfiées et une porte ouverte vers d'autres optimisations

Dans un souci de réalisme, notre chaîne de compilation est optimisante. Nous avons mis à profit la capacité d'avoir des fonctions à plusieurs arguments en Cminor pour optimiser la compilation des abstractions unaires imbriquées de notre langage source : il s'agit de la décurryfication.

La représentation uniforme des données est coûteuse en allocation mémoire dans le tas. En effet, pour chaque structure de données, on alloue un bloc dans le tas. Certaines optimisations peuvent limiter le nombre de blocs alloués lors de l'exécution d'un programme. Nous avons implémenté l'une d'entre elles dans notre chaîne de compilation. Il s'agit de la décurryfication, qui est le sujet du chapitre 3. Les abstractions et les applications de notre langage source sont *curryfiées*, autrement dit elles sont unaires. Une abstraction qui prend n paramètres est encodée par une imbrication de n abstractions. Cet encodage sera représenté par n fermetures et donnera lieu à n allocations dans le tas. Les fonctions Cminor peuvent avoir plusieurs paramètres, tandis que la représentation des abstractions curryfiées produit des fonctions Cminor à un paramètre (en fait deux puisqu'elles doivent prendre en paramètre la fermeture elle-même voir le chapitre 5). On peut optimiser le nombre de fermetures allouées en mémoire en utilisant le fait que les fonctions Cminor peuvent prendre plusieurs paramètres. Pour cela, on décurryfie les abstractions quand cela est possible. Décurryfier une abstraction consiste à transformer une imbrication de n abstractions unaires en une abstraction $n-$aire. Une abstraction $n-$aire ne nécessite qu'une seule allocation dans le tas.

La décurryfication est une optimisation effectuée sur un langage fonctionnel à la ML. D'autres optimisations sont envisageables. Les optimisations sur les langages fonctionnels sont difficilement exprimables dans leur style direct (le style dans lequel εML est décrit). Le style de passage par continuation CPS est un style particulièrement adéquat aux optimisations de compilation de langages fonctionnels. Une transformation en forme CPS a été implantée dans notre chaîne de compilation, elle est décrite au chapitre 4.

2.5.4 Gérer les allocations : interagir avec un gestionnaire d'allocation

Notre chaîne de compilation vise à produire un compilateur réaliste et optimisant. Tout comme dans les compilateurs pour langages fonctionnels modernes, une gestion de mémoire automatique est nécessaire. Nous avons donc spécifié et certifié l'interaction des programmes issus de notre compilation avec un gestionnaire de mémoire automatique.

L'espace disponible dans le tas peut ne pas suffire à allouer toutes les données lors de la compilation d'un programme. Cependant, toutes les données allouées lors de la compilation d'un programme ne sont pas utiles jusqu'à la fin de l'exécution du programme. Cet espace inutile peut être recyclé afin d'allouer de nouvelles données. Dans les langages de bas niveau, comme C ou C++, le programmeur peut lui-même gérer la mémoire en désallouant les blocs dont il n'a plus l'utilité. On parle alors de désallocation explicite. Ce mécanisme n'est pas sûr. Dans le cadre de notre développement, nous désirons produire un compilateur sûr. A ces fins, nous préférons récupérer automatiquement les blocs alloués qui ne sont plus utiles, comme dans la compilation des langages de haut niveau, en général et des langages fonctionnels, en particulier. C'est ce que l'on appelle *la gestion de mémoire automatique*. Notre compilation vers Cminor doit donc produire un code capable d'interagir avec un gestionnaire de mémoire. Notre choix s'est tourné vers un gestionnaire de mémoire automatique à "garbage collection". Nous devons donc être en position de lui fournir les racines mémoire d'un programme à un point de déclenchement potentiel de GC. Ce processus est décrit au chapitre 6. Il donne lieu à deux transformations, la première nous mettant dans un style de programme où le calcul des racines est facilité, la forme intermédiaire monadique. La seconde matérialise l'ensemble des racines courantes d'une fonction, Fminor. L'interaction est encodée concrètement lors de la génération de code Cminor (voir le chapitre 7).

3 Décurryfication

Dans le monde mathématique, tout comme dans les langages à la C, tel que Cminor, les fonctions peuvent avoir plusieurs paramètres formels. Ainsi, la fonction `plus` sur les entiers prend comme paramètres deux entiers : `plus` $(x\ y)$. Une fonction à n paramètres formels est dite *n-aire*. L'application d'une fonction n-aire requiert la présence de ses n arguments. Ainsi, la fonction `plus` s'applique à deux arguments.

Dans les langages fonctionnels Caml [73], Haskell [57], SML [84] ou encore le langage de spécification de Coq, il n'y a pas de fonction à plusieurs paramètres. Comme dans le $\lambda-$calcul, ces langages disposent uniquement de fonctions unaires $(\lambda x.\ t)$ et d'applications unaires $(t_0\ t_1)$.

Il existe deux possibilités d'encodage des fonctions à plusieurs arguments : 1 – grouper les arguments de la fonction dans un $n-$uplet, 2 – *curryfier*.[1] Une abstraction est dite *curryfiée* si elle s'écrit sous la forme suivante :

$$\lambda x_0.\ \ldots \lambda x_n.\ t$$

(avec n maximal) c'est-à-dire qu'elle attend $n+1$ arguments et t n'est pas une abstraction. Au lieu de prendre tous les $n+1$ arguments et d'effectuer l'évaluation du corps d'abstraction, une abstraction curryfiée prend le premier argument et retourne une abstraction qui attend les n autres arguments. Cette abstraction, à son tour, prend un argument et retourne une abstraction qui attend $n-1$ autres arguments et ainsi de suite, jusqu'à la $n+1$ième abstraction qui a pour corps t et donc effectue l'évaluation de t.

Nous appelons *application partielle* une application dont le résultat est une fermeture partielle : tous les arguments ne sont pas encore reçus. Ainsi, la fonction `plus` s'écrit dans le $\lambda-$calcul : $\lambda x.\ \lambda y.\ x+y$. L'application suivante :$(\lambda x.\ \lambda y.\ x+y)\ 4$ est une application partielle, elle produit la fermeture partielle de paramètre formel y et de corps $x+y$ avec $x=4$.

Décurryfier une abstraction consiste à transformer une abstraction curryfiée en une fonction n-aire :

$$\lambda x_0.\ \ldots \lambda x_n.\ t$$

devient

$$\lambda[x_0;\ldots;x_n].\ t(\text{ Abstractions } n\text{-aire dont les paramètres formels sont } x_0;\ldots;x_n)$$

[1] Dans la littérature anglaise *currying*, nommé ainsi en hommage à Haskell B.Curry, ce dernier attribue lui-même cette technique à Schönfinkel [24], mais elle remonte à Frege d'après [16].

Elle reçoit les $n+1$ arguments d'un coup lors de l'application. La décurryfication constitue une optimisation couramment utilisée dans la compilation de langages fonctionnels [2, 77, 64].

La décurryfication est avantageuse lors de l'évaluation d'*applications totales* : applications imbriquées au nombre d'arguments attendus par une abstraction curryfiée. L'évaluation d'un terme de la forme

$$(\ldots(\lambda x_0.\ \ldots\lambda x_n.\ t)\ t_0)\ \ldots)\ t_n)$$

se décompose en $n+1$ appels de fonctions successifs, tandis que l'évaluation de ce terme après décurryfication

$$(\lambda[x_0;\ldots;x_n].\ t)\ [t_0;\ldots;t_n]$$

n'effectue qu'un seul appel de fonction. Un autre bénéfice de la décurryfication est que le code obtenu par compilation est plus économe en allocations. En effet, chaque application intermédiaire à t_i, pour $i < n$, nécessite l'allocation en mémoire d'une fermeture pour la fonction $\lambda x_{i+1}.\ \ldots\lambda x_n.\ t$ représentant l'application partielle. En revanche, l'application décurryfiée ne nécessite aucune allocation de ce genre, puisqu'elle ne produit pas de fermeture intermédiaire.

La décurryfication peut s'effectuer soit *dynamiquement*, au vol pendant l'exécution, soit *statiquement*, par une transformation du programme pendant la compilation. La décurryfication dynamique, comme effectuée par les machines abstraites ZAM [64] et G-machine [92], reconnaît davantage d'applications curryfiées, mais ralentit l'exécution : il faut tester dynamiquement l'arité des fonctions, et les arguments doivent être passés dans une pile, l'utilisation de registres pour les passer étant impossible. Au contraire, la décurryfication statique peut échouer à reconnaître certaines applications curryfiées, mais se prête à la génération de code machine efficace, utilisant les registres du processeur pour le passage d'arguments. Pour ces raisons, le système Objective Caml utilise la décurryfication dynamique dans son compilateur de bytecode et la décurryfication statique dans son compilateur optimisant. Nous suivons l'approche statique de la décurryfication, qui s'intègre mieux dans une chaîne de compilation optimisante passant par le langage intermédiaire Cminor.

La décurryfication statique d'un programme écrit dans un style curryfié nécessite une analyse statique de ce programme. Cette analyse permet de repérer les abstractions curryfiées et de convertir en conséquence les applications de ces abstractions.

Considérons le terme

$$\texttt{let}\ f = \lambda x.\ \lambda y.\ x + y\ \texttt{in}\ ((f\ 3)\ 4)$$

l'analyse repère une fonction curryfiée à deux paramètres formels et une application totale à deux arguments.

Sa décurryfication est

$$\texttt{let}\ f = \lambda[x;y].\ x + y\ \texttt{in}\ f\ [3;4]$$

Cependant, la décurryfication statique doit gérer spécialement les applications partielles. Si l'on considère maintenant le terme en style curryfié suivant :

$$\texttt{let}\ f = \lambda x.\ \lambda y.\ x + y\ \texttt{in}\ (f\ 3)$$

l'analyse repère la même abstraction curryfiée à deux paramètres formels. Or,

$$\texttt{let } f = \lambda[x; y].\ x + y \texttt{ in } f\ [3]$$

n'est pas une décurryfication correcte : f [3] n'a pas d'évaluation, puisqu'il s'agit d'une application partielle. L'utilisation de *combinateurs de curryfication* permet d'exprimer les appels partiels dans un langage décurryfié. Par exemple, une décurryfication correcte de l'exemple ci-dessus est

$$\texttt{let } f = \lambda x.\ \lambda y.\ x + y \texttt{ in } (\texttt{curried}_1\ [f])\ [3]$$

Les combinateurs de curryfication prennent une fonction décurryfiée d'arité $n + 1$ et retournent la fonction recurryfiée correspondante :

$$\texttt{curried}_n = (\lambda f.\ \lambda x_0.\ \ldots \lambda x_n.\ f\ [x_0; \ldots; x_n]).$$

L'analyse que nous mettons en jeu se restreint aux fonctions curryfiées liées par `let`. On pourrait également décurryfier les fonctions curryfiées anonymes (par opposition aux fonctions liées par `let` qui sont donc liées via des noms de variables). Cependant, leur fréquence ne justifie pas l'augmentation de la complexité de l'analyse. La décurryfication présentée ici s'effectue au travers d'une transformation de εML, dont les abstractions et applications sont unaires, vers une variante décurryfiée, nML, dont les abstractions et applications sont n-aires. Cette transformation est simultanée à une analyse statique qui détermine :

Les abstractions curryfiées liées par `let` :

$$\texttt{let } \underbrace{\lambda.\ \ldots.\ \lambda.}_{(n+1)}\ t \texttt{ in } \ldots$$

Les applications décurryfiables :

$$(\ldots(x\ t_0)\ldots t_n)$$

où x est une variable liée par `let` à une abstraction curryfiée d'arité $n + 1$.

L'analyse se base sur les formes syntaxiques des termes. La preuve de préservation sémantique utilise la même approche : les formes syntaxiques des valeurs fermetures peuvent être caractérisées selon l'évaluation les produisant. Cette caractérisation permet de définir un cadre d'équivalence observationnelle.

Nous commençons par présenter nML avant de décrire la transformation puis sa preuve de préservation sémantique. Enfin, une variante de nML, mieux adaptée aux besoins de la transformation `CPS` du chapitre 5, sera présentée ainsi que la transformation de nML vers cette variante.

Les travaux présentés dans ce chapitre ont fait l'objet d'une publication [28].

3.1 nML

nML est une variante de εML portant dans sa syntaxe les traits de la décurryfication : les fonctions et applications de fonctions sont à n arguments.

3.1.1 Syntaxe

Termes : $t ::= n!$

$\mid \lambda^n.\ t$	abstraction d'arité $n+1$
$\mid t\ [t_0;\ldots;t_n]$	application n-aire
$\mid$ `let` t_1 `in` t_2	
$\mid$ `letrec`^{n} t_1 `in` t_2	définition récursive d'une fonction d'arité $n+1$
$\mid C\ (t_1;\ldots;t_n)$	
$\mid$ `match` t_1 `with` $\pi_1;\ldots;\pi_n$	

Motif : $\pi ::= n \rightarrow t$

nML diffère de εML avec constructeurs numérotés, en trois points reflétant le caractère décurryfié du langage.

1) L'abstraction porte une information d'arité. $\lambda^n.\ t$ est une abstraction d'arité $n+1$ et de corps t.

2) La définition de fonction récursive porte, elle aussi, une information d'arité.
`letrec`^{n} t_1 `in` t_2 définit la fonction récursive d'arité $n+1$ et de corps t_1 dans t_2.

3) L'application n-aire $t\ [t_0;\ldots;t_n]$ est l'application décurryfiée à n arguments.

Le programme p pris en exemple dans le chapitre précédent 2.1 s'écrit en nML comme ci-dessous :

```
letrec² f [x,y]
  match x with
  | 0 -> y
  | 1(z) -> C(1) (f [z,y])
in
  let z=(C(1) (C(1) C(0))) in
    (  (x). f [x, (C(1) C(0)]) z
```

Nous avons nommé les variables pour plus de clarté. La fonction récursive définie localement est d'arité 2. Dans le corps de l'abstraction, l'appel récursif `(f [z,y])` est décurryfiée. Dans le sous terme droit de la définition, l'appel à la fonction récursive est aussi décurryfié : `f [x,(C(1) C(0))]`.

3.1.2 Sémantique naturelle avec environnement

La sémantique de nML suit une stratégie d'évaluation faible en appel par valeur avec évaluation des sous termes de gauche à droite.

Les valeurs fermetures portent une information d'arité en plus du corps et de l'environnement de fermeture.

Valeurs :	v	$::= (n, t, e)$	fermeture non récursive d'arité $n+1$
		$\mid (n, t, e)_{rec}$	fermeture récursive d'arité $n+1$
		$\mid (C, \vec{v})$	constructeur de données
Environnement :	e	$= \varepsilon$	environnement vide
		$\mid v; e$	

$$e \vdash (\lambda^n.\ t) \Rightarrow (n, t, e) \qquad \frac{e(n) = v}{e \vdash n! \Rightarrow v} \qquad \frac{e \vdash t_1 \Rightarrow v_1 \quad v_1; e \vdash t_2 \Rightarrow v}{e \vdash (\texttt{let}\ t_1\ \texttt{in}\ t_2) \Rightarrow v}$$

$$\frac{(n, t_1, e)_{rec}; e \vdash t_2 \Rightarrow v}{e \vdash \texttt{letrec}^n\ t_1\ \texttt{in}\ t_2 \Rightarrow v} \qquad \frac{e \vdash t \Rightarrow (n, t_b, e_1) \quad e \vdash t_i \Rightarrow v_i \quad \text{pour tout } i \in [0; n] \quad v_n; \ldots; v_0; e_1 \vdash t_b \Rightarrow v}{e \vdash t\ [t_0; \ldots; t_n] \Rightarrow v}$$

$$\frac{e \vdash t \Rightarrow (n, t_b, e_1)_{rec} \quad e \vdash t_i \Rightarrow v_i \quad \text{pour tout } i \in [0; n] \quad v_n; \ldots; v_0; (n, t_b, e_1)_{rec}; e_1 \vdash t_b \Rightarrow v}{e \vdash t\ [t_0; \ldots; t_n] \Rightarrow v}$$

$$\frac{e \vdash t_i \Rightarrow v_i \quad \text{pour tout } i \in [1; n]}{e \vdash C\ (t_1; \ldots; t_n) \Rightarrow (m, (v_1; \ldots; v_n))}$$

$$\frac{e \vdash t \Rightarrow (i, (v_1; \ldots; v_k)) \quad \pi_i = k \rightarrow t_i \quad v_k; \ldots; v_1; e \vdash t_i \Rightarrow v}{e \vdash \texttt{match}\ t\ \texttt{with}\ \pi_1; \ldots; \pi_n \Rightarrow v}$$

Une abstraction d'arité $(n+1)$, $\lambda^n.\ t$ dans l'environnement e, s'évalue en la fermeture n-aire (n, t, e).
La définition locale récursive $\texttt{letrec}^n\ t_1\ \texttt{in}\ t_2$ s'évalue dans e en v si t_2 dans l'environnement $(n, t_1, e)_{rec}; e$ s'évalue en v.
Enfin, une application décurryfiée $t\ [t_0; \ldots; t_n]$ s'évalue en v dans e, si t dans e s'évalue en la fermeture d'arité $n+1$ (n, t_b, e_1), que $t_0; \ldots; t_n$ s'évaluent en $v_0; \ldots; v_n$ alors t_b dans $v_n; \ldots; v_0; e_1$ s'évalue en v.
De même, une application d'une fermeture récursive $(n, t_b, e_1)_{rec}$ s'évalue en v si t_b dans $(v_n; \ldots; v_0; (n, t_b, e_1)_{rec}; e_1)$ s'évalue en v.
On remarquera qu'une application ne s'évalue que si tous ses arguments sont présents.

Concernant l'évaluation des autres termes, elle est identique à celle de εML.

3.2 Algorithme

La transformation menant à la décurryfication d'un terme εML s'effectue simultanément à l'analyse statique détectant les abstractions et applications décurryfiables. Nous détaillons d'abord l'analyse statique, puis nous donnons une présentation algorithmique de la transformation. L'algorithme n'est pas récursif structurellement sur les termes nML. En effet,

il nécessite quelques analyses structurelles des sous-termes, notamment dans le cas de la décurryfication d'une application. Nous voulons identifier les applications décurryfiables de la forme

$$((\ldots(n!\ t_0)\ldots)\ t_{m-1})\ t_m$$

où l'on sait que $n!$ est liée à une abstraction d'arité $(n+1)$. Pour cela, lors de la transformation d'une application $(t_a\ t_b)$, nous voulons tester si t_a est de la forme :

$$(\ldots(n!\ t_0)\ldots)\ t_{m-1})$$

(où $n!$ est connue d'arité $(m+1)$) avant de choisir la traduction. Or, nous nous retrouvons alors dans une stratégie non structurelle récursive. Notre implémentation en `Coq` est un peu différente de la présentation algorithmique, pour contourner cette limitation. Enfin, cette implantation est peu intuitive et mal adaptée à la preuve ; nous présentons donc une spécification relationnelle de l'algorithme. La présentation relationnelle est plus proche de la version algorithmique et plus adéquate au raisonnement.

3.2.1 Analyse statique

L'analyse statique qui détermine les possibilités de décurryfication fait partie de la famille générale des analyses de flot de contrôle (*control-flow analyses* ou *closure analyses* dans la littérature anglophone) [105, 90]. De manière générale, ces analyses associent à chaque variable et à chaque sous-expression d'un programme fonctionnel, une sur-approximation de ses valeurs possibles. En particulier, pour une application $t\ t'$, l'analyse de flot détermine un ensemble de fonctions $\{\ldots\lambda x_i.t_i\ldots\}$ en lesquelles t peut s'évaluer. Lorsque cet ensemble ne contient qu'un élément, la fonction appelée est *statiquement connue* et plusieurs optimisations deviennent possibles : expanser la fonction (*inlining*), décurryfier l'appel si la fonction est n-aire, spécialiser la représentation de sa fermeture (*lightweight closure conversion*), *etc.*

Les analyses de flot de contrôle classiques utilisent des itérations de point fixe, les rendant coûteuses en temps de compilation et difficiles à certifier en `Coq`. Nous allons donc utiliser une analyse très simplifiée, qui garde seulement trace des arités des fonctions liées par `let`. On notera qu'une analyse simplifiée similaire est utilisée dans le compilateur optimisant d'OCaml.

L'analyse statique vise dans un premier temps à reconnaître de potentielles abstractions n-aires associées à une variable liée, en attribuant à de telles variables une indication d'arité. Dans un second temps, munie de ces informations, elle détecte de telles variables dans des applications curryfiées totales, c'est-à-dire dans lesquelles tous les arguments sont présents. Cette analyse suppose la connaissance d'informations liées aux variables apparaissant dans un terme. Elle se fait donc au travers d'un environnement qui associe à chaque variable une information indiquant le caractère fonctionnel et l'arité potentielle.

Les informations manipulées par l'analyse sont définies comme suit :

$$\gamma ::= \texttt{Un} \mid \texttt{Kn}(m).$$

Nous associerons à une variable liée à une abstraction une information de la forme $\texttt{Kn}(m)$ où m précise que la décurryfication donnerait une fonction d'arité $m+1$. Si la variable est liée à autre chose qu'une abstraction, nous lui associerons l'information `Un`.

Considérons le terme suivant : `let` $\lambda.\ \lambda.\ t$ `in` t_2. Nous associerons l'information $\texttt{Kn}(1)$ à la variable liée par le `let`. En effet, elle est liée à une abstraction dont la décurryfication

serait d'arité 2. Par contre, dans le terme `let` $(t_1\ t_2)$ `in` t, nous associerons à la variable liée par le `let` l'information `Un`.

Nous avons besoin de propager ces informations le long de l'analyse, afin de les utiliser, notamment dans le traitement des applications. Pour ce faire, nous les propageons au travers d'un environnement :

$$\Gamma ::= \varepsilon \mid \gamma;\Gamma.$$

Donnons nous quelques abréviations pour plus de lisibilité :

– Les abstractions curryfiées :

$$\texttt{nabs}\ m\ t = \underbrace{\lambda.\ \ldots\lambda}_{m+1}.\ t.$$

– Les applications imbriquées :

$$\texttt{napp}\ t\ (t_0;\ldots;t_m) = (\ldots(t\ t_0)\ldots\ t_m).$$

Repérer les abstractions curryfiées se fait tout simplement en repérant les définitions locales (`let` t_1 `in` t_2) où t_1 est de la forme `nabs` $m\ t$, auquel cas, cette variable sera associée à l'information `Kn`(m) dans l'environnement d'analyse. Sinon, on lui associera l'information `Un`. La fonction `is_curried_abs` teste si un terme t est une abstraction curryfiée, si c'est le cas, elle retourne l'information `Kn`(n) (où l'arité est $n+1$) sinon elle retourne l'information `Un`.

$$\texttt{is\_curried\_abs}\ (t) = \begin{cases} \texttt{Kn}(m) & \text{si } t = \texttt{nabs}\ m\ t_1 \\ \texttt{Un} & \text{sinon} \end{cases}$$

La décurryfication des applications nécessite le repérage des applications imbriquées (`napp` $t\ (t_0;\ldots;t_m)$ où l'appelé le plus interne t est une variable $n!$ que l'on sait liée à une abstraction curryfiée d'arité $m+1$, c'est-à-dire qu'elle est associée à `Kn`(m) dans l'environnement d'analyse. Comme l'algorithme de transformation est récursif sur la structure du terme εML, il s'agit de repérer les applications $(t_a\ t_b)$ dont le sous-terme gauche t_a est de la forme (`napp` $n!\ (t_0;\ldots;t_{m-1})$) avec $\Gamma(n) =$ `Kn`(m) où Γ est l'environnement d'analyse. En effet, en appliquant le sous-terme droit t_b, on obtient une application totale à $m+1$ arguments.

Lors de l'analyse, l'environnement Γ se comporte comme une approximation de l'environnement d'évaluation. Si on analyse le terme $\lambda.\ t$ dans l'environnement Γ alors, on analyse le sous-terme t dans l'environnement `Un`$;\Gamma$. Si `let` t_1 `in` t_2 s'analyse dans Γ alors t_2 s'analyse dans `is_curried_abs` $t_1;\Gamma$. En particulier, si $t_1 =$ `nabs` $m\ t_0$ alors on considère t_0 dans $\underbrace{\texttt{Un};\ldots;\texttt{Un}}_{m+1};\Gamma$ que nous notons $\texttt{Un}^{(m+1)}\ \Gamma$.

Nous effectuons l'analyse et la traduction d'un terme εML en un terme nML en même temps, en utilisant l'environnement d'analyse Γ comme environnement de traduction. Les mécanismes décrits plus haut permettent alors de traduire convenablement les applications, les sous-termes gauches de liaisons locales, selon que ce sont des abstractions ou pas, et les variables.

3.2.2 Combinateur de curryfication

Le langage εML permet d'exprimer des applications partielles. Considérons le terme εML suivant :

$$\texttt{let}\ \lambda.\ \lambda.\ \lambda.\ (2!\ +\ 1!)\ -\ 0!\ \texttt{in}\ (0!\ 5).$$

Pour plus de clarté, voici le même terme avec variables nommées :

$$\texttt{let f} = \lambda x.\ \lambda y.\ \lambda z.\ (x\ +\ y)\ -\ z\ \texttt{in}\ (f\ 5).$$

Ce terme a un sens en εML (supposons + et − définis et 5 encodé en entier `nat`). Lors de la décurryfication, l'analyse signale la variable nouvellement créée comme étant liée à une abstraction d'arité 3, c'est-à-dire $\texttt{K}n(2)$. Le sous-terme gauche devient alors $\lambda^2.\ (2! + 1!) - 0!$ (soit avec variables nommées $\lambda[x; y; z].\ (x\ +\ y)\ -\ z$). Considérons maintenant le sous-terme droit, l'application $(0!\ 5)$ (sans codage $f\ 5$). Il s'agit d'une application partielle. La traduction directe, sous terme par sous terme, produirait un terme qui n'a pas d'évaluation en nML. En effet, les règles d'évaluation de l'application présument la présence de tous les arguments, seules les applications totales ont une évaluation en nML. Pour encoder les applications partielles, nous utilisons les combinateurs de curryfication. Ces combinateurs permettent de consommer un par un les arguments d'une application n-aire via une séquence de n η-expansions. Autrement dit, il curryfie l'application n-aire.

Notons $\texttt{NCurry}(m)$ le combinateur de curryfication à $m + 1$ arguments défini comme suit :

$$\lambda^0.\ \underbrace{\lambda^0 \ldots \lambda^0}_{m+1}.\ (m+1)!\ [m!, \ldots, 0!].$$

Soit avec des variables nommées :

$$\lambda[f].\ \lambda[x_0] \ldots \lambda[x_m].\ f\ [x_0, \ldots, x_m].$$

Concernant notre application partielle : $(f\ 5)$, nous utilisons le combinateur d'arité 3 :

$$\lambda[f].\ \lambda[x].\ \lambda[y].\ \lambda[z].f\ [x; y; z],$$

appliqué à l'abstraction et au premier argument 5.

$$(\lambda[f]\ .\lambda[x].\ \lambda[y].\ \lambda[z].f\ [x; y; z])[f][5].$$

En effet, ces deux applications imbriquées ont une évaluation et cette évaluation résulte en une fermeture partielle qui attend les deux autres arguments avant de pouvoir effectuer l'application totale (corps le plus interne). La traduction d'une variable $n!$ que l'on sait liée à une abstraction d'arité $m+1$ sera son application au combinateur de curryfication d'arité $m + 1$. Ainsi, la traduction d'application partielle se fera terme par terme.

Le combinateur de curryfication d'arité m, appliqué à un premier argument s'évalue par la règle dérivée suivante :

$$\frac{e \vdash t \Rightarrow v}{e \vdash \texttt{NCurry}(m)\ [t] \Rightarrow (0,\ \underbrace{\lambda^0. \ldots . \lambda^0}_{m}.\ (m)![(m-1)!; \ldots; 0!], v; e)}\ \text{(eval-}\texttt{NCurry}\text{)}$$

Notons que la forme de la valeur fermeture est très expressive. Elle indique dans sa structure syntaxique :

- le nombre d'arguments manquant pour avoir une application totale : c'est le nombre de λ dans le corps de la fermeture, ici m.

- L'arité de cette application totale : c'est la forme syntaxique de l'application du corps le plus interne.
- Les valeurs des arguments déjà passés : elles sont dans l'environnement de fermeture.
- L'abstraction décurryfiée mise en jeu, ici v : elle aussi stockée dans l'environnement de fermeture.

Cette caractérisation nous permettra de raisonner sur les différentes formes de fermetures apparaissant par décurryfication. Cela forme la base principale du raisonnement nous menant à la preuve de correction de cette décurryfication.

3.2.3 Présentation de l'algorithme

Nous pouvons maintenant présenter l'algorithme de décurryfication d'un terme εML en un terme nML.

$$[\![t, \Gamma]\!]$$

désigne la décurryfication du terme εML t dans l'environnement d'analyse Γ.

Décurryfication d'abstraction liée : Les abstractions décurryfiables sont celles liées localement. Lors de la traduction du `let`, l'analyse statique détermine selon la structure syntaxique du sous-terme gauche t_1 :

- s'il s'agit d'une abstraction décurryfiable $t_1 = \underbrace{\lambda. \ldots \lambda.}_{m+1}\ t$, alors on décurryfie l'abstraction en $\lambda^m.\ [\![t, \mathtt{Un}^{(m+1)}\ \Gamma]\!]$. La traduction du corps d'abstraction se fait dans l'environnement $\mathtt{Un}^{(m+1)}\ \Gamma$. En effet, les paramètres formels d'abstraction ne sont pas décurryfiés. L'analyse associe l'information $\mathtt{Kn}(m)$ dans l'environnement de traduction du sous-terme droit.
- t_1 n'est pas une abstraction décurryfiable, t_1 est décurryfié dans l'environnement courant. Le sous-terme droit est décurryfié dans l'environnement où l'analyse associe `Un` à la nouvelle variable.

On a donc deux cas de décurryfication d'une liaison locale :

$$\begin{aligned} [\![\mathtt{let}\ \underbrace{\lambda. \ldots \lambda.}_{m+1}\ t\ \mathtt{in}\ t_2, \Gamma]\!] &= \mathtt{let}\ \lambda^m.\ [\![t, \mathtt{Un}^{(m+1)}\ \Gamma]\!]\ \mathtt{in}\ [\![t_2, \mathtt{Kn}(m); \Gamma]\!] \\ [\![\mathtt{let}\ t_1\ \mathtt{in}\ t_2, \Gamma]\!] &= \mathtt{let}\ [\![t_1, \Gamma]\!]\ \mathtt{in}\ [\![t_2, \mathtt{Un}; \Gamma]\!] \end{aligned}$$

Décurryfication d'abstractions récursive : Nous ne décurryfions pas les abstractions unaires. Au niveau des définitions locales d'abstractions récursives, la traduction diffère selon que l'abstraction soit unaire ou $n-$aire. L'analyse effectue la même discrimination syntaxique sur le sous-terme gauche t_1 que dans le cas de la liaison locale :

- $t_1 = \underbrace{\lambda. \ldots \lambda.}_{m+1}\ t$, alors on décurryfie l'abstraction en $\lambda^m.\ [\![t, \mathtt{Un}^{(m+2)}\ (\mathtt{Kn}(m+1); \Gamma]\!]$. La traduction du corps d'abstraction se fait dans l'environnement $\mathtt{Un}^{(m+2)}\ (\mathtt{Kn}(m+1)\Gamma$. En effet, dans t, la variable récursive est $(m+2)$!. Au niveau de l'arité, il ne faut pas oublier le paramètre formel lié au niveau du `letrec`. L'analyse associe l'information $\mathtt{Kn}(m+1)$ dans l'environnement de traduction du sous-terme droit.

– t_1 n'est pas une abstraction décurryfiable, t_1 est décurryfié dans $(\mathtt{Un};\mathtt{Un};\Gamma)$. Le paramètre formel et la variable récursive ne sont pas décurryfiés. Le sous-terme droit est décurryfié dans l'environnement où l'analyse associe `Un` à la nouvelle variable.

La traduction algorithmique de la définition d'abstraction récursive locale se définit comme suit :

$$\begin{aligned}[\![\mathtt{letrec}\ \underbrace{\lambda.\ \ldots\lambda.}_{m+1}\ t\ \mathtt{in}\ t_2, \Gamma]\!] &= \mathtt{letrec}^{m+1}\ [\![t, \mathtt{Un}^{(m+2)}\ (\mathtt{Kn}(m+1);\Gamma)]\!] \\ &\quad\ \mathtt{in}\ [\![t_2, \mathtt{Kn}(m+1);\Gamma]\!] \\ [\![\mathtt{letrec}\ t_1\ \mathtt{in}\ t_2, \Gamma]\!] &= \mathtt{letrec}^{0}\ [\![t_1, \mathtt{Un};\mathtt{Un};\Gamma]\!]\ \mathtt{in}\ [\![t_2, \mathtt{Un};\Gamma]\!]\end{aligned}$$

Traduction des variables : La traduction des variables est guidée par l'analyse. Une variable dont l'analyse ne sait rien, c'est-à-dire soit un paramètre formel d'abstraction ou de motif, soit une variable liée localement à une abstraction unaire ou à un terme qui n'est pas une abstraction, est traduite à l'identique.

Si une variable $n!$ lie une abstraction décurryfiable d'arité $m+1$, alors l'analyse lui associe l'information $\mathtt{Kn}(m)$. Cette variable sera alors traduite par son application au combinateur de curryfication d'arité $m+1$.

Une variable $n!$ liée à une abstraction d'arité 1, c'est-à-dire $\Gamma(n) = \mathtt{Kn}(0)$, est traduite telle qu'elle. Il est inutile de l'appliquer à un combinateur de curryfication puisqu'elle ne peut être partiellement appliquée.

Au niveau algorithmique :

$$[\![n!, \Gamma]\!] = \begin{cases} \mathtt{NCurry}(m)\ n! & \text{si } \Gamma(n) = \mathtt{Kn}(m) \text{ et } m > 0\,; \\ n! & \text{sinon} \end{cases}$$

Décurryfication d'applications : Les applications décurryfiables se repèrent syntaxiquement. Il s'agit d'imbrications d'applications de forme : $(\ldots(t\ t_0)\ldots t_m)$ où l'appelant le plus interne t est une variable $n!$, telle que l'analyse associe à n l'information $\mathtt{Kn}(m)$. C'est donc une application totale et elle est décurryfiée.

Dans tous les autres cas, c'est-à-dire application partielle dont l'appelant le plus interne est une variable liée à une abstraction décurryfiable, ou toute autre application, la traduction produit une application unaire.

La décurryfication de l'application se résume algorithmiquement comme suit :

$$\begin{aligned}[\![(\ldots(n!\ t_0)\ldots)\ t_{m-1})\ t_m, \Gamma]\!] &= n!\ [[\![t_0;\ldots;t_m, \Gamma]\!]] \text{ avec } \Gamma(n) = \mathtt{Kn}(m) \\ [\![t_1\ t_2, \Gamma]\!] &= [\![t_1, \Gamma]\!]\ [[\![t_2, \Gamma]\!]]\end{aligned}$$

Traduction des abstractions anonymes : Les abstractions non liées localement sont traduites par des abstractions unaires. Ainsi :

$$[\![\lambda.\ t, \Gamma]\!] = \lambda^0.\ [\![t, \mathtt{Un};\Gamma]\!]$$

Le corps de l'abstraction est traduit dans l'environnement $\mathtt{Un};\Gamma$.

Décurryfication d'un motif : Les liaisons par paramètres formels ne sont pas soumises à l'analyse statique. Le corps d'un motif $m \to t$ est traduit dans l'environnement associant l'information $\mathtt{Un}$ à tous ses paramètres formels :

$$[\![m \to t, \Gamma]\!] \quad = \quad m \to [\![t, \mathtt{Un}^{(m)}\ \Gamma]\!]$$

Concernant les autres termes, la décurryfication s'effectue récursivement. L'algorithme est résumé sur la figure 3.2.1 .

$$\begin{array}{rcl}
[\![n!, \Gamma]\!] & = & \begin{cases} \mathtt{NCurry}(m)\ n! & \text{si } \Gamma(n) = \mathtt{Kn}(m)\,; \\ n! & \text{sinon} \end{cases} \\
[\![\lambda.\ t, \Gamma]\!] & = & \lambda^0.\ [\![t, \mathtt{Un}; \Gamma]\!] \\
\\
[\![(\ldots(n!\ t_0)\ldots)\ t_{m-1})\ t_m, \Gamma]\!] & = & n!\ [[\![t_0; \ldots; t_m, \Gamma]\!]] \text{ avec } \Gamma(n) = \mathtt{Kn}(m+1) \\
[\![t_1\ t_2, \Gamma]\!] & = & [\![t_1, \Gamma]\!]\ [[\![t_2, \Gamma]\!]] \\
\\
[\![\mathtt{let}\ \underbrace{\lambda.\ \ldots \lambda.}_{m+1}\ t\ \mathtt{in}\ t_2, \Gamma]\!] & = & \mathtt{let}\ \lambda^m.\ [\![t, \mathtt{Un}^{(m+1)}\ \Gamma]\!]\ \mathtt{in}\ [\![t_2, \mathtt{Kn}(m); \Gamma]\!] \\
[\![\mathtt{let}\ t_1\ \mathtt{in}\ t_2, \Gamma]\!] & = & \mathtt{let}\ [\![t_1, \Gamma]\!]\ \mathtt{in}\ [\![t_2, \mathtt{Un}; \Gamma]\!] \\
\\
[\![\mathtt{letrec}\ \underbrace{\lambda.\ \ldots \lambda.}_{m+1}\ t\ \mathtt{in}\ t_2, \Gamma]\!] & = & \mathtt{letrec}^{m+1}\ [\![t, \mathtt{Un}^{(m+2)}\ (\mathtt{Kn}(m+1); \Gamma)]\!] \\
 & & \mathtt{in}\ [\![t_2, \mathtt{Kn}(m+1); \Gamma]\!] \\
[\![\mathtt{letrec}\ t_1\ \mathtt{in}\ t_2, \Gamma]\!] & = & \mathtt{letrec}^0\ [\![t_1, \mathtt{Un}; \mathtt{Un}; \Gamma]\!]\ \mathtt{in}\ [\![t_2, \mathtt{Un}; \Gamma]\!] \\
\\
[\![C\ (t_1; \ldots; t_m), \Gamma]\!] & = & C([\![t_1; \ldots; t_m, \Gamma]\!]) \\
[\![\mathtt{match}\ t\ \mathtt{with}\ \pi_0; \ldots; \pi_n, \Gamma]\!] & = & \mathtt{match}\ [\![t, \Gamma]\!]\ \mathtt{with}\ [\![\pi_0; \ldots; \pi_n, \Gamma]\!] \\
[\![t_0; \ldots; t_n, \Gamma]\!] & = & [[\![t_0, \Gamma]\!]; \ldots; [\![t_n, \Gamma]\!]]
\end{array}$$

FIG. 3.2.1 – Algorithme de décurryfication

3.2.4 Présentation relationnelle

Dans le cadre d'une preuve de préservation sémantique, nous utilisons une relation binaire paramétrée par un environnement de traduction identique à celui défini plus haut. Il s'agit de la présentation relationnelle de l'algorithme de décurryfication.

L'utilisation d'une relation nous permet de discriminer directement sur la forme syntaxique des termes εML. Nous ne sommes pas restreints à raisonner uniquement sur les formes récursives structurelles comme dans la présentation fonctionnelle.

La relation de décurryfication se traduit par le jugement :

$$\Gamma \vdash t \ \sim \ t'.$$

Ce jugement se lit comme suit : le terme εML t se décurryfie en le terme nML t' dans l'environnement Γ. Cet environnement est identique à celui défini plus haut.

Contrairement à l'algorithme, la relation ne construit pas le terme t' nML correspondant à la décurryfication d'un terme t εML, elle énonce sous quelles conditions t' est la décurryfication de t.

En particulier, lors de la décurryfication d'une application décurryfiable, nous pouvons utiliser un prédicat caractérisant la forme du sous-terme gauche (égalité syntaxique). Ce prédicat précise s'il s'agit d'une application décurryfiable, c'est-à-dire si c'est une imbrication d'application attendant un dernier argument pour être décurryfié ($(\ldots(n!\ t_0)\ldots)t_{m-1}$) avec $\Gamma(n) = \texttt{Kn}(m+1)$) ou pas (les autres cas).

S'il s'agit d'une application partielle composée de m applications d'une variable dont on sait que l'arité est de $m+1$ alors on décurryfie l'application en ajoutant le sous-terme droit de l'appel.

$$\frac{\Gamma(n) = \texttt{Kn}(m+1) \qquad \Gamma \vdash t_i \ \sim \ n_i, (\text{pour tout } i \in [0; m])}{\Gamma \vdash (\ldots(n!\ t_0)\ldots)t_{m-1})\ t_m \ \sim \ n!\ (n_0; \ldots; n_m)}$$

De même, nous regroupons tous les autres cas d'application en une seule règle. Le même prédicat,

$$\frac{\Gamma \vdash t_1 \ \sim \ n_1 \qquad \Gamma \vdash t_2 \ \sim \ n_2}{\Gamma \vdash t_1\ t_2 \ \sim \ n_1\ (n_2)}$$

Concernant la décurryfication des abstractions, lors des liaisons locales, simples ou récursives, on discrimine selon la forme du sous-terme gauche.

La relation dans son intégralité est définie dans la figure 3.2.2

3.3 Préservation sémantique

La décurryfication étant formellement définie par la traduction d'un terme εML en un terme nML, nous pouvons entrer dans le vif du sujet : sa preuve de correction. Il nous faut montrer qu'il y a préservation du comportement entre un terme εML et sa traduction. Autrement dit il faut montrer que l'évaluation de l'un correspond à l'évaluation de l'autre. Il nous faut matérialiser cette notion de correspondance avant d'exhiber la preuve de correction. Il s'agit de mettre en évidence un lien entre valeurs εML et nML. Ce lien est donné sous forme d'une relation paramétrée par l'approximation d'évaluation donnée par l'analyse statique. En effet, l'analyse permet de donner une approximation des formes des fermetures apparaissant dans un programme εML ainsi que dans sa décurryfication.

$$\frac{\Gamma(n) = \texttt{Un}}{\Gamma \vdash n! \ \sim \ n!} \qquad \frac{\Gamma(n) = \texttt{Kn}(m)}{\Gamma \vdash n! \ \sim \ \texttt{Curry}\ (m)\ n!} \qquad \frac{\texttt{Un}; \Gamma \vdash t \ \sim \ n}{\Gamma \vdash \lambda.\ t \ \sim \ \lambda^0.\ n}$$

$$\frac{t_1 \neq \lambda.\ \ldots \qquad \Gamma \vdash t_1 \ \sim \ n_1 \qquad \texttt{Un}; \Gamma \vdash t_2 \ \sim \ n_2}{\Gamma \vdash \texttt{let}\ t_1\ \texttt{in}\ t_2 \ \sim \ \texttt{let}\ n_1\ \texttt{in}\ n_2}$$

$$\frac{t_1 \ = \ \texttt{nabs}\ m\ t \qquad \texttt{Un}^{(m+1)}; \Gamma \vdash t \ \sim \ n \qquad \texttt{Kn}(m); \Gamma \vdash t_2 \ \sim \ n_2}{\Gamma \vdash \texttt{let}\ t_1\ \texttt{in}\ t_2 \ \sim \ \texttt{let}\ \lambda^m.\ n\ \texttt{in}\ n_2}$$

$$\frac{t_1 \neq \lambda.\ \ldots \qquad \texttt{Un}; \texttt{Un}; \Gamma \vdash t_1 \ \sim \ n_1 \qquad \texttt{Un}; \Gamma \vdash t_2 \ \sim \ n_2}{\Gamma \vdash \texttt{letrec}\ t_1\ \texttt{in}\ t_2 \ \sim \ \texttt{letrec}\ n_1\ \texttt{in}\ n_2}$$

$$\frac{t_1 \ = \ \texttt{nabs}\ m\ t \qquad \texttt{Un}^{(m+2)}; \texttt{Kn}(m+1); \Gamma \vdash t \ \sim \ n \qquad \texttt{Kn}(m+1); \Gamma \vdash t_2 \ \sim \ n_2}{\Gamma \vdash \texttt{letrec}\ t_1\ \texttt{in}\ t_2 \ \sim \ \texttt{letrec}^{m+1}\ n\ \texttt{in}\ n_2}$$

$$\frac{\Gamma(n) = \texttt{Kn}(m+1) \qquad \Gamma \vdash t_i \ \sim \ n_i, (\text{pour tout } i \in [0; m])}{\Gamma \vdash (\ldots(n!\ t_0)\ldots) t_{m-1})\ t_m \ \sim \ n!\ (n_0; \ldots; n_m)} \qquad \frac{\Gamma \vdash t_1 \ \sim \ n_1 \qquad \Gamma \vdash t_2 \ \sim \ n_2}{\Gamma \vdash t_1\ t_2 \ \sim \ n_1\ (n_2)}$$

$$\frac{\Gamma \vdash t_i \ \sim \ n_i, (\text{pour tout } i \in [1; m])}{\Gamma \vdash C\ (t_1; \ldots; t_m) \ \sim \ C\ (n_1; \ldots; n_m)}$$

$$\frac{\Gamma \vdash t \ \sim \ n \qquad \Gamma \vdash \pi_i \ \sim \ \pi'_i, (\text{pour tout } i \in [0; m])}{\Gamma \vdash \texttt{match}\ t\ \texttt{with}\ \pi_0; \ldots; \pi_m \ \sim \ \texttt{match}\ n\ \texttt{with}\ \pi'_0; \ldots; \pi'_m} \qquad \frac{\texttt{Un}^{(m)}; \Gamma \vdash t \ \sim \ n}{\Gamma \vdash m \rightarrow \ t \ \sim \ m \rightarrow \ n}$$

FIG. 3.2.2 – Relation de décurryfication

3.3.1 Caractérisation des fermetures : les fermetures comme pense-bête

L'analyse statique nous permet lors de la traduction de différencier plusieurs cas d'application. Elle nous permet de caractériser structurellement des fermetures (*i.e.* la structure de leurs composants) apparaissant lors de l'évaluation d'un terme εML tout comme pour le terme nML issu de la traduction. Nous commençons par caractériser les fermetures simples avant de nous intéresser aux fermetures récursives.

Caractérisation des fermetures εML

Intéressons-nous aux règles de sémantique de εML qui sont susceptibles de faire apparaître des fermetures :

$$\frac{e(n) = (t, e_1)}{e \vdash n! \Rightarrow (t, e_1)}\ (1) \qquad \frac{e \vdash t_1 \Rightarrow (\lambda.\ t, e_1) \qquad e \vdash t_2 \Rightarrow v_2 \qquad v_2; e_1 \vdash t \Rightarrow (t, v_2; e_1)}{e \vdash (t_1\ t_2) \Rightarrow (t, v_2; e_1)}\ (2)$$

$$e \vdash (\lambda.\ t) \Rightarrow (t, e)\ \ (3)$$

Utilisons maintenant les informations provenant de l'analyse pour caractériser ces fermetures.

Considérons la règle (1). Supposons que $\Gamma(n) = \texttt{Kn}(m)$, alors il vient $t = \texttt{nabs}\ m\ t'$ et nous avons que dans e, $n!$ s'évalue en $(\texttt{nabs}\ m\ t', e_1)$.

Intéressons nous à l'évaluation d'un terme de la forme $\texttt{napp}\ n!\ (t_0, \ldots, t_k)$ dans l'environnement e. Supposons $\Gamma(n) = \texttt{Kn}(m)$ et $k < m$. Sachant que t_0 dans e s'évalue en v_0, ..., t_{k-1} en v_{k-1}, nous pouvons prévoir que $\texttt{napp}\ n!\ (t_0, \ldots, t_{k-1})$ s'évaluera en $(\texttt{nabs}\ (m-k)\ b, v_{k-1}; \ldots; v_0; e_1)$ (k application de la règle (2)).

Dans les autres cas, nous n'avons pas d'information nous permettant de caractériser les fermetures introduites. Nous notons $e_1@e_2$ la concaténation de la liste e_1 à la liste e_2.

Notons $\mu\texttt{clos}\ m\ t\ e\ v_0; \ldots; v_k = (\texttt{nabs}\ (m-k)\ t, v_0; \ldots; v_k@e)$.

Nous pouvons ainsi construire des règles d'évaluation hybrides : l'évaluation d'un terme t dans un environnement d'évaluation e dont les fermetures sont caractérisées grâce aux informations provenant d'un environnement d'analyse Γ.

$$\frac{e(n) = (t, e_1) \qquad \Gamma(n) = \texttt{Kn}(m) \qquad t = \texttt{nabs}\ m\ t'}{\Gamma, e \vdash n! \Rightarrow (\mu\texttt{clos}\ m\ t'\ e_1\ \varepsilon)} \text{ (eval-var-abs)}$$

$$\frac{\begin{array}{c} e(n) = (\mu\texttt{clos}\ m\ t'\ e_1\ \varepsilon) \\ \Gamma, e \vdash t_0 \Rightarrow v_0 \quad \ldots \quad \Gamma, e \vdash t_k \Rightarrow v_k \quad k < m \end{array}}{\Gamma, e \vdash \texttt{napp}\ n!\ (t_0; \ldots; t_k) \Rightarrow (\mu\texttt{clos}\ m\ t'\ e_1\ v_k; \ldots; v_0)} \text{ (eval-curried-app)}$$

Nous avons donc trois sortes de fermetures qui peuvent apparaître lors de l'évaluation d'un terme εML :

- les fermetures correspondant à une variable liée à une abstraction curryfiée : $(\mu\texttt{clos}\ m\ t'\ e_1\ \varepsilon)$,
- les fermetures correspondant à une application imbriquée partielle, dont l'appelé le plus interne est une variable que l'on sait liée à une abstraction curryfiée : $(\mu\texttt{clos}\ m\ t'\ e_1\ v_k; \ldots; v_0)$,
- les fermetures quelconques pour les autres cas : (t, e).

Caractérisation des fermetures nML

De même nous pouvons caractériser les fermetures issues de la transformation. Commençons par nous intéresser à celles issues de l'évaluation d'une décurryfication d'une fonction n-aire.

$$\frac{e \vdash t \Rightarrow v}{e \vdash \texttt{NCurry}(m)\ t \Rightarrow (0, \underbrace{\lambda^0.\ldots.\lambda^0}_{m}.\ (m+1)!(m!; \ldots; 0!), v; e)} \text{ (eval-NCurry)}$$

En effet, $\texttt{NCurry}(m)\ t = (\lambda^0.\ (\underbrace{\lambda^0.\ \ldots\lambda^0}_{m+1}.\ (m+1)!(m!; \ldots; 0!)))\ [t]$.

Or, $e \vdash \texttt{NCurry}(m) \Rightarrow (0, \underbrace{\lambda^0.\ \ldots\lambda^0}_{m+1}.\ (m+1)!(m!; \ldots, 0!); e)$. En appliquant la règle d'application, on obtient eval-NCurry.

Notons : $(\texttt{nclos}\ m\ clos\ e_0\ args) = (0, \underbrace{\lambda^0.\ \ldots \lambda^0}_{m-|args|}.\ (m+1)!(m!;\ldots;0!), args@(clos; e_0))$.
Grâce à l'analyse statique nous savons un peu plus de choses, notamment que $t = n!$ tel que $\Gamma(n) = \texttt{Kn}(m)$. En effet, le combinateur `NCurry` n'apparaît que dans la traduction des variables. De plus, nous pouvons caractériser l'évaluation d'une application curryfiée de la forme $(\ldots((\texttt{NCurry}(m)\ [n!])\ [t_0])\ldots[t_k])$ où $k < m$ et $\Gamma(n) = \texttt{Kn}(m)$.

Dans un environnement d'évaluation e et un environnement d'analyse Γ, considérons les deux règles hybrides suivantes :

$$\frac{e(n) = (m, t, e_0) \qquad \Gamma(n) = \texttt{Kn}(m)}{\Gamma, e \vdash \texttt{NCurry}(m)\ [n!] \Rightarrow (\texttt{nclos}\ m\ (m, t, e_0)\ e\ \varepsilon)}\ \text{(eval-NCurry-var)}$$

$$\frac{\begin{array}{c} e(n) = (m, t, e_0) \qquad \Gamma(n) = \texttt{Kn}(m) \\ \Gamma, e \vdash t_0 \Rightarrow v_{(k-1)} \qquad \ldots \qquad \Gamma, e \vdash t_{k-1} \Rightarrow v_0 \qquad k < m \end{array}}{\Gamma, e \vdash (\ldots((\texttt{NCurry}(m)\ [n!])\ [t_0])\ldots[t_{k-1}]) \Rightarrow (\texttt{nclos}\ m\ (m, t, e_0)\ e\ v_0; \ldots; v_k)}\ \text{(eval-curried-app)}$$

Tout comme pour les fermetures εML, nous avons trois sortes de fermetures pouvant apparaître lors de l'évaluation d'un terme nML issu de la transformation correspondant aux mêmes conditions :

- les fermetures correspondant à une variable liée à une abstraction curryfiée : $(\texttt{nclos}\ m\ (m, t, e_0)\ e\ \varepsilon)$,
- les fermetures correspondant à une application imbriquée partielle, dont l'appelé le plus interne est une variable que l'on sait liée à une abstraction curryfiée : $(\texttt{nclos}\ m\ (m, t, e_0)\ e_0\ args)$,
- les fermetures quelconques unaires pour les autres cas provenant de la transformation : $(0, t, e)$.

Caractérisation des fermetures récursives :

Tout comme pour les fermetures simples, nous pouvons caractériser les fermetures récursives apparaissant lors de l'évaluation d'un terme εML. La présence de fermetures récursives alliée aux informations provenant de l'environnement d'analyse, nous permet non seulement de caractériser les fermetures récursives mais aussi des fermetures partielles provenant de l'évaluation d'applications imbriquées.

Notons : $(\mu\texttt{closrec}\ m\ t\ e\ v_0; \ldots; v_n) = (\texttt{nabs}\ \ (m-n)\ t, v_0; \ldots; v_n@(\texttt{nabs}\ \ m\ t, e); e)$.
Il en résulte les deux règles hybrides supplémentaires :

$$\frac{e(n) = (t, e_1)_{rec} \qquad \Gamma(n) = \texttt{Kn}(m) \qquad t = \texttt{nabs}\ \ m\ b}{e \vdash n! \Rightarrow (\texttt{nabs}\ \ m\ b, e_1)_{rec}}$$

$$\frac{e(n) = (\texttt{nabs}\ \ m\ b, e_1)_{rec} \qquad e \vdash t_0 \Rightarrow v_{k-1} \qquad \ldots \qquad e \vdash t_k \Rightarrow v_0 \qquad k \leq m}{e \vdash \texttt{napp}\ n!(t_0; \ldots; t_k) \Rightarrow (\mu\texttt{closrec}\ m\ t\ e_1\ v_0; \ldots; v_k)}$$

De même, nous obtenons des caractérisations supplémentaires pour les fermetures récursives de nML .

$$\frac{e(n) = (m, t, e_0)_{rec} \qquad \Gamma(n) = \texttt{Kn}(m)}{e \vdash \texttt{NCurry}\ m\ [n!] \Rightarrow (\texttt{nclos}\ m\ (m, t, e_0)_{rec}\ e\ \varepsilon)}$$

$$\frac{\begin{array}{c} e(n) = (m, t, e_0)_{rec} \qquad \Gamma(n) = \texttt{Kn}(m) \\ e \vdash t_0 \Rightarrow v_{k-1} \qquad \ldots \qquad e \vdash t_{k-1} \Rightarrow v_0 \qquad k \leq m \end{array}}{e \vdash (\ldots((\texttt{NCurry}\ m\ [n!])\ [t_0])\ldots[t_k]) \Rightarrow (\texttt{nclos}\ m\ (m, t, e_0)_{rec}\ e\ v_0; \ldots; v_k)}$$

3.3.2 Correspondance entre les valeurs d'évaluation εML et nML

Les distinctions de fermetures en εML et nML proviennent des mêmes exploitations de l'analyse. Il paraît donc naturel d'exhiber un lien entre elles vis-à-vis de l'analyse. Ce lien se matérialise au travers d'une relation entre les fermetures εML et les fermetures nML paramétrées par les informations contenues dans l'environnement d'analyse.

Cette relation se stratifie en trois relations définies mutuellement :

Correspondance entre les fermetures provenant de l'évaluation d'une variable
Elle définit dans quelles conditions une fermeture εML v est en relation avec une fermeture nML v' selon l'information γ. Ce qui se note : $\gamma \vdash v \sim v'$.

Propagation de cette relation aux environnements d'évaluation
Si $\forall n, \Gamma(n) \vdash e(n) \sim e'(n)$ alors e et e' sont en correspondance par rapport à Γ, ce qui se note $\Gamma \vdash e \sim e'$.

Correspondance entre valeurs issues d'évaluation
Il s'agit d'une relation binaire entre une valeur εML v et une valeur nML v' se notant $v \sim v'$.

Correspondance entre fermetures présentes dans les environnements : Il s'agit des valeurs fermetures sémantiques associées aux variables et qui peuvent donc avoir été décurryfiées. La relation entre de telles valeurs est définie par les quatre règles d'inférence suivantes :

$$\frac{v \sim v'}{\texttt{Un} \vdash v \sim v'} \qquad \frac{[\![t, \texttt{Un}^{(m+1)}\ \Gamma]\!] = u \qquad \Gamma \vdash e \sim e'}{\texttt{Kn}(m) \vdash (\mu\texttt{clos}\ m\ t\ e\ \varepsilon) \sim (m, u, e')}$$

$$\frac{[\![u, (\texttt{Un}; \texttt{Un}; \Gamma)]\!] = m \qquad \Gamma \vdash e \sim e'}{\texttt{Kn}(0) \vdash (u, e)_{rec} \sim (0, m, e')_{rec}} \qquad \frac{[\![u, (\texttt{Un}^{(n+2)}\ (\texttt{Kn}(n+1); \Gamma)]\!] = m \qquad \Gamma \vdash e \sim e'}{\texttt{Kn}(n+1) \vdash (\texttt{nabs}\ \ n\ u, e)_{rec} \sim (n+1, m, e')_{rec}}$$

Elles sont paramétrées par une information de compilation. Dans le cas d'une information ne donnant aucun renseignement, Un, deux valeurs sont en relation si elles sont en relation selon la relation binaire de correspondance entre valeurs d'évaluation que nous définirons dans quelques lignes.

Si par contre, l'information est de la forme $\texttt{Kn}(m)$ alors nous distinguons selon trois possibilités. Si l'information associée à une variable dans l'environnement de compilation est $\texttt{Kn}(m)$ alors la fermeture εML est :

1) une fermeture simple décurryfiable d'arité $m+1$, c'est-à-dire de la forme

$$\mu\texttt{clos}\ m\ t\ e\ \varepsilon$$

correspond à la fermeture nML de même arité, dont l'environnement est en correspondance avec l'environnement de la fermeture εML par rapport à un environnement Γ. Enfin, son corps est la décurryfication de t dans l'environnement Γ auquel on a ajouté en tête $m+1$ `Un` pour les paramètres formels.

2) une fermeture récursive d'arité 1, ie $m = 0$. La fermeture εML

$$(u, e)_{rec},$$

correspond à la fermeture récursive d'arité 1,

$$(0, m, e')_{rec}$$

où son environnement est en correspondance avec l'environnement de la fermeture source, par rapport à l'environnement de compilation Γ. De plus, son corps est la décurryfication du corps de la fermeture source dans l'environnement Γ, en tête duquel on a indiqué que l'on ne connaissait rien concernant la variable de récursion et le paramètre formel.

3) une fermeture récursive d'arité supérieure à 1, ie. $m = n + 1$, donc de la forme :

$$(\texttt{nabs}\ \ n\ u, e)_{rec}$$

correspond à la fermeture récursive nML d'arité $n+2$, dont l'environnement est en correspondance avec l'environnement de la fermeture source par rapport à un environnement de compilation Γ. De plus, son corps est la décurryfication du corps de la fermeture d'origine dans Γ, auquel on a ajouté en tête $n+2$ `Un` pour les paramètres formels précédant l'information $\texttt{Kn}(n+1)$ en position $n+2$ pour la variable de récursion.

La relation de correspondance entre un environnement εML e et un environnement nML e' par rapport à un environnement de compilation Γ est l'itération de la relation sur les valeurs des variables élément par élément :

$$\varepsilon \vdash \varepsilon \sim \varepsilon \qquad\qquad \frac{\gamma \vdash v \sim v' \qquad \Gamma \vdash e \sim e'}{\gamma; \Gamma \vdash v; e \sim v'; e'}$$

La relation binaire de correspondance entre valeurs fermetures en cours d'évaluation, se définit par un jeu de cinq règles. La première règle traite des fermetures simples que l'on peut rencontrer, par exemple lors de l'évaluation d'une abstraction non liée. La seconde concerne les définitions récursives d'arité 1.

Les trois suivantes traitent les applications partielles. Concernant l'application partielle d'une abstraction décurryfiable simple, la fermeture εML produite est de la forme

$$(\mu\texttt{clos}\ m\ t\ e\ v_0; \ldots; v_n).$$

Il s'agit de l'évaluation de l'abstraction décurryfiable d'arité $m+1$ ayant déjà reçu n arguments dont les valeurs sont $v_n, \ldots, v_0$. Il lui reste donc à recevoir $m+1-n$ arguments. Cette fermeture est en correspondance avec la valeur associée au combinateur de curryfication d'arité $m+1$, auquel on a passé les n valeurs correspondantes aux n arguments passés

à la fermeture d'origine et la fermeture décurryfiée correspondant à l'abstraction d'origine :(m, u, e'). Le corps de cette dernière est la décurryfication de t dans l'environnement de compilation Γ enrichi des informations concernant les paramètres formels :$(\mathtt{Un}^{(m+1)}\ \Gamma)$. De plus, son environnement est en correspondance avec e par rapport à Γ.

Selon le même principe, on obtient la correspondance d'applications partielles, dont l'abstraction mise en jeu est récursive.

Enfin, nous avons besoin d'une règle distincte pour traiter le premier appel partiel à une abstraction décurryfiable récursive.

$$\frac{[\![t, (\mathtt{Un}; \Gamma)]\!] = u \quad \Gamma \vdash e \sim e'}{(t, e) \sim (0, u, e')}\ \text{(clos-simpl)} \qquad \frac{[\![u, (\mathtt{Un}; \mathtt{Un}; \Gamma)]\!] = m \quad \Gamma \vdash e \sim e'}{(u, e)_{rec} \sim (0, m, e')_{rec}}$$

$$\frac{[\![t, (\mathtt{Un}^{(m+1)}\ \Gamma)]\!] = u \quad \Gamma \vdash e \sim e' \quad n \leq m \quad v_i \sim w_i \quad (\text{pour tout } i \in [0; n[)}{(\mu\mathtt{clos}\ m\ t\ e\ v_0; \ldots; v_n) \sim (\mathtt{nclos}\ m\ (m, u, e')\ e'\ w_0; \ldots; w_n)}\ \text{(clos-curryfied)}$$

$$\frac{\begin{array}{c}[\![u, (\mathtt{Un}^{(n+1)}\ (\mathtt{Kn}(n); \Gamma)]\!] = m \\ \Gamma \vdash e \sim e' \quad k \leq n \quad v_i \sim w_i \quad (\text{pour tout } i \in [0; k[)\end{array}}{(\mu\mathtt{closrec}\ n\ u\ e\ v_0; \ldots; v_k) \sim (\mathtt{nclos}\ (n)\ (n, m, e')_{rec}\ e'\ w_0; \ldots; w_k)}$$

$$\frac{[\![u, (\mathtt{Un}^{(n+2)}\ (\mathtt{Kn}(n+1); \Gamma)]\!] = m \quad \Gamma \vdash e \sim e'}{(\mathtt{nabs}\ \ n\ u, e)_{rec} \sim (\mathtt{nclos}\ (n+1)\ (n+1, m, e')_{rec}\ e'\ \varepsilon)}$$

L'ensemble de ces correspondances forme un jeu de règles d'inférence qui mène à trois relations mutuellement inductives non déterministes. En effet, la plupart des fermetures mises en jeu ne sont pas distinguables. Lors de la preuve, c'est le reste des invariants qui permettra de discriminer les cas incohérents.

3.3.3 Preuve de correction sémantique

Grâce aux relations définies ci-dessus, nous pouvons comparer une évaluation εML à une évaluation nML. Cette notion de correspondance de valeur d'évaluation nous permet de comparer l'évaluation d'un programme εML à l'évaluation de sa traduction en nML. Nous avons démontré le théorème de correction suivant :

Théorème 3.3.1 *(Préservation sémantique de la décurryfication)*
Si $[\![t, \varepsilon]\!] = u$ *et*

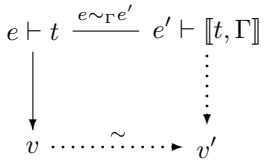
$$\varepsilon \vdash\ t \Rightarrow v,$$

avec alors il existe v' *telle que*

$$\varepsilon \vdash\ u \Rightarrow v' \text{ et } v \sim v'.$$

□

La preuve se fait par induction sur les règles d'évaluation εML. On prouve, pour chaque règle d'évaluation εML, le diagramme de simulation suivant :



Si le terme εML t s'évalue en une valeur v dans l'environnement e et que $e \sim_\Gamma e'$ alors il existe une valeur v', telle que dans l'environnement e', le terme $[\![t, \Gamma]\!]$ s'évalue en v' et $v \sim v'$.

Le cas intéressant de la preuve est celui de l'application (les propriétés présentées comme des résultats ont aussi été démontrées). Nous en détaillons ici la démonstration, on veut :
Si $e \vdash (t_a\ t_b) \Rightarrow v$ *et* $\Gamma \vdash e \sim e'$ *alors* $\exists v', e' \vdash [\![(t_a\ t_b), \Gamma]\!] \Rightarrow v'$ *et* $v \sim v'$.

Par induction, nous avons les trois propriétés suivantes :

si $e \vdash t_a \Rightarrow (t, e_a)$ et $\Gamma \vdash e \sim e'$ alors il existe v'_a telle que :
$e' \vdash [\![t_a, \Gamma]\!] \Rightarrow v'_a$ et $(t, e_a) \sim v'_a$

si $e \vdash t_b \Rightarrow v_b$ et $\Gamma \vdash e \sim e'$ alors il existe v'_b telle que :
$e' \vdash [\![t_b, \Gamma]\!] \Rightarrow v'_b$ et $v_b \sim v'_b$

si $(v_b; e_a) \vdash t \Rightarrow v$ et $\Gamma' \vdash (v_b; e_a) \sim e''$ alors il existe v' telle que :
$e'' \vdash [\![t, \Gamma]\!] \Rightarrow v'$ et $v \sim v'$

De la transformation, nous distinguons deux cas à traiter :

Cas non optimisé : $[\![(t_a\ t_b), \Gamma]\!] = [\![t_a, \Gamma]\!]\ [[\![t_b, \Gamma]\!]]$ et t_a n'est pas une application partielle d'une fonction connue attendant un dernier argument.

Cas optimisé : $[\![(t_a\ t_b), \Gamma]\!] = n!\ [[\![t_0, \Gamma]\!]; \ldots; [\![t_m, \Gamma]\!]; [\![t_b, \Gamma]\!]]$ et $t_a = \texttt{napp}\ n\ (t_0, \ldots, t_m)$ avec $\Gamma(n) = \texttt{Kn}(m+1)$.

I. Cas non optimisé :

En utilisant l'hypothèse d'induction sur l'évaluation de t_a et t_b, il en résulte les deux résultats :

(a) : $e' \vdash [\![t_a, \Gamma]\!] \Rightarrow v'_a$ et $(t, e_a) \sim v'_a$

(b) : $e' \vdash [\![t_b, \Gamma]\!] \Rightarrow v'_b$ et $v_b \sim v'_b$

Or, v est le résultat de l'évaluation $v_b; e_a \vdash t \Rightarrow v$. Pour appliquer le diagramme à cette évaluation, il nous faut identifier v'_a comme étant une fermeture et identifier le corps et l'environnement. Nous devons donc nous intéresser à la forme de la fermeture issue de l'évaluation de t_a. Pour ce faire, nous utilisons le résultat **(a)** et plus particulièrement $(t, e_a) \sim v'_a$. Selon la règle utilisée pour obtenir cette correspondance, il s'offre à nous deux cas :

1) Il s'agit de la règle (clos-simpl). Il en résulte $v'_a = (0, u, e'_a)$ avec $[\![t, (\texttt{Un}; e_{a1})]\!] = u$ et $e_{a1} \vdash e_a \sim e'_a$. Nous pouvons appliquer l'hypothèse d'induction sur l'évaluation de t et nous obtenons ainsi le résultat.

2) Il s'agit de la règle (clos-curryfied). (t, e_a) provient d'une application partielle d'une fermeture curryfiée :
$(t, e_a) = (\mu\texttt{clos}\ m\ b\ e_{a_1}\ \mathit{args})$. Il en résulte

$$v'_a = (\texttt{nclos}\ m\ (m, u, e'_{a_1})\ e'_c\ \mathit{args'})$$

où $e_{a_1} \sim_{\Gamma_{a_1}} e'_{a_I}, |args| \leq m$ et $args \sim_{\mathtt{Un}((m+1))\ \varepsilon} args'$ et $e_{a_1} \sim_{\Gamma_{a_1}} e'_{a_I}$.
Nous allons distinguer selon le nombre de paramètres restant à passer pour avoir une application totale :

a – il reste plus d'un paramètre

Il s'agit d'une application partielle : il en résulte
$v = (\mu\mathtt{clos}\ m\ b\ e_{a_1}\ (v_b; args))$ et
$v' = (\mathtt{nclos}\ m\ (m, u, e'_{a_1})\ e'_c\ (v'_b; args'))$ et le résultat se démontre par une application de la règle d'évaluation de l'application nML et en montrant par construction
$(\mu\mathtt{clos}\ m\ b\ e_{a_1}\ v_b; args) \sim (\mathtt{nclos}\ m\ (m, u, e'_{a_1})\ e'_c\ (v'_b; args'))$.

b – il reste un paramètre Nous avons donc affaire à une application totale. Le résultat de l'évaluation est équivalent à celui de u dans l'environnement $((v'_b; args')@e'_{a_1})$. On obtient cette évaluation en exploitant convenablement l'hypothèse d'induction sur l'évaluation de t. Il en résulte $((v'_b; args')@e'_{a_1}) \vdash u \Rightarrow v'$ et $v \sim v'$. Cependant la traduction de t_a est de la forme :
$(\ldots((NCurry\ m\ n!)\ [\![t_0, \Gamma]\!])\ldots)[\![t_m, \Gamma]\!]$.
Nous avons donc à montrer que $(m+1)!\ [m!; \ldots; 0!]$ s'évalue en v' dans l'environnement
$((v'_b; args'; (m, u, e'_{a_1}))@e'_c)$.
Pour cela nous utilisons le résultat intermédiaire suivant :
Si $|args| = n$ et $(v; args)@me \vdash m \Rightarrow vres$ alors
$(v; args')@((n, m, me); e') \vdash (n+1)![n!; \ldots; 0!] \Rightarrow vres$.
Nous obtenons ainsi le résultat.

II. Cas optimisé :

Nous avons $[\![t_a, \Gamma]\!] = n!\ [\![t_0; \ldots; t_m, \Gamma]\!]$, $t_a = \mathtt{napp}\ n!\ t_0; \ldots; t_m$ et que $\Gamma(n) = \mathtt{Kn}(m)$.
Nous avons $\Gamma \vdash e \sim e'$, il en résulte $e(n) = (\mu\mathtt{clos}\ \ m\ b\ e'\ \varepsilon)$.
De là, nous utilisons la correspondance entre environnements pour en déduire que :
$e'(n) = (m, u, e'')$ et $(\mu\mathtt{clos}\ \ m\ t\ e'\ \varepsilon) \sim_{\mathtt{Kn}(m)} (m, u, e'')$.
Nous avons en hypothèse $e \vdash \mathtt{napp}\ n!\ args \Rightarrow (t', e)$.
Nous utilisons le résultat intermédiaire suivant : Si $e \vdash \mathtt{napp}\ n!\ args \Rightarrow v$ alors il existe vn et $vargs$ tels que :
$e \vdash n! \Rightarrow vn$ et $e \vdash args \Rightarrow vargs$.
Nous obtenons donc $vargs$. En appliquant la règle hybride d'évaluation d'application partielle (eval-curried-app), nous obtenons $(t, e) = (\mu\mathtt{clos}\ |args|\ t\ e'\ (\mathtt{rev}\ vargs))$ si $e(n) = (\mu\mathtt{clos}\ \ |args|\ b\ e'\ \varepsilon)$ et $e \vdash args \Rightarrow vargs$. On note `rev` la fonction qui renvoie la renversée d'une liste.

De plus, nous savons que la transformation de t_a est une imbrication de m applications curryfiées de $n!$ aux éléments de $nargs$: $[\![t_a, \Gamma]\!] = \mathtt{Nnapp}\ (\mathtt{NCurry}(|args|)\ n!)\ nargs$. En

utilisant l'hypothèse d'induction associée à l'évaluation de t_a, nous obtenons :
$e' \vdash [\![t_a, \Gamma]\!] \Rightarrow v'_a$ et $(\mu\texttt{clos}\ |args|\ t\ e'\ (\texttt{rev}\ vargs)) \sim v'_a$.
Nous essayons donc d'en savoir un peu plus sur v'_a. On sait que
$e' \vdash (\texttt{NCurry}(|args|)\ n!) \Rightarrow (\texttt{nclos}\ |args|\ (|args|, u, e'')\ e'\ \varepsilon)$ *via* la règle (eval-NCurry).

De plus, de $e' \vdash \texttt{Nnapp}\ (\texttt{NCurry}(|args|)\ n!)\ nargs \Rightarrow v'_a$, on déduit :
$\exists nvargs, e' \vdash nargs \Rightarrow nvargs$ et $v'_a = (\texttt{nclos}\ |args|\ (|args|, u, e'')\ e'\ (\texttt{rev}\ nvargs))$.
Nous avons donc la correspondance entre fermetures εML et nML suivante :
$(\mu\texttt{clos}\ |args|\ t\ e'\ (\texttt{rev}\ vargs)) \sim (\texttt{nclos}\ |args|\ (|args|, u, e'')\ e'\ (\texttt{rev}\ nvargs))$.

Bien que nous ayons caractérisé les fermetures, nous devons traiter les deux règles conduisant à une correspondance entre fermetures : (clos-simpl) et (clos-curryfied). En effet, $(\mu\texttt{clos}\ \ldots)$ est de la forme (t, e) et $(\texttt{nclos}\ \ldots)$ est de la forme $(0, u, e)$. Nous avons donc deux cas à traiter.

1) Correspondance provenant de (clos-simpl) :

Nous avons donc en hypothèse : $1 - [\![t, \Gamma]\!] = ((m+1)!\ [m!; \ldots; 0!]$ et
$2 - \Gamma \vdash ((\texttt{rev}\ vargs)@e) \sim ((\texttt{rev}\ nvargs)@(|args|, u, e''); e')$.
En utilisant l'hypothèse d'induction sur l'évaluation de t, on obtient :
$(nv2; (\texttt{rev}\ nvargs)@(|args|, u, e''); e') \vdash ((m+1)![m!, \ldots, 0!] \Rightarrow v'$. Or nous avons à montrer : $\exists v', (nv2; (\texttt{rev}\ nvargs)@e') \vdash u \Rightarrow v'$.
Nous avons le résultat suivant :
Si $|args| = n$ et $(nv2; (\texttt{rev}\ nvargs)@(n, u, e''); e') \vdash (m+1)!\ [m!; \ldots; 0!] \Rightarrow v'$ alors
$(nv2; (\texttt{rev}\ nvargs)@e') \vdash u \Rightarrow v'$.
Appliqué ici , nous obtenons le résultat.

2) Correspondance provenant de (clos-curryfied) :

Nous avons en hypothèses : $1 - [\![t, (\texttt{Un}^{(|args|)}\ \Gamma)]\!] = u$,
$2 - \Gamma \vdash e' \sim e''$
et $3 - (\texttt{Un}^{(|args|)}\ \varepsilon) \vdash vargs \sim nvargs$.
Nous en déduisons
$(\texttt{Un}^{(|args|+1)}\ \Gamma) \vdash (v_2; (\texttt{rev}\ vargs)@e') \sim (v'_2; (\texttt{rev}\ nvargs)@e'')$. Enfin, en utilisant l'hypothèse d'induction sur l'évaluation de t, nous obtenons le résultat.

3.4 Autour de la décurryfication

3.4.1 Présentation fonctionnelle de l'algorithme

L'algorithme comme présenté plus haut, n'est pas exactement celui implémenté dans notre chaîne de compilation. En effet, nous avons dû accommoder notre transformation aux restrictions du langage de programmation `Coq`.

Il y a différents moyens en `Coq` de définir des fonctions récursives : induction structurelle, induction bien fondée, mesure .... Dans notre transformation, nous avons choisi de définir

la décurryfication par une fonction récursive structurelle. Or, le cas de la décurryfication d'application, comme défini dans l'algorithme ne répond pas à ce critère. En effet,

$$\begin{aligned}[[(\ldots(n!\ t_0)\ldots)\ t_{m-1})\ t_m, \Gamma]] &= n!\ [[t_0;\ldots;t_m, \Gamma]] \text{ avec } \Gamma(n) = \texttt{Kn}(m+1)\\ [[t_1\ t_2, \Gamma]] &= [[t_1, \Gamma]]\ [[t_2, \Gamma]]\end{aligned}$$

Les appels récursifs à la transformation sur les t_i pour $i \in [0, m]$ ne sont pas des appels récursifs structurels. La conséquence est qu'il n'est pas possible de reconnaître une application décurryfiable avant la transformation. Par contre, il est possible de définir une fonction locale récursive qui peut étudier le sous-terme gauche de l'application. Cette fonction a pour but :

1) de reconnaître les applications imbriquées de la forme $(\ldots(n!\ t_0)\ldots)\ t_{m-1})$ où $n!$ est une variable reliée à une abstraction décurryfiable d'arité $m+1$, tel que l'ajout du sous-terme droit de l'application en cours de transformation rend l'application décurryfiable,

2) Si tel est le cas retourner la liste des arguments transformés.

Ainsi, si le sous-terme gauche de l'application forme une application potentiellement décurryfiable, cette fonction retourne : `nary` n $args$ où n est l'indice de la variable appelant et $args$ la liste décurryfiée des m premiers arguments. Dans tout autre cas, on retourne `unary` et on ignore la liste d'accumulation. Autant d'appels à la transformation inutiles.

La transformation d'une application $(t_1\ t_2)$, dans notre implémentation , cas $t_1\ t_2$ de la fonction `trad` est la suivante :

```
let fix app_curried
   (ce : cenv) (a : term) (args : list Nterm) {struct a}
   : cur_app :=
      match a with
        | TVar n =>
            if var_has_arity ce n (List.length args)
            then (nary n args) else  unary
        | TApply a1 a2 =>
            app_curried ce a1 ((trad a2 ce) ; args)
        | _ =>  unary
     end
   in
   match (app_curried ce t1 nil) with
   | unary =>  (NApply (trad t1 ce) ((trad t2 ce) ;nil))
   | nary n args => (NApply (NVar n) (args++(trad t2 ce) ;nil))
   end
```

3.4.2 Une variante de nML pour la substitution

Nous préparons ici, le terrain pour la preuve de préservation sémantique de la prochaine transformation : mise en style CPS, que nous présenterons au prochain chapitre 4. Pour cela

nous avons besoin d'exprimer la sémantique de nML par substitution (voir le chapitre 2) en préservant la stratégie d'évaluation : évaluation faible, par appel par valeur, avec évaluation des sous-termes de gauche à droite.

La substitution étant difficilement exprimable avec la construction syntaxique `letrec`, nous proposons une variante nML sans `letrec` mais avec des abstractions récursives en utilisant le lieur μ.

Ainsi, la syntaxe se modifie comme suit :

Termes : $t ::= n!$
$\mid \lambda^n.\ t$
$\mid \mu^n.\ t$ abstraction récursive d'arité $(n+1)$
$\mid t\ [t_0; \ldots; t_n]$
$\mid$ `let` t_1 `in` t_2
$\mid C\ (t_1; \ldots; t_n)$
$\mid$ `match` t_1 `with` $\pi_0; \ldots; \pi_n$

Motif : $\pi ::= n \rightarrow t$

L'abstraction récursive $\mu^n.\ t$ est d'arité $(n+1)$ et de corps t. Dans le corps de l'abstraction t la variable de récursion est désignée par $(n+1)!$.

Notre exemple devient :

```
let f = mu(1) (f,x,y) .
       match x with
       | 0 -> y
       | 1(z) -> C(1) (f (x,z))
      end
in
 let z=(C(1) (C(1) C(0))) in
   (\x. f (x, (C(1) (C(0)))) z
```

Comme l'exemple le suggère, et de manière assez naturelle, le passage de nML à sa variante sans `letrec` s'effectue en éliminant les occurrences de la construction syntaxique `letrec` :

$$\texttt{letrec}^n\ t\ \texttt{in}\ t_2,$$

en les remplaçant par :

$$\texttt{let}\ \mu^n.\ t\ \texttt{in}\ t_2.$$

La transformation d'un programme est la propagation de cette élimination à tous ses sous termes.

Afin d'être complet dans nos travaux, nous exhibons une sémantique de cette variante avec environnement pour montrer la préservation sémantique de l'élimination des `letrec`. La sémantique naturelle avec environnement de nML avec μ est identique à celle de nML. Bien entendu, il n'y a pas de règle d'évaluation de la construction `letrec`. Une abstraction récursive : $\mu^n.\ t$ s'évalue en la valeur fermeture récursive d'arité $n+1$, de corps t avec une copie de l'environnement courant :$(n, t, e)_{rec}$.

$$e \vdash \mu^n.\ t \Rightarrow (n, t, e)_{rec}$$

L'élimination du `letrec` préserve la sémantique. Le point central de la preuve réside en la reconstitution de la règle d'évaluation du `letrec` :

$$\frac{(n, t, e)_{rec}; e \vdash t_2 \Rightarrow v}{e \vdash \texttt{letrec}^n\ t\ \texttt{in}\ t_2 \Rightarrow v}$$

par une combinaison de la règle d'évaluation de la liaison locale et de l'abstraction récursive :

$$\frac{e \vdash t_1 \Rightarrow v_1 \qquad v_1; e \vdash t_2 \Rightarrow v}{e \vdash \texttt{let}\ t_1\ \texttt{in}\ t_2 \Rightarrow v}$$

$$e \vdash \mu^n.\ t \Rightarrow (n, t, e)_{rec}$$

En effet,

$$\frac{e \vdash \mu^n\ .t \Rightarrow (n, t, e)_{rec} \qquad (n, t, e)_{rec}; e \vdash t_2 \Rightarrow v}{e \vdash \texttt{let}\ \mu^n.\ t\ \texttt{in}\ t_2 \Rightarrow v}$$

La sémantique à grand pas en appel par valeur par substitution de nML avec μ est définie par le jugement suivant :

$$t \Rightarrow v.$$

Lire : " le terme t s'évalue en le terme v". Une évaluation par substitution ne produit pas de valeur sémantique contrairement aux sémantiques avec environnement, un terme se réduit en terme valeur. Évaluant selon une stratégie faible, ces valeurs sont les abstractions simples et récursives qui s'évaluent en elles-mêmes, et les constructeurs appliqués où les arguments ont été évalués en valeurs (voir le chapitre 2).

Concernant l'évaluation des termes liants, nous avons besoin de définir la substitution. Puisque les abstractions de nML sont $n$aires, nous définissons une *substitution simultanée* (n substitutions d'un coup).

Le formalisme de de Bruijn nous permet de gérer le problème de capture de variables libres de manière arithmétique. Cela a un coût, pour définir la substitution, il nous faut définir l'opération d'actualisation des idices ("lifting") des variables libres.

$$\lambda^n.\ t \Rightarrow \lambda^n.\ t \qquad\qquad \mu^n.\ t \Rightarrow \mu^n.\ t$$

$$\frac{t \Rightarrow \lambda^n.\ t' \qquad t_i \Rightarrow v_i \qquad 0 \leq i \leq n \qquad t'\{v_n, \ldots, v_0\} \Rightarrow v}{t\ [t_0; \ldots; t_n] \Rightarrow v}$$

$$\frac{t \Rightarrow \mu^n.\ t' \qquad t_i \Rightarrow v_i \qquad 0 \leq i \leq n \qquad t'\{v_n, \ldots, v_0, \mu^n.\ t'\} \Rightarrow v}{t\ [t_0; \ldots; t_n] \Rightarrow v} \qquad \frac{t_1 \Rightarrow v_1 \qquad t_2\{v_1\} \Rightarrow v}{(\texttt{let}\ t_1\ \texttt{in}\ t_2) \Rightarrow v}$$

$$\frac{t_i \Rightarrow v_i \qquad 1 \leq i \leq n}{C(t_1; \ldots; t_n) \Rightarrow C(v_1; \ldots; v_n)} \qquad \frac{t \Rightarrow C(v_1, \ldots, v_k) \qquad \pi_c = (k \to t_c) \qquad t_c\{v_k, \ldots, v_1\} \Rightarrow v}{(\texttt{match}\ t\ \texttt{with}\ \pi_0; \ldots; \pi_n) \Rightarrow v}$$

Étant en évaluation faible, les abstractions s'évaluent en elles-mêmes. Un constructeur appliqué $C(t_1;\ldots;t_n)$ s'évalue comme le même constructeur m appliqué aux valeurs d'évaluations des t_i.

Une liaison locale `let` t_1 `in` t_2 s'évalue en v si t_2 dans lequel on a remplacé la variable d'indice 0 par la valeur d'évaluation de t_1, s'évalue en v.

La substitution simultanée prend toute son ampleur lors de l'évaluation d'un filtrage ou d'une application. Dans le cas d'une application $t\ [t_0;\ldots;t_n]$, là où l'on plaçait les valeurs en tête d'environnement, on effectue la substitution simultanément des indices concernés par les valeurs obtenues.

En effet, si l'on observe la règle d'évaluation par environnement de l'application :

$$\frac{e \vdash t \Rightarrow (n, t_b, e_1) \qquad e \vdash t_i \Rightarrow v_i \qquad 0 \leq i \leq n \qquad v_n;\ldots;v_0;e_1 \vdash t_b \Rightarrow v}{e \vdash t\ [t_0;\ldots;t_n]}$$

et celle par substitution :

$$\frac{t \Rightarrow \lambda^n.\ t' \qquad t_i \Rightarrow v_i \qquad 0 \leq i \leq n \qquad t'\{v_n,\ldots,v_0\} \Rightarrow v}{t\ [t_0;\ldots;t_n] \Rightarrow v}$$

On observe que l'ajout de n éléments en tête d'environnement, correspond à une substitution simultanée des n premières variables. Autrement dit, on peut voir l'évaluation avec environnement comme une forme spéciale de substitution explicite [1].

La preuve de correspondance entre sémantique avec environnement et sémantique par substitution repose sur cette même observation.

3.5 Perspective : la décurryfication d'ordre supérieur

Notre décurryfication est une décurryfication de premier ordre, elle ne se mêle pas de décurryfier les abstractions prises en paramètres de fonctions. Considérons la fonction `map2`, identique à la fonction `map2` du module `List` d'OCaml :

```
letrec map2 =  λ f.  λ l.  λ m.
               match (l,m) with
               | nil,nil -> nil
               | (a::p),(b::q) ->f a b :: map2 f p q
in map2 ( λ x.  λ y. x+y) l m
```

En appliquant notre décurryfication de premier ordre, nous obtenons le code suivant :

```
letrec map2 =  λ [f ;l ;m].
               match (l,m) with
               | nil,nil -> nil
               | (a::p),(b::q) ->f a b :: map2 [f ;p ;q]
in map2 [( λ x.  λ y. x+y) ;l ;m]
```

Nous obtenons bien la décurryfication de `map2` et celle de son application, que cela soit dans le corps de l'abstraction ou dans sa portée lexicale.

Cependant, le paramètre `f` de `map2` apparaît clairement comme une fonction à deux arguments, sa décurryfication serait une optimisation. On parle alors de décurryfication d'ordre supérieur. On obtiendrait idéalement,

```
letrec map2 =  λ [f ;l ;m].
               match (l,m) with
               | nil,nil -> nil
               | (a::p),(b::q) ->f [a ;b]::map2 [f ;p ;q]
in map2 [( λ [x ;y]. x+y) ;l ;m]
```

Cependant, on veut garder toute l'expressivité du langage curryfié, notamment l'utilisation d'une fonction curryfiée là où on attend une fonction décurryfiée. Par exemple, on peut appliquer à `map2` une fonction, `map2 [(λx. λy. x+y) ;l ;m].`

Cette situation est symétrique à celle qui utilisait une fonction décurryfiée là où l'on attendait une fonction curryfiée. On utilise alors des combinateurs de curryfication `Ncurry`(n). Pour utiliser une fonction curryfiée là où on attend une fonction décurryfiée, on utilise symétriquement des *combinateurs de décurryfication* : `Uncurry`(n). L'exemple précédent peut alors être décurryfié comme suit :

`map2 [Uncurry`(2)`(λx. λy. x+y) ;l ;m].`

Plus généralement, on peut vouloir utiliser une fonction d'arité différente de celle attendue, lors de la transformation. Afin de résoudre les désaccords d'arités, on utilise une combinaison de combinateurs `Ncurry`(n) et `Uncurry`(n).

La décurryfication d'ordre supérieur semble être un prolongement naturel de la décurryfication de premier ordre. Cependant, son étude a donné lieu à peu de travaux. A notre connaissance, elle n'est implantée dans aucun compilateur pour langage fonctionnel, sa vérification a fait l'objet de travaux menés par Hannan et Hicks [48, 49]. Leur preuve est essentiellement une vérification a posteriori qu'un terme est le décurryfié d'un autre de manière stricte, sans considérer de réajustement d'arité en utilisant les combinateurs de décurryfication et curryfication, seules les fonctions toujours appelées avec tous les arguments sont décurryfiées. Cette relation est guidée par le système de types du langage source (Hindley-Milner polymorphique). Ce système de types est très spécifique au langage source, Leur méthode est difficilement réadaptable à des langages dont le systèmes de type serait plus puissant, comme les assistants de preuve.

Dans des travaux en cours menés avec Xavier Leroy, nous définissons un cadre de spécification générale à la vérification des optimisations de décurryfication d'ordre supérieur.

L'utilisation d'un système de typage léger [118] (dans la littérature anglophone *soft typing*), nous permettrait de décurryfier de manière indépendante au système de types du langage source. Ce système de type se contenterait de décrire les termes du langage cible selon leur arités. Ce système de types dispose d'un type universel $\top$. La fonction `map2` a pour type $((\top, \top \to \top), \top, \top) \to \top$.

Contrairement à la preuve de préservation sémantique présentée dans ce chapitre, la preuve de préservation sémantique de la décurryfication d'ordre supérieur ne peut se contenter de raisonner sur la forme syntaxique des termes et des valeurs fermetures. Cette preuve nécessite un raisonnement sur le comportement propre des arguments par rapport aux arités des termes, ce qui est indépendant de la forme syntaxique. Le raisonnement se ferait donc par une équivalence observationnelle comportementale. De telles relations sont difficilement exprimables en `Coq`, elles ne sont pas strictement récursives.

Une solution élégante serait d'utiliser les relations logiques indexées [5] (dans la littérature anglophone *step-indexed logical relations*). Ces relations ne capturent pas l'équivalence observationnelle entre deux termes dans l'absolu, mais plutôt le fait que deux termes restent indistinguables pendant un certain nombre fini de pas de calcul.

La préservation sémantique de la transformation de décurryfication de premier ordre présentée dans ce chapitre, pourrait être une instantiation de ce cadre de spécification pour optimisation de décurryfication d'ordre supérieur.

4 Transformation CPS

Le style par passage de continuation appelé CPS pour *continuation passing style* (acronyme apparu dans [108]), est un style de programmation dans le $\lambda-$calcul et les langages fonctionnels s'en inspirant. La principale caractéristique du style CPS réside dans le fait qu'une fonction ne retourne jamais directement son résultat, mais le passe en argument à une autre fonction : la *continuation*. Chaque fonction prend en argument supplémentaire une continuation qui a pour rôle de représenter la suite du calcul : un peu comme un contexte dans lequel le résultat du calcul de la fonction sera injecté. Le $\lambda-$terme écrit en style direct

$$\lambda x.x + 1$$

devient en style CPS

$$\lambda x.\ \lambda k.\ k(x + 1).$$

k est le paramètre supplémentaire correspondant à la continuation. Une continuation est une abstraction à un paramètre. Dans le corps de la continuation, le paramètre apparaît comme un trou dans un contexte. Par exemple,

$$((\lambda x.\ x + 1)0) :: l$$

devient en CPS

$$\lambda k.\ ((\lambda x.\ \lambda k'.\ k'(x + 1))0)(\lambda v.\ k(v :: l)).$$

La première abstraction sur k attend la *continuation courante*, celle qui récupérera le résultat du calcul : $k(v :: l)$. Après avoir pris son premier argument 0, l'abstraction récupère son calcul et le renvoie à la continuation abstraite par k'. Ici, il s'agit du second argument $(\lambda v.\ k(v :: l))$ qui récupère le résultat du calcul auquel elle est, elle-même, passée en argument. Dans son corps, ce calcul est abstrait par le paramètre de la continuation v qui est placé en tête de liste l. Enfin, on passe en paramètre à la continuation courante, k, le résultat du calcul $(v :: l)$.

Comme pour les contextes, nous avons besoin d'une *continuation initiale*, celle que l'on applique à un programme en entier. Il s'agit de l'identité $\lambda x.\ x$. Ainsi le programme :

$$(\lambda x.\ x + 1)0$$

devient en CPS :

$$((\lambda x.\ \lambda k.\ k(x + 1))0)(\lambda y.\ y).$$

Utilisation du style CPS Le style CPS, introduit originellement par Plotkin [95] et les transformations de mise en forme CPS d'un programme interviennent dans trois domaines centraux de l'étude des langages de programmation : la sémantique, la programmation et la compilation.

Dans le cadre de la sémantique, le style CPS permet de définir une stratégie d'évaluation sur le $\lambda-$calcul pur. Le style CPS est alors considéré comme un métalangage qui décrit la stratégie d'évaluation grâce, notamment, au caractère "contextuel" des continuations. Si l'on considère le $\lambda-$terme suivant $M(N\ P)$, selon que l'on veuille décrire une stratégie d'évaluation en appel par nom ou en appel par valeur avec évaluation des sous-termes de gauche à droite, la forme CPS obtenue est différente. Dans le cadre de l'application ces deux stratégies (avec évaluation des sous-termes de gauche à droite) se différencient par :

Appel par valeur : Les deux sous-termes sont évalués avant l'évaluation du corps de l'abstraction. Si l'on considère $M(N\ P)$, on commence par évaluer M (en v_1 par exemple), puis on évalue $(N\ P)$. C'est-à-dire, on évalue ensuite N, puis P et leur application. Si $N \Rightarrow v_n$ et $P \Rightarrow v_p$ on évalue $v_n\ v_p$. Supposons que l'on obtient alors v_2, on évalue enfin l'application de $v_1\ v_2$. Ce qui, pour notre exemple donne le terme en style CPS suivant :

$$\lambda k.\ [\![M]\!](\lambda m.\ (\lambda k'.\ [\![N]\!](\lambda m'.\ [\![P]\!](\lambda n'.\ m'n'k')))(\lambda n.\ mnk)).$$

$[\![M]\!]$ est la forme CPS du terme M.

Appel par nom : L'argument n'est pas évalué et est passé tel quel au corps de l'abstraction. Si l'on considère $N\ P$, on évalue N, supposons en v_n, puis on passe le terme P tel quel et enfin on évalue le corps $v_n\ P$. Ce qui, concernant notre exemple, donne en style CPS :

$$\lambda k.\ [\![M]\!](\lambda m.\ m(\lambda k'.\ [\![N]\!](\lambda m'.\ m'[\![P]\!]k')k)).$$

La mise en forme CPS d'un programme peut donc encoder la stratégie d'évaluation directement dans le programme résultant de la transformation. Ce résultat est mis en évidence par Plotkin dès l'origine de CPS, au travers du théorème d'indifférence [95]. En plus de décrire la stratégie d'évaluation directement dans la syntaxe, la mise en forme CPS d'un programme permet d'exhiber une sémantique formelle des structures de contrôle avancées telles que le mécanisme des exceptions, le backtracking (retour en arrière dans le code et donc dans l'évaluation), les coroutines et les opérateurs de contrôle.

Au niveau de la programmation, le style CPS permet d'encoder, dans un langage ne les supportant par forcément, ces mêmes structures de contrôle.

Enfin, au niveau de la compilation, un programme en style CPS se retrouve être dans une configuration tout à fait adaptée à des optimisations concrètes, difficilement réalisables en style direct. Ces optimisations utilisent les propriétés syntaxiques et sémantiques d'un programme en style CPS telles que :

- L'élémentarisation des calculs, il n'y a plus de calcul complexe dans un terme en forme CPS ;
- Le nommage de tous les calculs intermédiaires ;
- La validité systématique, à la compilation, de toutes les $\beta-$réductions du programme.

Le style CPS est utilisé comme langage intermédiaire de nombreux compilateurs optimisant pour langage fonctionnel, tels que Orbit Scheme [62] et Standard ML of New Jersey [2].

Le parallèle entre trou de contexte et une abstraction de continuation suggère la définition de machines abstraites à continuation, en particulier la machine `CEK` [36]. Si l'on

compare une machine à pile, comme la `SECD` [63] à la `CEK` il s'avère que cette dernière est plus adéquate à produire du bytecode pour langage fonctionnel. En effet, tout comme en style CPS, la `CEK` a tous ses appels de fonction en *position terminale*. Ce qui signifie que ces appels ne rendent pas la main à une fonction appelante. La pile d'appel devient inutile tout comme la nécessité d'accumuler le contexte.

L'étude et l'utilisation du style CPS dans le domaine de la compilation sont encore d'actualité comme le signifie Kennedy [60].

Vérifications formelles et enseignement Les preuves de correction de transformations CPS sont nombreuses et cela depuis la présentation originelle [95, 27].

Concernant les vérifications formelles sur machine, plusieurs corrections de transformations CPS préexistent : Minamide et Okuma [86] utilisent l'assistant de preuve `Isabelle/HOL` ; Tian [110] utilisant la syntaxe d'ordre supérieur au travers de Twelf ; et Chlipala [17] en utilisant Coq.

Ces travaux ont rencontré les mêmes difficultés :

– l'$\alpha-$conversion,

– les *réductions administratives*.

Les formalisations sur machine de sémantiques de langages de programmation, de systèmes de types ou bien encore de transformations de programmes présentent sur la plupart des assistants de preuve la même difficulté : la gestion des noms de variables, leurs liaisons et l'$\alpha-$conversion, nous avons discuté de cela dans le chapitre 2. Cette difficulté est d'autant plus importante dans la formalisation de transformation CPS, puisque ces transformations, comme nous le verrons dans la section 4.1 introduisent des nouvelles variables et donc de nouvelles liaisons. De plus, la substitution qui est sensible, par définition, au problème d'$\alpha-$conversion et en particulier au problème des captures de variables libres a un rôle important non seulement dans les preuves de correction de ces transformations, mais aussi dans la définition de leurs sémantiques.

Dans leurs travaux, Minamide et Okuma [86] utilisent des variables sans $\alpha-$conversion pour la transformation CPS de Plotkin (section 4.1.1) et un renommage explicite pour la transformation optimisée de Danvy-Nielsen [27]. Tian [110], dans ses travaux utilise la syntaxe d'ordre supérieur. Chlipala [17] utilise les indices de de Bruijn. Nous avons fait le même choix. Cependant, afin d'avoir moins de réactualisations d'indices au cours de la transformation, nous utilisons deux sortes d'indices de de Bruijn différents : l'une pour les variables sources l'autre pour les variables nouvelles, voir la section 4.1.3.

L'autre difficulté rencontrée concerne les redex administratifs. Lors de la mise en style CPS d'un terme, certaines transformations (section 4.1.1) ajoutent des redex supplémentaires (ne correspondant à aucun redex présent dans le terme d'origine). Ces redex ont la caractéristique d'être toujours dynamiquement valides (en appel par valeur). Ils constituent du code inefficace que l'on peut simplifier (section 4.1.1), d'autres transformations n'en produisent pas (sections 4.1.2 et 4.1.3). De plus, les réductions de ces redex administratifs sont à considérer dans la preuve de correction. Plotkin dans sa preuve originelle [95] utilise une transformation auxiliaire (*colon transformation*) afin de gérer ces réductions administratives. Plotkin utilise une sémantique à petit pas. L'utilisation de sémantiques à grand pas, nous permet de contourner cette difficulté. La preuve de préservation sémantique que nous présentons en section 4.3.2 ne fait pas intervenir de fonction auxiliaire.

Le travail présenté dans ce chapitre a fait l'objet d'une publication d'article [29].

4.1 Transformations CPS

Une mise en forme CPS d'un programme explicite la stratégie d'évaluation. Par cohérence avec le reste de notre chaîne de compilation, nous nous intéressons aux transformations CPS pour l'appel par valeur, avec évaluation des sous-termes de gauche à droite.

4.1.1 Transformation de Plotkin pour l'appel par valeur

Nous commençons par l'une des transformations originelles présentées par Plotkin dans [95] :

$$\begin{aligned} [\![x]\!]_1 &= \lambda k.\ k\ x \\ [\![\lambda x.M]\!]_1 &= \lambda k.\ k\ (\lambda x.\ [\![M]\!]_1) \\ [\![M\ N]\!]_1 &= \lambda k.\ [\![M]\!]_1\ (\lambda m.\ [\![N]\!]_1\ (\lambda n.\ m\ n\ k)) \end{aligned}$$

Cette transformation prend en entrée un terme du $\lambda-$calcul pur en style direct et retourne un terme en style CPS. Les termes renvoyés par cette transformation sont des abstractions sur la continuation courante $\lambda k. \ldots$. Le corps de ces abstractions constitue une injection de résultat du calcul du terme initial dans le reste du calcul, qui est symbolisé par continuation courante k.

Les termes du $\lambda-$calcul peuvent se distinguer selon leur complexité :

Les atomes : il s'agit des termes simples, qui sont soit des valeurs $\lambda x.\ M$ soit des variables x. Ils ne nécessitent pas une élémentarisation puisqu'ils sont déjà des calculs élémentaires. La mise en forme CPS d'un atome est une abstraction sur la continuation courante k et le corps de cette abstraction est l'application de k à l'atome ; un peu comme si on injectait le calcul élémentaire qu'est l'atome directement dans le contexte symbolisé par k.

Les termes complexes : les autres termes qui généralement sont constitués de sous-termes, ici $M\ N$. La transformation est toujours une abstraction sur la continuation courante k. Son corps encode la stratégie d'évaluation en une imbrication d'applications et passe le résultat à k en dernier lieu. Considérons le squelette de la règle d'évaluation de l'application pour l'appel par valeur avec évaluation des sous-termes de gauche à droite :

$$\frac{M \Rightarrow v_m \qquad N \Rightarrow v_n \qquad v_m @ v_n \Rightarrow v}{M\ N \Rightarrow v}$$

Le corps de l'abstraction issu de la transformation de $M\ N$ suit ce squelette. Il commence par la transformation de M.

$$\lambda k.\ [\![M]\!]_1 \ldots .$$

Cette transformation donnera aussi une abstraction sur la continuation qui représentera le reste du calcul après l'évaluation de M. On passe donc le reste de ce calcul sous

forme d'une continuation dans laquelle l'évaluation de M est abstraite et devient le paramètre de cette continuation m. Cette continuation commence par la transformation de N.

$$\lambda k.\ [\![M]\!]_1(\lambda m.\ [\![N]\!]_1 \ldots.$$

En effet, nous sommes en appel par valeur et l'argument est évalué. La transformation de N produit également une abstraction sur le reste du calcul c'est-à-dire l'application des valeurs associées à M et N. On passe donc comme continuation à la transformation de N, la continuation où l'évaluation de N est abstraite par n et le corps est $m\ n\ k$:

$$\lambda k.\ [\![M]\!]_1(\lambda m.\ [\![N]\!]_1(\lambda n.\ m\ n\ k)).$$

Cette transformation CPS a une faiblesse non négligeable : elle introduit des $\beta-$redex ne correspondant à aucun des $\beta-$redex initiaux. On appelle de tels redex des *redex administratifs*.

Si l'on considère la transformation CPS de l'application $(x\ y)$ de la variable x à la variable y, il apparaît quatre redex administratifs (les termes soulignés ci-dessous) :

$$\begin{aligned}
[\![x\ y]\!]_1 &= \lambda k.\ \underline{(\lambda k.\ k\ x)\ (\lambda m.\ (\lambda k.\ k\ y)\ (\lambda n.\ m\ n\ k))} \\
&\xrightarrow{\beta} \lambda k.\ \underline{(\lambda m.\ (\lambda k.\ k\ y)\ (\lambda n.\ m\ n\ k))\ x} \\
&\xrightarrow{\beta} \lambda k.\ \underline{(\lambda k.\ k\ y)\ (\lambda n.\ x\ n\ k)} \\
&\xrightarrow{\beta} \lambda k.\ \underline{(\lambda n.\ x\ n\ k)\ y} \\
&\xrightarrow{\beta} \lambda k.\ x\ y\ k
\end{aligned}$$

Considérant la transformation CPS dans le cadre de la compilation, l'introduction de redex administratifs est contradictoire avec la mise en place d'optimisations. Il est donc nécessaire d'éliminer cette inefficacité par une transformation ultérieure.

Dans la variante de l'algorithme de Plotkin suivante, certaines réductions administratives sont éliminées lors de la transformation elle-même. Le principe est de prendre en paramètre supplémentaire la continuation courante k, au lieu de construire une abstraction de cette continuation $(\lambda k. \ldots)$.

$$\begin{aligned}
[\![x]\!]_2 \rhd k &= k\ x \\
[\![\lambda x.M]\!]_2 \rhd k &= k\ (\lambda x.\lambda k.\ [\![M]\!]_2 \rhd k) \\
[\![M\ N]\!]_2 \rhd k &= [\![M]\!]_2 \rhd (\lambda m.\ [\![N]\!]_2 \rhd \lambda n.\ m\ n\ k)
\end{aligned}$$

La transformation de la variable x devient son application à la continuation prise en paramètre supplémentaire : $k\ x$. De même, la transformation de l'abstraction est appliquée à k. Concernant l'application $M\ N$, les continuations correspondant à l'encodage de la stratégie d'évaluation sont passées en paramètres au lieu d'être appliquées.

Nous avons maintenant comme transformation CPS de l'application de la variable x à la variable y le terme suivant :

$$[\![x\ y]\!]_2 \rhd k = (\lambda m.\ (\lambda n.\ m\ n\ k)\ y)\ x,$$

dans lequel il reste encore deux redex administratifs. Les redex administratifs au niveau des variables et des abstractions ont disparu, seuls ceux générés par la transformation des applications persistent.

4.1.2 Transformation optimisante de Danvy et Nielsen

Danvy et Nielsen [27] proposent une transformation ne produisant aucun redex administratif. Pour cela, les *atomes*, variables et valeurs :

$$A, B ::= x \mid \lambda\ x.\ M$$

sont distingués des autres termes :

$$P, Q ::= M_1\ M_2.$$

La transformation devient alors la composée de deux fonctions mutuellement récursives :

Transformation auxiliaire des atomes : La fonction $\Psi_3(A)$ pré-traite la mise en style CPS des atomes :

$$\begin{aligned} \Psi_3(x) &= x \\ \Psi_3(\lambda x.M) &= \lambda x.\lambda k.\ [\![M]\!]_3 \rhd k \end{aligned}$$

Transformation des termes : Comme précédemment, la fonction $[\![P]\!]_3 \rhd k$ prend en paramètre une continuation k. Dans le cas où P est un atome, elle retourne l'application de k à la transformation auxiliaire de l'atome :

$$[\![A]\!]_3 \rhd k\ =\ k\ \Psi_3(A).$$

Dans le cas de l'application, elle discrimine sur les deux sous-termes afin de traiter différemment selon la présence d'atomes :

$$\begin{aligned} [\![A]\!]_3 \rhd k &= k\ \Psi_3(A) \\ [\![A\ B]\!]_3 \rhd k &= \Psi_3(A)\ \Psi_3(B)\ k \\ [\![P\ B]\!]_3 \rhd k &= [\![P]\!]_3 \rhd \lambda p.\ p\ \Psi_3(B)\ k \\ [\![A\ Q]\!]_3 \rhd k &= [\![Q]\!]_3 \rhd \lambda q.\ \Psi_3(A)\ q\ k \\ [\![P\ Q]\!]_3 \rhd k &= [\![P]\!]_3 \rhd \lambda p.\ [\![Q]\!]_3 \rhd \lambda q.\ p\ q\ k \end{aligned}$$

Nous obtenons alors une mise en forme CPS sans redex administratifs. Par exemple, $[\![x\ y]\!]_3 \rhd k = x\ y\ k$.

4.1.3 Transformation CPS proposée

Généraliser cette discrimination aux applications $n-$aires et aux constructeurs appliqués, conduirait à une disjonction de cas difficilement gérable. Le traitement de l'application requiert une discrimination sur les combinaisons atomes - non atomes possibles. Cette étude de cas, nous conduisant à la considération de 4 cas dans le $\lambda-$calcul pur. Nous proposons une alternative pour éviter cette explosion combinatoire qui réduit, lors de la transformation, les $\beta-$redex produits par la transformation de la forme : $(\lambda\ x.\ M)\ A$. Pour cela, nous définissons *l'application intelligente*, $@_\beta$ définie telle que :

$$(\lambda x.M)\ @_\beta\ A = M\{x \leftarrow A\} \qquad\qquad M\ @_\beta\ N = M\ N \text{ sinon}$$

Les applications produites par la transformation lors du traitement des variables et des abstractions sont alors remplacées par des applications intelligentes :

$$\begin{aligned}
[\![x]\!]_4 \rhd k &= k \mathbin{@_\beta} x \\
[\![\lambda x.M]\!]_4 \rhd k &= k \mathbin{@_\beta} (\lambda x.\lambda k.\ [\![M]\!]_4 \rhd k) \\
[\![M\ N]\!]_4 \rhd k &= [\![M]\!]_4 \rhd \lambda m.\ [\![N]\!]_4 \rhd \lambda n.\ m\ n\ k
\end{aligned}$$

Nous avons donc un seul traitement de l'application, à partir duquel nous retrouvons les quatre cas de la transformation de Danvy et Nielsen [27] :

$$\begin{aligned}
[\![x\ y]\!]_4 \rhd &= [\![x]\!]_4 \rhd \lambda m.\ [\![y]\!]_4 \rhd \lambda n.\ m\ n\ k \\
&= (\lambda m.\ [\![y]\!]_4 \rhd \lambda n.\ m\ n\ k) \mathbin{@_\beta} x \\
&= [\![y]\!]_4 \rhd \lambda n.\ x\ n\ k \\
&= \lambda n.\ x\ n\ k \mathbin{@_\beta} y \\
&= x\ y\ k
\end{aligned}$$

$$\begin{aligned}
[\![M\ y]\!]_4 \rhd &= [\![M]\!]_4 \rhd \lambda m.\ [\![y]\!]_4 \rhd \lambda n.\ m\ n\ k \\
&= [\![M]\!]_4 \rhd \lambda m.\ \lambda n\ m\ n\ k \mathbin{@_\beta} y \\
&= [\![M]\!]_4 \rhd \lambda m.\ m\ y\ k
\end{aligned}$$

$$\begin{aligned}
[\![x\ N]\!]_4 \rhd &= [\![x]\!]_4 \rhd \lambda m.\ [\![N]\!]_4 \rhd \lambda n.\ m\ n\ k \\
&= \lambda m.\ [\![N]\!]_4 \rhd \lambda n.\ m\ n\ k \mathbin{@_\beta} x \\
&= [\![N]\!]_4 \rhd \lambda n.\ x\ n\ k
\end{aligned}$$

$$[\![M\ N]\!]_4 \rhd = [\![M]\!]_4 \rhd \lambda m.\ [\![N]\!]_4 \rhd \lambda n.\ m\ n\ k$$

Pour plus de clarté, nous avons choisi des variables pour les atomes, mais le résultat est identique pour les abstractions. Notons toutefois, que ces substitutions au vol effectuent des $\beta-$réductions sous les λ (dans le cas $M\ A$ en particulier). Tout comme la transformation de Danvy et Nielsen, cette transformation produit des termes en style CPS sans redex administratifs.

4.1.4 Transformation proposée avec double indiçage à la De Bruijn

Tout comme Minamide et Okuma [86], nous allons être, une fois de plus, confrontés au problème des variables et de leurs liaisons. Nous avons déjà fait le choix d'utiliser le formalisme à la de Bruijn pour contourner ces difficultés. Cependant, la transformation que nous proposons, ainsi que toutes les autres transformations CPS, ajoutent des variables nouvelles. Dans la transformation de Plotkin, un jeu de trois nouvelles variables suffit : k, m et n. Dans le formalisme à la de Bruijn, il faudrait que pour chaque abstraction ajoutée, correspondant à une continuation, les indices de toutes les variables soient recalculés, comme Minamide et Okuma l'ont fait remarquer. La transformation naïve à la Plotkin de $x_i\ x_j$ serait :

$$\lambda\ .(\lambda\ .x_0\ (x_{(i+2)})\ (\lambda\ .\ (x_0\ x_{(j+3)})\ (\lambda\ .x_1\ (x_2\ x_0)))).$$

Les opérations de réactualisation d'indices sur les variables sources compliquent l'écriture de la transformation et le raisonnement autour de cette transformation.

Partant du constat que les variables insérées par la transformation sont caractérisables : *variables de continuation* k et *paramètres formels de continuation* m et n, nous proposons de distinguer entre :

les variables sources, présentes dans le programme d'origine, x_n et

les variables introduites par la transformation, dites variables de continuation κ_n.

Bien entendu, chaque sorte de variables est introduite par un lieur différent. Les variables de continuation sont liées par le lieur λ_κ, tandis que les variables sources sont liées par un λ. L'exemple précédent devient alors :

$$\lambda_\kappa\ .(\lambda_\kappa\ .\kappa_0\ (x_i)\ (\lambda_\kappa\ .(\kappa_0\ x_j)\ (\lambda_\kappa\ .\kappa_1\ (\kappa_2\ \kappa_0)))).$$

Utilisant cette technique, les indices des variables sources ne varient pas lors de la transformation. De plus, la transformation connaissant par construction les portées lexicales et le nombre des variables qu'elle introduit, les opérations de réactualisation des indices sont restreintes et localisées. La transformation devient alors :

$$\begin{aligned}
[\![x_n]\!]_4 \rhd k &= k\ @_\beta\ x_n \\
[\![\lambda.M]\!]_4 \rhd k &= k\ @_\beta\ (\lambda.\lambda_\kappa.\ [\![M]\!]_4 \rhd k) \\
[\![M\ N]\!]_4 \rhd k &= [\![M]\!]_4 \rhd \lambda_\kappa.\ [\![N]\!]_4 \rhd \lambda_\kappa.\ m\ n\ (\ {}_\kappa\Uparrow_1\ \ {}_\kappa\Uparrow_1\ k)
\end{aligned}$$

L'opérateur ${}_\kappa\Uparrow_n M$ réactualise de n toutes les variables de continuation libre dans M. Autrement dit, si κ_j est libre dans M, elle devient κ_{j+m}. Ici, on effectue deux réactualisations de 1 consécutifs. Chaque réactualisation correspond à un λ_κ ajouté par la transformation.

4.2 CPS comme langage intermédiaire

Dans le cadre du développement du compilateur pour εML certifié, nous proposons une transformation CPS sur nML avec μ vers un langage intermédiaire CPS. Cette transformation est originale par rapport aux autres transformations pour deux raisons. D'une part, parce qu'elle implémente l'algorithme optimisant que nous proposons plus haut :

- aucune production de redex administratifs ;
- utilisation de l'application intelligente afin d'éviter une explosion combinatoire ;
- utilisation de deux sortes d'indices de de Bruijn afin d'éviter les réactualisations d'indices des variables sources.

D'autre part, elle s'étend à un langage plus riche que le $\lambda-$calcul pur, nML en entier. Autrement dit, nous traitons les abstractions et applications $n-$aires, la récursivité et les types concrets.

Nous présentons le langage intermédiaire CPS par sa syntaxe et sa sémantique. Puis nous présentons deux transformations CPS. La première est une adaptation à notre langage de la transformation de Plotkin ; la deuxième est une adaptation de notre algorithme proposé. Seule la dernière intervient lors de la compilation, puisqu'elle produit des termes sans redex administratifs. La première transformation nous est utile dans la preuve.

4.2.1 Syntaxe

La syntaxe du langage intermédiaire CPS est décrite par la grammaire suivante :

termes :	$t ::= x_n$		variable normale
	$\mid \kappa_n$		variable de continuation
	$\mid \lambda^n.\ t$		
	$\mid \mu^n.\ t$		
	$\mid t\ [t_0; \ldots; t_n]$		
	$\mid$ `let` t_1 `in` t_2		lie x_0 à t_1 dans t_2
	$\mid C\ (t_1; \ldots; t_n)$		
	$\mid$ `match` t `with` $\pi_0; \ldots; \pi_n$		
Motifs :	$\pi ::= n \to t$		

Le langage CPS est aussi expressif que nML avec μ. Cependant, il porte syntaxiquement les traits du style CPS :

Deux sortes de variables : CPS comporte deux sortes de variables indicées indépendamment :

- **Les variables normales :** x_n est une variable normale d'indice n, elle correspond à une variable présente avant la transformation (variable source).
- **Les variables de continuation :** κ_n est une variable de continuation, elle a été introduite lors de la transformation.

Paramètre de continuation supplémentaire : les abstractions, simples et récursives, prennent un paramètre formel supplémentaire, leur premier paramètre. Ce paramètre représente la considération de la continuation à laquelle l'abstraction retournera son résultat. Ce paramètre est introduit par la transformation, il s'agit d'une variable de continuation κ_0. Les autres paramètres formels d'abstractions sont ceux présents dans le terme source, ce sont donc des variables normales. Par conséquent :

- **Une abstraction simple :** $\lambda^n.\ t$ a pour corps le terme t et est d'arité $n+1$. Le premier paramètre formel de cette abstraction est le paramètre formel de continuation : κ_0, les n autres paramètres formels sont ceux présents originellement $x_{n-1} \ldots x_0$.
- **Une abstraction récursive :** $\mu^n.\ t$ lie les paramètres formels comme dans le cas d'une abstraction simple, et lie l'abstraction elle-même au paramètre x_n.

Les autres termes : ils sont identiques à ceux de nML. Les motifs de clause et les liaisons locales `let` lient des variables normales.

Nous avons vu dans les transformations CPS présentées plus haut que la distinction des termes atomes par rapport aux autres termes joue un rôle important. Un atome est un terme qui correspond à un calcul simple. Un atome est soit une variable soit une abstraction simple ou récursive, soit un constructeur appliqué à des atomes.

$$A ::= \kappa_i \mid x_i \mid \lambda^n.\ M \mid \mu^n.\ M \mid C(A_0, \ldots, A_n).$$

La double substitution simultanée

La sémantique de CPS sera donnée à grand pas, selon une stratégie faible, en appel par valeur et par substitution. Tout comme pour nML, le caractère $n-$aire des abstractions et la présence de constructeurs appliqués à plusieurs arguments, nous poussent à adopter un mode de substitution simultané (substitutions de plusieurs variables à la fois). De plus, chaque application donne lieu à une substitution d'une variable de continuation κ_0 et une substitution simultanée du reste des arguments. À la sortie de la mise en forme CPS (la traduction de nML vers CPS), les termes seront retranscrits en nML. Cette retranscription aplatit les deux sortes de variables en une seule. Le paramètre de continuation de chaque abstraction CPS deviendra un paramètre normal. Lors de l'évaluation d'un appel, la continuation sera donc passée en argument simultanément aux autres arguments. Par cohérence avec ce fait, nous définissons *la double substitution simultanée* pour la sémantique du langage CPS. On notera par :

$$M\{\vec{N}_n\}\{\vec{P}_p\},$$

la substitution doublement simultanée dans le terme M des variables de continuation $\kappa_0, \ldots, \kappa_n$ par les termes $\vec{N}_n$ et des variables normales $x_0, \ldots, x_p$ par les termes $\vec{P}_p$.

Comme nous avons vu pour nML, la définition de la substitution dans un formalisme à la de Bruijn induit la définition de l'opération de réactualisation des indices (lift), afin d'éviter de capturer des variables libres.

L'opérateur $\Uparrow$ dénote cette réactualisation d'indice de de Bruijn : ${}_x \Uparrow_n M$ remplace tous les x_i libres dans M par x_{i+n}. De même, ${}_\kappa \Uparrow_n M'$ remplace tous les κ_i libres dans M par κ_{i+n}.

4.2.2 Sémantique naturelle par substitution

Nous sommes maintenant en mesure de définir la sémantique naturelle par appel par valeur de CPS par substitution. Un jugement d'évaluation CPS est de la forme :

$$t \Rightarrow v$$

se lisant, le terme t s'évalue en le terme v. Il n'y a pas de valeur sémantique, les termes "valeurs" vers lesquels les termes s'évaluent sont soit des abstractions simples ou récursives, soit des constructeurs appliqués dont les arguments sont eux mêmes des termes valeurs.

$$\lambda^n.\ t \Rightarrow \lambda^n.\ t \qquad \mu^n.\ t \Rightarrow \mu^n.\ t$$

$$\frac{t \Rightarrow \lambda^n.\ t' \quad t_i \Rightarrow v_i \text{ pour tout } i \in [0.n] \quad t'\{v_0\}\{v_n, \ldots, v_1\} \Rightarrow v}{t\ [t_0; \ldots; t_n] \Rightarrow v}$$

$$\frac{t \Rightarrow \mu^n.\ t' \quad t_i \Rightarrow v_i \text{ pour tout } i \in [0.n]] \quad t'\{v_0\}\{v_n, \ldots, v_1, \mu^n.\ t'\} \Rightarrow v}{t\ [t_0; \ldots; t_n] \Rightarrow v}$$

$$\frac{t_1 \Rightarrow v_1 \quad t_2\{\,\}\{v_1\} \Rightarrow v}{let\ \texttt{t}_1\ \texttt{in}\ \texttt{t}_2 \Rightarrow \texttt{v}} \qquad \frac{t_i \Rightarrow v_i \text{ pour tout } i \in [1; n]}{C(t_1; \ldots; t_n) \Rightarrow C(v_1; \ldots; v_n)}$$

$$\frac{t \Rightarrow C(v_1, \ldots, v_k) \quad \pi_i = (\ |\ k \rightarrow t_i) \quad t_i\{\,\}\{v_k, \ldots, v_1\} \Rightarrow v}{match\ \texttt{t with}\ \pi_0; \ldots; \pi_n \Rightarrow \texttt{v}}$$

Les abstractions qu'elles soient simples ou récursives s'évaluent en elles-mêmes. Un constructeur appliqué $C(t_1;\ldots;t_n)$ s'évalue comme le même constructeur appliqué aux évaluation des t_i.

L'évaluation d'une application $t[t_0;\ldots;t_n]$ se définit par deux règles, selon que t s'évalue en une abstraction simple $\lambda^n.\ t'$ ou récursive $\mu^n.\ t'$. Dans le cas d'une application simple, si les t_i s'évaluent en v_i, alors l'application s'évalue comme le corps de l'abstraction t', dans lequel le paramètre de continuation κ_0 est substitué par v_0 et les paramètres formels suivants $x_0,\ldots,x_{n-1}$ sont substitués par les $v_1,\ldots,v_n$. Si $t'\{v_0\}\{v_1,\ldots,v_n\} \Rightarrow v$ alors $t\ [t_0;\ldots;t_n]$ s'évalue en v.

Concernant un appel à une fermeture récursive, il y a en plus, la substitution simultanée de la variable de récursion désignée par x_n par la fermeture elle-même.

L'évaluation des autres termes est similaire à celle dans nML.

4.2.3 Propriétés de la substitution doublement simultanée

Décrivant le comportement du langage CPS par sa sémantique à grand pas par substitution, lors de la preuve, nous aurons besoin de résultat intermédiaire concernant la substitution et les opérations de réactualisation d'indices. Notamment, le lemme de substitution est utile à la preuve de préservation de notre variante de la transformation de Plotkin.

Nous avons choisi de définir nos substitutions et réactualisation d'indices à la manière de Vouillon [7], c'est-à-dire d'une manière plus adaptée à la preuve par induction syntaxique. En effet, afin de garantir la non capture de variable libre, les indices du substituant sont actualisés à chaque passage par un lieur lors de la substitution au lieu d'actualiser uniquement en lieu et position de la substitution.

Ce mécanisme nous est précieux dans la preuve. Nous présentons ici quelques uns des résultats intermédiaires concernant les substitutions et réactualisations d'indices nécessaires à la preuve.

Un terme CPS est $\kappa-$clos, si toutes ses variables de continuation sont liées. Afin de définir, la propriété d'être $\kappa-$clos pour un terme, nous définissons la propriété d'être $\kappa-$valide pour une profondeur dans un terme.

Définition 4.2.1 ($\kappa-$valide)
Un terme CPS M est $\kappa-$valide pour une profondeur d si tous les indices des variables de continuation libres de M sont plus petites que d.

Définition 4.2.2 ($\kappa-$clos)
Un terme M est $\kappa-$clos s'il est $\kappa-$valide pour la profondeur 0.

La réactualisation d'indices de variables de continuation sur un terme $\kappa-$clos est neutre. En effet, seules les variables libres subissent la réactualisation, or un terme $\kappa-$clos ne contient pas de variables libres.

Lemme 4.2.3 (Neutralité de la réactualisation des indices sur les termes $\kappa-$clos)

Si M est $\kappa-$clos, alors quel que soit n : ${}_\kappa\Uparrow_n M = M$.

De même, la substitution de variable de continuation sur un terme où il n'y a pas de variable de continuation libre ne change pas le terme.

Lemme 4.2.4 (Neutralité de substitution des $\kappa-$variables sur les terme $\kappa-$clos)

Si M est $\kappa-$clos, alors $M\{\vec{N}\}\{\vec{P}\} = M\{\}\{\vec{P}\}$.

Propriétés des réactualisation d'indices : Les propriétés sur les réactualisations d'indices sont importantes dans la preuve du lemme de substitution. Tout d'abord, les deux sortes de réactualisation d'indice commutent :

Lemme 4.2.5 (Commutation de ${}_\kappa\Uparrow$ et ${}_x\Uparrow$)

$$ {}_\kappa\Uparrow_n \ {}_x\Uparrow_m M \ = \ {}_x\Uparrow_m \ {}_\kappa\Uparrow_n M. $$

Ensuite, deux réactualisations consécutives de variables normales se composent :

Lemme 4.2.6 (composition de ${}_x\Uparrow$)

$$ {}_x\Uparrow_{m_1} \ {}_x\Uparrow_{m_2} M = \ {}_x\Uparrow_{m_1+m_2} M. $$

Ils s'ajoutent à ces résultats quelques lemmes de recombinaison des réactualisations d'indices utiles ponctuellement dans les preuves.

Réactualisation d'indice et substitution : En jouant sur les réactualisation d'indices, normaux ou de continuation, on peut neutraliser la substitution :

Lemme 4.2.7

$$ {}_\kappa\Uparrow_{|\vec{N}|} \ {}_x\Uparrow_{|\vec{P}|} M\{\vec{N}\}\{\vec{P}\} = M. $$

La réactualisation d'indice de continuation commute avec la substitution :

Lemme 4.2.8

$$ {}_\kappa\Uparrow_n M\{ \ {}_\kappa\Uparrow_n \vec{N}_{(1+i+n)}\}\{ \ {}_\kappa\Uparrow_n \vec{P}_j\} = \ {}_\kappa\Uparrow_n M\{\vec{N}_{(i+n)}\}\{\vec{P}_j\}. $$

De même pour la réactualisation de variables normales commute avec la substitution :

Lemme 4.2.9

$$ {}_x\Uparrow_m M\{ \ {}_x\Uparrow_m \vec{N}_i\}\{ \ {}_x\Uparrow_m \vec{P}_{(j+m)}\} = \ {}_x\Uparrow_m M\{\vec{N}_i\}\{\vec{P}_{(j+m)}\}. $$

Compositions de substitutions : Enfin, les deux principaux résultats utiles directement dans la preuve de préservation sémantique :

Théorème 4.2.10 (Lemme de substitution)

$$ (M\{\vec{N}\}\{\vec{P}\})\{\vec{Q}\}\{\vec{R}\} = (M\{ \ {}_\kappa\Uparrow_{|\vec{N}|} \ {}_x\Uparrow_{|\vec{P}|} \vec{Q}\}\{ \ {}_\kappa\Uparrow_{|\vec{N}|} \ {}_x\Uparrow_{|\vec{P}|} \vec{R}\})\{\vec{N}\}\{\vec{P}\}. $$

Il s'agit de faire commuter deux substitutions doublement simultanées consécutives. Un autre résultat important est de composer deux substitutions doublement simultanées consécutives en une :

Théorème 4.2.11 (Composition de substitutions)

$$(M\{\ _{\kappa}\Uparrow_{|\vec{N}|}\ _{x}\Uparrow_{|\vec{P}|}\vec{Q}\}\{\ _{\kappa}\Uparrow_{|\vec{N}|}\ _{x}\Uparrow_{|\vec{P}|}\vec{R}\})\{\vec{N}\}\{\vec{P}\} = M\{\vec{N},\vec{Q}\}\{\vec{P},\vec{R}\}.$$

4.3 Transformation CPS non optimisée

Nous avons étudié deux mises en forme CPS dans le cadre du développement du front-end pour εML. La première transformation est une adaptation de la transformation originelle de Plotkin (section 4.1.1) pour l'appel par valeur au langage CPS. La deuxième est l'adaptation de la transformation optimisante que nous proposons (section 4.1.3). Seule la transformation optimisante fait partie de la chaîne de compilation. La première transformation n'apparaît pas dans la chaîne de compilation. Cependant, elle joue un rôle important dans la preuve de préservation sémantique de la transformation optimisante.

4.3.1 Algorithme

Nous commençons par décrire la transformation naïve. Il s'agit d'une extension de la transformation de Plotkin pour l'appel par valeur. La transformation se présente sous forme de deux fonctions mutuellement récursives : une fonction auxiliaire Ψ traitant les termes atomiques et la fonction de transformation elle-même $[\![\cdot]\!]$.

La fonction auxiliaire Ψ traite des atomes ($A ::= x_n \mid \lambda^n.\ M \mid \mu^n.\ M \mid CA_1;\ldots;A_n$). Dans le cas d'une variable, elle est traduite par elle-même. Concernant les abstractions, elles prennent un paramètre formel supplémentaire correspondant à la continuation et leur corps est mis en style CPS. Le passage en argument de la continuation courante est aussi assuré : la transformation du corps de l'abstraction initiale est appliquée au paramètre de continuation de l'abstraction obtenue. Concernant les constructeurs appliqués dont les arguments sont tous des atomes, leur transformation est inductive.

$$\begin{aligned}
\Psi(x_n) &= x_n \\
\Psi(\lambda^n.\ M) &= \lambda^{n+1}.\ [\![M]\!]\ [\kappa_0] \\
\Psi(\mu^n.\ M) &= \mu^{n+1}.\ [\![M]\!]\ [\kappa_0] \\
\Psi(C(A_1,\ldots,A_n)) &= C(\Psi(A_1),\ldots,\Psi(A_n))
\end{aligned}$$

La transformation CPS fournit des termes abstrayant sur une continuation : les termes issus de la transformation sont de la forme $\lambda^0.\ \ldots$. Ces termes sont donc prêts à recevoir la continuation courante, qui sera liée au paramètre κ_0. C'est ce qu'illustre la transformation d'un atome :

$$[\![A]\!] = \lambda^0.\ \kappa_0\ [\Psi(A)].$$

La transformation élémentarise les calculs selon une stratégie d'appel par valeur et d'évaluation de gauche à droite. La mise en style CPS d'une liaison locale `let` M `in` N commence par abstraire sur la continuation courante :

$$\lambda^0.\ \ldots$$

Comme lors de l'évaluation, on transforme M en premier lieu. Son résultat sera replacé dans une continuation représentant le reste du calcul. Cette continuation est appliquée à $[\![M]\!]_1$, elle abstrait sur le résultat de M :

$$\lambda^0.[\![M]\!]\ [(\lambda^0.\ldots)]$$

Le reste du calcul consiste à lier le résultat de M localement et évaluer N ensuite. On lie donc κ_0 à x_0 avant de transformer N :

$$\lambda^0.[\![M]\!]\ [(\lambda^0.\texttt{let}\ \kappa_0\ \texttt{in}\ [\![N]\!]\ldots$$

Enfin, on passe le résultat à la continuation courante. Pour ce faire, on applique $[\![N]\!]$ à κ_1. Soit,

$$[\![\texttt{let}\ M\ \texttt{in}\ N]\!] = \lambda^0.\ [\![M]\!]\ [\lambda^0.\ \texttt{let}\ \kappa_0\ \texttt{in}\ [\![N]\!]\ [\kappa_1]].$$

Considérons le terme $\texttt{let}\ (\texttt{let}\ t_0\ \texttt{in}\ t_1)\ \texttt{in}\ t_2$. Sa mise en style CPS, abstrait sur la continuation courante et commence par la transformation $\texttt{let}\ t_0\ \texttt{in}\ t_1$, comme pour l'évaluation du terme. On passe alors la continuation reprenant le reste calcul, l'évaluation de t_1. Dans cette continuation le sous-terme gauche est abstrait :

$$\lambda^0.\ [\![\texttt{let}\ t_0\ \texttt{in}\ t_1]\!]\ [\lambda^0.\ \texttt{let}\ \kappa_0\ \texttt{in}\ [\![t_2]\!]\ [\kappa_1]]$$

On transforme de même la première liaison locale, la mise en style CPS du terme en entier est :

$$\lambda^0.\ (\lambda^0.\ [\![t_0]\!]\ [\lambda^0.\ \texttt{let}\ \kappa_0\ \texttt{in}\ [\![t_1]\!]\ [\kappa_1]])\ [\lambda^0.\ \texttt{let}\ \kappa_0\ \texttt{in}\ [\![t_2]\!]\ [\kappa_1]].$$

De même, lors d'un filtrage, on commence par l'évaluation du sujet avant de pouvoir sélectionner la clause qui sera exécutée. La transformation d'un filtrage
$\texttt{match}\ M\ \texttt{with}\ \pi_1,\ldots,\pi_n$ abstrait sur la continuation courante et commence par la transformation du sujet M qui prend pour continuation une abstraction dont le corps est un filtrage sur la variable de continuation qui représente le sujet M. Les clauses de ce filtrage sont sous forme CPS. La transformation d'une clause $n \to M$ est une clause CPS de même motif, dont l'action est la transformation CPS de l'action d'origine qui prend en argument de continuation κ_1, la continuation courante au filtrage. Soit :

$$[\![n \to M]\!]_\pi = n \to [\![M]\!][\kappa_1].$$

Soit la transformation d'un filtrage :

$$[\![\texttt{match}\ M\ \texttt{with}\ \pi_1,\ldots,\pi_n]\!] = \lambda^0.\ [\![M]\!]\ [\lambda^0.\ \texttt{match}\ \kappa_0\ \texttt{with}\ [\![\pi_1]\!]_\pi,\ldots,[\![\pi_n]\!]_\pi].$$

La transformation explicite constructivement la stratégie d'appel par valeur avec évaluation de gauche à droite notamment en ce qui concerne la transformation des applications $n-$aires et des constructeurs appliqués. Ces deux sortes de termes contiennent une liste de sous-termes, dont les termes s'évalueront de gauche à droite. Prenons l'exemple du constructeur C appliqué à trois arguments $[t_0, t_1, t_2]$. Sa mise en forme CPS commence par abstraire sur la continuation courante, le contexte où il sera replongé :

$$\lambda^0.\ldots.$$

Évaluant sa liste d'arguments de gauche à droite, on commence par la transformation de t_0. Le résultat de l'évaluation de t_0 sera replongé dans une continuation qui représente la suite du calcul : l'évaluation du reste de la liste et l'injection dans le reste du calcul. Cette continuation abstrait sur la valeur de $[\![t_0]\!]$:

$$\lambda^0.\ [\![t_0]\!][\lambda^0.\ \ldots]$$

L'évaluation suivante est celle de t_1 qui sera aussi appliquée à la continuation exprimant le reste du calcul : l'évaluation du reste de la liste et l'injection dans le reste du calcul. La valeur de $[\![t_1]\!]$ est aussi abstraite dans cette continuation :

$$\lambda^0.\ [\![t_0]\!]\ [\lambda^0.\ [\![t_1]\!]\ [\lambda^0.\ \ldots]]$$

De même pour t_2 :

$$\lambda^0.\ [\![t_0]\!]\ [\lambda^0.\ [\![t_1]\!]\ [\lambda^0.\ [\![t_2]\!]\ [\lambda^0.\ \ldots]]]$$

Enfin , la dernière continuation la continuation courante, représentant le contexte dans lequel le constructeur appliqué doit s'injecter, est représentée par la variable de continuation κ_3. En effet, elle est passée sous trois nouvelles abstractions.

De même, les arguments sont représentés par des variables de continuation dans cette dernière continuation : t_0 est représenté par κ_2 ; t_1 par κ_1 et t_2 par κ_0. Enfin, le rôle de cette dernière continuation est de réinjecter le constructeur dont les arguments ont été élémentarisés au reste du calcul : $\kappa_3\ [C\ (\kappa_2, \kappa_1, \kappa_0)]$.

Considérons maintenant l'application $n-$aire suivante, $t\ [t_0, t_1]$. Nous commençons à abstraire sur la continuation courante :

$$\lambda^0.\ldots.$$

La stratégie d'évaluation commençant par évaluer t, nous commençons par transformer t, qui sera abstrait dans la continuation représentant le reste de l'évaluation :

$$\lambda^0.\ [\![t]\!]\ [\lambda^0.\ldots.]$$

Il s'en suit la transformation du premier argument dont le résultat sera abstrait dans une continuation représentant le reste de l'évaluation :

$$\lambda^0.\ [\![t]\!]\ [\lambda^0.\ [\![t_0]\!]\ [\lambda^0.\ldots.]]$$

De même pour le dernier argument :

$$\lambda^0.\ [\![t]\!]\ [\lambda^0.\ [\![t_0]\!]\ [\lambda^0.\ [\![t_1]\!]\ [\lambda^0.\ \ldots]]]$$

Enfin, dans la dernière continuation, la continuation courante à l'application est représentée par la variable de continuation κ_3. Le sous-terme t est représenté par la variable de continuation κ_2, t_0 par κ_1 et t_1 par κ_0. Le passage à la continuation courante est assurée par le fait que l'application prend un argument supplémentaire en tête d'argument. Il vient : $\kappa_2\ [\kappa_3, \kappa_1, \kappa_0]$. Soit :

$$\lambda^0.\ [\![t]\!]\ [\lambda^0.\ [\![t_0]\!]\ [\lambda^0.\ [\![t_1]\!]\ [\lambda^0.\ \kappa_2\ [\kappa_3, \kappa_1, \kappa_0]]]]$$

Nous notons :

$$[\![M_1 \dots M_n \texttt{ then } N']\!] = [\![M_1]\!]\ [\lambda^0. \dots [\![M_n]\!]\ [\lambda^0.\ N'] \dots].$$

Cette construction systématise le chaînage des applications de continuation et de calcul correspondant à l'évaluation en appel par valeur des sous-termes de gauche à droite.

La mise en style CPS d'un constructeur C appliqué aux n arguments se fait alors comme suit :

$$[\![C(N_1, \dots, N_n)]\!] = \lambda^0.\ [\![N_1 \dots N_n \texttt{ then } \kappa_n(C(\kappa_{n-1}, \dots, \kappa_0)]\!].$$

Les transformations des N_i sont séquencées de gauche à droite tel que : N_i apparaît dans la continuation des transformation des N_j avec $j < i$ et dans la continuation de N_k apparaissent les N_j avec $k < j$. Enfin, la dernière continuation réinjecte le résultat du constructeur appliqué,$C(\kappa_{n-1}, \dots, \kappa_0)$, à la continuation courante, κ_n. Dans $C(\kappa_{n-1}, \dots, \kappa_0)$, les arguments sont représentés par les variables de continuation correspondant à la séquence de continuation simulant leur évaluation.

La mise en style CPS d'une application $n-$aire $M\ [N_1, \dots, N_n]$ est la suivante :

$$[\![M\ [N_1, \dots, N_n]]\!] = \lambda^0.\ [\![M.N_1 \dots N_n \texttt{ then } \kappa_n\ [\kappa_{n+1}, \kappa_{n-1}, \dots, \kappa_0]]\!].$$

L'abstraction sur la continuation courante a pour corps la transformation de M à laquelle on passe pour continuation le chaînage des transformations des arguments dont la dernière continuation est l'application de la variable de continuation correspondant à l'appelant κ_n à la liste d'arguments dont le premier est la continuation courante représentant le reste du calcul κ_{n+1} et le reste de la liste est formé par les variables de continuation représentant chaque argument : $\kappa_n\ [\kappa_{n+1}, \kappa_{n-1}, \dots, \kappa_0]$.

La transformation d'un terme nML en terme CPS produisant un terme en style CPS non optimisé est résumé sur la figure 4.3.1.

$$\begin{array}{rcl}
[\![A]\!] & = & \lambda^0.\ \kappa_0\ [\Psi(A)] \\
[\![M\ [N_1, \dots, N_n]]\!] & = & \lambda^0.\ [\![M.N_1 \dots N_n \texttt{ then } \kappa_n\ [\kappa_{n+1}, \kappa_{n-1}, \dots, \kappa_0]]\!] \\
[\![\texttt{let } M \texttt{ in } N]\!] & = & \lambda^0.\ [\![M]\!]\ [\lambda^0.\ \texttt{let } \kappa_0 \texttt{ in } [\![N]\!]\ [\kappa_1]] \\
[\![C(N_1, \dots, N_n)]\!] & = & \lambda^0.\ [\![N_1 \dots N_n \texttt{ then } \kappa_n(C(\kappa_{n-1}, \dots, \kappa_0)]\!] \\
 & & \text{si } C\ (N_1, \dots, N_n) \text{ n'est pas un atome} \\
[\![\texttt{match } M \texttt{ with } \pi_1, \dots, \pi_n]\!] & = & \lambda^0.\ [\![M]\!]\ [\lambda^0.\ \texttt{match } \kappa_0 \texttt{ with } [\![\pi_1]\!]_\pi, \dots, [\![\pi_n]\!]_\pi] \\
[\![n \to M]\!]_\pi & = & n \to [\![M]\!]\ [\kappa_1]
\end{array}$$

FIG. 4.3.1 – Transformation CPS non optimisée

Considérons le terme $M\ [N, P]$. Notre transformation produit le terme en style CPS correspondant à une stratégie d'appel par valeur avec évaluation des sous-termes de gauche à droite suivant :

$$[\![M\ [N, P]]\!] = \lambda^0.\ [\![M]\!]\ [\lambda^0.\ [\![N]\!]\ [\lambda^0.\ [\![P]\!]\ [\lambda^0.\ \kappa_2\ [\kappa_3, \kappa_1, \kappa_0]]]].$$

Supposons maintenant que M et P soit des atomes, par exemple les variables x et y (notation pour faciliter la lecture), alors :

$$[\![x\ [N, y]]\!] = \lambda^0.\ [\![x]\!]\ [\lambda^0.\ [\![N]\!]\ [\lambda^0.\ [\![y]\!]\ [\lambda^0.\ \kappa_2(\kappa_3, \kappa_1, \kappa_0)]]]].$$

La transformation fait apparaître, alors deux redex non présents au niveau source. Ce sont les redex administratifs qui introduisent de l'inefficacité.

4.3.2 Préservation sémantique

Une manière simple d'énoncer la préservation sémantique de notre transformation non optimisante est la suivante : si le terme nML M s'évalue en une constante, alors $[\![M]\!]$ appliqué à *la continuation initiale* $(\lambda^0.\ \kappa_0)$ s'évalue en la même constante. En effet, le principe du style CPS est de passer le résultat de son calcul au reste du terme qui est représenté par la continuation prise en argument. Dans le langage CPS les continuations sont des abstractions n'attendant qu'un paramètre κ_0 et sont de la forme $\lambda^0.P$. La continuation initiale est la fonction identité : un peu comme le contexte initial, elle retourne le résultat tel quel.

En nML, tout comme en CPS, les valeurs ne sont pas toutes des constantes. Une valeur peut être une abstraction, qui n'est pas préservée exactement par la transformation : son corps est mis en forme CPS. Nous montrons donc la préservation sémantique de cette transformation non optimisée par le théorème suivant :

Théorème 4.3.1 (Préservation sémantique de la transformation non optimisée)

Si $M \Rightarrow v$ *alors* $[\![M]\!]\ [\lambda^0.\ \kappa_0] \Rightarrow \Psi(v)$.

En effet, $\Psi(v)$ a la même structure syntaxique que v.

Ce théorème se prouve par induction sur une dérivation de l'évaluation $M \Rightarrow v$ en nML. Pour cette simulation, il nous faut pouvoir raisonner sur des continuations moins spécifiques que la continuation initiale, notamment sur les continuations générées par la transformation. Par construction, nous savons que les continuations générées par la transformation sont $\kappa-$closes.

Lemme 4.3.2 *Pour tout atome* A, $\Psi(A)$ *est* $\kappa-$*clos et pour tout terme* M, $[\![M]\!]$ *est* $\kappa-$*clos.*

La preuve se fait par induction structurelle sur M et A. Dans le cas des applications $n-$aires et des constructeurs appliqués, nous avons recours aux résultats intermédiaires suivants :

Lemme 4.3.3 κ_i *est libre dans* $[\![M_1, \ldots, M_n$ `then` $N]\!]$ *si* κ_{i+n} *est libre dans* N.

Cette caractéristique nous permet de raffiner notre théorème de simulation en considérant en invariant que les continuations mises en jeu dans la preuve sont $\kappa-$closes.

Nous voulons donc raisonner sur l'évaluation de $[\![M]\!]\ [\lambda^0.\ P]$ telle que $\lambda^0.\ P$ soit $\kappa-$clos, sachant $M \Rightarrow v$. Rappelons que le style CPS se caractérise par le fait que les calculs intermédiaires sont retournés en argument à la continuation prise en argument, la continuation courante. La valeur équivalente de v en CPS est $\Psi(v)$, $[\![M]\!]\ [\lambda^0.\ P]$ se comporte donc comme P dans lequel la variable de continuation κ_0 est substituée par $\Psi(v)$. On remarquera que, contrairement à la preuve de correction de Plotkin [95], nous n'avons pas besoin d'une

fonction auxiliaire pour gérer les réductions administratives, ceci grâce à l'utilisation de sémantiques à grand pas. Notre théorème de simulation se définit comme suit :

Théorème 4.3.4 (Simulation)
Considérons $K = \lambda^0.\ P\ \kappa-clos.$ *Si*

$$M \Rightarrow v \quad et \quad P\{\Psi(v)\}\{\} \Rightarrow v',$$

alors

$$[\![M]\!]\ [K] \Rightarrow v'.$$

Les propriétés et résultats exposés plus haut autour de la substitution sont cruciaux, non seulement parce que l'on raisonne sur une sémantique pas substitution mais aussi à cause des invariants du théorème de simulation.

En plus de ce lemme de substitution, il est important dans la preuve de raisonner sur le devenir par transformation d'une substitution dans le terme source.

Le double indiçage à la de Bruijn, nous permet de conserver les variables sources avec les mêmes indices dans le terme source et le terme traduit. Les variables sources sont préservées à l'identique par la transformation, ce qui nous permet d'avoir une commutation simple entre la transformation et la substitution des variables sources :

Lemme 4.3.5 (Commutation entre substitution et transformation)

$$[\![M\{v_0, \ldots, v_n\}]\!] = [\![M]\!]\{\}\{\Psi(v_0), \ldots, \Psi(v_n)\}.$$

De plus, une conséquence de la $\kappa-$fermeture des transformations (lemme 4.3.2) est la neutralité de substitutions de variables de continuation sur les termes issus de la transformation.

Lemme 4.3.6

$$[\![M]\!]\{\vec{N}\}\{\vec{P}\} = [\![M]\!]\{\ \}\{\vec{P}\}.$$

Afin de donner une idée de la preuve, nous décrivons succinctement un cas de la simulation 4.3.4. Considérons le terme $M = \texttt{let}\ M_1\ \texttt{in}\ M_2$. Nous avons donc en hypothèses :

L'évaluation du terme d'origine : $M_1 \Rightarrow v_1$ et $M_2\{v_1\} \Rightarrow v$.

La mise en place contextuelle : $K = \lambda^0.\ P\ \kappa-$clos et $P\{\Psi(v)\}\{\} \Rightarrow v'$.

Nous voulons conclure :

$$(\lambda^0.\ [\![M_1]\!](\lambda^0.\ \texttt{let}\ \kappa_0\ \texttt{in}\ [\![M_2]\!](\kappa_1)))\ [K] \Rightarrow v' \tag{1}$$

Commençons par appliquer l'hypothèse d'induction sur la seconde prémisse $M_2 v_1 \Rightarrow v$ avec comme continuation K, on obtient :

$$[\![M_2\{v_1\}]\!][K] \Rightarrow v' \tag{2}$$

En utilisant le lemme 4.3.5 et le fait que v_1 est une valeur et donc un atome, (2) devient :

$$([\![M_2]\!]\{\ \}\{\Psi(v_1)\})[K] \Rightarrow v' \tag{3}$$

Prenons maintenant $P_1 = \texttt{let}\ \kappa_0\ \texttt{in}\ [\![M_2]\!](\ _x \Uparrow_1 K)$. En utilisant la règle d'évaluation de la liaison locale, le lemme 4.3.6 et quelques résultats sur la substitution présentés en section 4.2.3, (3) implique :

$$P_1\{\Psi(v_1)\}\{\} \Rightarrow v' \tag{4}$$

La conclusion (1) est obtenue en appliquant l'hypothèse d'induction sur la première prémisse $M_1 \Rightarrow v_1$ et (4) avec pour continuation $K_1 = \lambda^0.P_1$ qui est $\kappa-$clos par le lemme 4.3.2.

4.4 Transformation CPS optimisée

La transformation optimisée utilisée dans le front-end est une adaptation de la transformation que nous proposions plus haut (section 4.1.3) sous forme de traduction de nML vers CPS.

4.4.1 Algorithme

Nous utilisons l'application intelligente $@_\beta$ afin de contourner l'explosion combinatoire que l'analyse de cas peut générer lors, notamment de la transformation des applications n-aires. Au lieu de discriminer sur les combinaisons possibles d'atomes/non atomes sur les sous-termes de l'application, nous utilisons l'application intelligente $@_\beta$ au niveau de la transformation des atomes. L'effet de $@_\beta$ est d'effectuer au vol la substitution d'un atome dans une continuation.

$$\begin{aligned} (\lambda^0.\ M)\ @_\beta\ A &= M\{A\}\{\} \\ M_1\ @_\beta\ M_2 &= M_1\ [M_2] \quad \text{sinon} \end{aligned}$$

Ce mécanisme nous permet d'éliminer les redex administratifs en évitant une explosion combinatoire.

Nous retrouvons l'organisation de la transformation en deux fonctions :

- la fonction auxiliaire qui pré-traite les atomes, notée $\Psi()$;
- la fonction de transformation elle-même. Elle prend un argument supplémentaire, K, qui représente la continuation courante. La transformation du terme M avec pour continuation courante K se note $[\![M]\!] \rhd K$.

Les mécanismes conduisant à l'explicitation de la stratégie d'évaluation en appel par valeur sont identiques à ceux de l'algorithme non optimisé. Par contre, la transformation optimisée ne produit pas d'abstractions sur la continuation comme la transformation optimisée le faisait. Les applications à des continuations de la transformation non optimisée, $\lambda^0.\ [\![M]\!]\ [K]$ deviennent dans la transformation optimisée une transformation dont les deux paramètres sont M et la continuation K.

Contrairement à la transformation non optimisée, les continuations générées par la transformation optimisée ne sont pas $\kappa-$closes. Par conséquent, des opérations de réactualisation d'indices apparaissent explicitement dans la transformation afin d'éviter de capturer des variables libres de la continuation, le paramètre K.

Ainsi, considérons la mise en style CPS d'une liaison locale $\texttt{let}\ M\ \texttt{in}\ N$ dans la continuation courante K. Nous commençons par transformer M en considérant la continuation

dans laquelle son résultat sera replacé $\lambda^0.\ \texttt{let}\ \kappa_0\ \texttt{in}\ B$. B est la transformation de N en considérant la continuation dans laquelle elle replacera son résultat, la continuation courante K. Or, nous avons ajouté deux liaisons : l'abstraction de continuation λ^0 et le `let`. Il faut donc réactualiser les indices dans K en conséquence. Pour cela les indices des variables normales sont incrémentés de 1. Il en est de même pour les indices des variables de continuation. La transformation de la liaison locale devient :

$$[\![\texttt{let}\ M\ \texttt{in}\ N]\!] \rhd K = [\![M]\!] \rhd \lambda^0.\ \texttt{let}\ \kappa_0\ \texttt{in}\ [\![N]\!] \rhd\ {}_{\kappa}\Uparrow_1\ {}_{x}\Uparrow_1 K.$$

La transformation des applications $n-$aires et des constructeurs implique la transcription de la stratégie d'évaluation tout comme dans la transformation non optimisée :

$$[\![M_1, \ldots, M_n\ \texttt{then}\ N]\!] = [\![M_1]\!] \rhd \lambda^0.\ \ldots [\![M_n]\!] \rhd \lambda^0.\ N.$$

Cette construction contient $n+1$ liaisons de variables de continuation. Lors de la transformation d'une application ou d'un constructeur appliqué, les variables de continuation de la continuation courante K doivent être réactualisées en conséquence :

$$[\![M\ [N_1, \ldots, N_n]]\!] \rhd K = [\![M, N_1, \ldots, N_n\ \texttt{then}\ \kappa_n\ [\ {}_{\kappa}\Uparrow_{n+1} K, \kappa_{n-1}, \ldots, \kappa_0]]\!].$$

$$[\![C(N_1, \ldots, N_n)]\!] \rhd K = [\![N_1, \ldots, N_n\ \texttt{then}\ \ {}_{\kappa}\Uparrow_n K(C(\kappa_{n-1}, \ldots, \kappa_0))]\!].$$

De même, lors de la transformation d'une clause de filtrage, les indices des variables de continuation sont réactualisés puisque l'évaluation du sujet du filtrage précède le filtrage et que par conséquent, les clauses du filtrage apparaissent dans le continuation de la transformation du sujet.

La transformation en style CPS optimisante est résumée sur la figure 4.4.1.

Cette transformation ne produit pas de redex administratifs. Les applications intelligentes ont fait disparaître les redex créés par transformation de la forme continuation appliquée à un atome.

Considérons le terme $x\ [N, y]$ où x et y sont des variables alors :

$$\begin{aligned}
[\![x\ [N, y]]\!] \rhd K &= [\![x]\!] \rhd ([\![N, y\ \texttt{then}\ \kappa_2\ [\ {}_{\kappa}\Uparrow_3 K, \kappa_1, \kappa_0]]\!] \\
&= \lambda^0.\ [\![N]\!] \rhd \lambda^0.\ [\![y]\!] \rhd \lambda^0.\ \kappa_2\ [\ {}_{\kappa}\Uparrow_3 K, \kappa_1, \kappa_0]\ @_\beta\ x \\
&= [\![N]\!] \rhd \lambda^0.\ [\![y]\!] \rhd \lambda^0.\ x\ [\ {}_{\kappa}\Uparrow_2 K, \kappa_1, \kappa_0] \\
&= [\![N]\!] \rhd \lambda^0.\ (\lambda^0.\ x\ [\ {}_{\kappa}\Uparrow_2 K, \kappa_1, \kappa_0])\ @_\beta\ y \\
&= [\![N]\!] \rhd \lambda^0.\ x\ [\ {}_{\kappa}\Uparrow_1 K, \kappa_0, y]
\end{aligned}$$

Il n'y a pas de redex administratif, l'application intelligente a permis de les réduire au vol durant la transformation.

4.4.2 Préservation sémantique

Intelligence et conséquences

Comme nous l'avons esquissé plus haut, l'application intelligente conduit à des $\beta-$réductions sous les λ, ce qui n'est pas compatible avec une stratégie d'évaluation faible.

$$
\begin{array}{rcl}
\Psi(x_n) & = & x_n \\
\Psi(\lambda^n.\ M) & = & \lambda^{n+1}.\ [\![M]\!] \rhd \kappa_0 \\
\Psi(\mu^n.\ M) & = & \mu^{n+1}.\ [\![M]\!] \rhd \kappa_0 \\
\Psi(C\ (A_1, \ldots, A_n)) & = & C\ (\Psi(A_1), \ldots, \Psi(A_n)) \\
\\
[\![A]\!] \rhd K & = & K\ @_\beta\ \Psi(A) \\
[\![M\ [N_1, \ldots, N_n]]\!] \rhd K & = & [\![M.N_1 \ldots N_n\ \texttt{then}\ \kappa_n\ [\ {}_\kappa \Uparrow_{n+1} K, \kappa_{n-1}, \ldots, \kappa_0]]\!] \\
[\![\texttt{let}\ M\ \texttt{in}\ N]\!] \rhd K & = & [\![M]\!] \rhd \lambda^0.\ \texttt{let}\ \kappa_0\ \texttt{in}\ [\![N]\!] \rhd\ {}_\kappa \Uparrow_1\ {}_x \Uparrow_1 K \\
[\![C\ (N_1, \ldots, N_n)]\!] \rhd K & = & [\![N_1 \ldots N_n\ \texttt{then}\ {}_\kappa \Uparrow_n K(C\ (\kappa_{n-1}, \ldots, \kappa_0)]\!] \\
 & & \text{si } C(N_1, \ldots, N_n) \text{ n'est pas un atome} \\
[\![\texttt{match}\ M\ \texttt{with}\ \pi_1, \ldots, \pi_n]\!] \rhd K & = & [\![M]\!] \rhd \lambda^0.\ \texttt{match}\ \kappa_0\ \texttt{with}\ [\![\pi_1]\!]_\pi \rhd K, \ldots, [\![\pi_n]\!]_\pi K \\
\\
[\![n \to M]\!]_\pi \rhd K & = & n \to [\![M]\!] \rhd\ {}_\kappa \Uparrow_1 K
\end{array}
$$

FIG. 4.4.1 – Transformation CPS optimisante

En effet, si l'on considère l'application d'un terme complexe M à un atome A, l'application intelligente va effectuer une substitution pour éliminer un redex administratif au niveau de la transformation de A et cette substitution s'effectue sous un λ :

$$[\![M\ [A]]\!] \rhd K = [\![M]\!] \rhd \lambda^0.\ [\![A]\!] \rhd \lambda^0.\ \kappa_1\ [\ {}_\kappa \Uparrow_2 K, \kappa_0]$$

Or,

$$
\begin{array}{rcl}
[\![A]\!] \rhd \lambda^0.\ \kappa_1(\ {}_\kappa \Uparrow_2 K, \kappa_0) & = & (\lambda^0.\ \kappa_1(\ {}_\kappa \Uparrow_2 K, \kappa_0))\ @_\beta\ \Psi(A) \\
 & = & \kappa_1[\ {}_\kappa \Uparrow_1 K, \Psi(A)]
\end{array}
$$

On obtient donc :

$$[\![M\ [A]]\!] \rhd K = [\![M]\!] \rhd \lambda^0.\kappa_1\ [\ {}_\kappa \Uparrow_1 K, \Psi(A)].$$

Par conséquent, la substitution effectuée par $@_\beta$ ne commute pas avec la substitution comme utilisée dans la sémantique de CPS. Considérons le terme :

$$(x_0\ @_\beta\ C)\{x_0 \leftarrow \lambda^0.\ \kappa_0\} = x_0\ [C]\{x_0 \leftarrow \lambda^0.\ \kappa_0\} = (\lambda^0.\ \kappa_0)\ [C]. \quad (A)$$

D'un autre côté, considérons ce second terme :

$$(x_0\{x_0 \leftarrow \lambda^0.\ \kappa_0\})\ @_\beta\ (Cx_0 \leftarrow \lambda^0.\ \kappa_0) = (\lambda^0.\ \kappa_0)\ @_\beta\ C = C. \quad (B)$$

La conséquence directe est que nous ne pouvons obtenir un résultat de préservation sémantique dans le même style que pour la transformation non optimisée. Par contre, les

substitutions effectuées par $@_\beta$ sont des $\beta-$réductions sous les λ. En considérant ce point, nous pouvons conclure non pas sur une évaluation vers une transformation mais vers un terme β_v-équivalent à la transformation. Autrement dit un terme où des $\beta-$réductions internes ont déjà été effectuées.

En effet, une $\beta-$réduction sépare A de B.

Théorème 4.4.1 (Préservation sémantique de la transformation optimisée) *Si*

$$M \Rightarrow v$$

alors il existe une valeur w telle que :

$$[\![M]\!] \rhd \lambda^0.\ \kappa_0 \Rightarrow w \ \textit{et}\ \Psi(v) \overset{*}{\rightsquigarrow} w.$$

La notation $\overset{*}{\rightsquigarrow}$ dénote zéro, une ou plusieurs réductions de $\beta v-$redex dans $\Psi(v)$. Nous formalisons maintenant cette réduction.

β_v-relation entre CPS termes

Partant de l'observation que les substitutions effectuées par $@_\beta$ sont des instances de réductions β_v, nous utilisons le fait que de telles réductions sont valides dans les sémantiques en appel par valeur [95].

Le cas de base pour la β_v-réduction est la contraction d'un $\beta-$redex dont l'argument est un atome :

$$(\lambda^0.\ M)\ [A] \rightsquigarrow M\{A\}\{\} \quad (\beta_v)$$

Nous étendons ensuite la relation $\rightsquigarrow$ par le jeu de règles d'inférence présenté sur la figure 4.4.2. Ces règles d'inférence nous permettent de contracter en parallèle zéro, un ou plusieurs redex à n'importe quelle position dans le terme, y compris sous les λ.

La β_v-relation, $\overset{*}{\rightsquigarrow}$, est la fermeture réflexive et transitive de $\rightsquigarrow$.

Comme dit plus haut, la β_v-réduction est valide dans la sémantique en appel par valeur.

Lemme 4.4.2 (Fermeture de $\rightsquigarrow$ par évaluation) *Supposons*

$$M \Rightarrow v \ \textit{et}\ M \rightsquigarrow M'$$

alors il existe v' tel que :

$$M' \Rightarrow v' \ \textit{et}\ v \rightsquigarrow v'.$$

La preuve se fait par induction sur l'évaluation $M \Rightarrow v$. En observant les règle d'inférence définissant $\rightsquigarrow$, on se rend compte que la plupart des règles préservent les structures syntaxiques. Cela correspond à autant de cas qui se prouvent par simple induction. Le cas intéressant est l'application qui découle du lemme de fermeture de $\rightsquigarrow$ par la double substitution simultanée :

$$\frac{A_1 \rightsquigarrow A_2 \quad A_1 \text{ est un atome} \quad M \rightsquigarrow N}{(\lambda^0.\ M)\ A_1 \rightsquigarrow N\{A_2\}\{\,\}}$$

$$x_i \rightsquigarrow x_i \qquad \kappa_i \rightsquigarrow \kappa_i \qquad \frac{M \rightsquigarrow N}{\lambda^n.\ M \rightsquigarrow \lambda^n.\ N} \qquad \frac{M \rightsquigarrow N}{\mu^n.\ M \rightsquigarrow \mu^n.\ N}$$

$$\frac{M \rightsquigarrow N \quad M_1 \rightsquigarrow N_1 \quad \ldots \quad M_n \rightsquigarrow N_n}{M\ [M_1, \ldots, M_n] \rightsquigarrow N\ [N_1, \ldots, N_n]} \qquad \frac{M_1 \rightsquigarrow N_1 \quad M_2 \rightsquigarrow N_2}{(\ \texttt{let}\ M_1\ \texttt{in}\ M_2) \rightsquigarrow (\ \texttt{let}\ N_1\ \texttt{in}\ N_2)}$$

$$\frac{M_1 \rightsquigarrow N_1 \quad \ldots \quad M_n \rightsquigarrow N_n}{C(M_1, \ldots, M_n) \rightsquigarrow C(N_1, \ldots, N_n)} \qquad \frac{M \rightsquigarrow N \quad \pi_1 \rightsquigarrow \pi'_1 \quad \ldots \quad \pi_n \rightsquigarrow \pi'_n}{(\texttt{match M with}\ \pi_1 \ldots \pi_n) \rightsquigarrow (\ \texttt{match M with}\ \pi'_1 \ldots \pi'_n)}$$

$$\frac{M \rightsquigarrow N}{C^n \to M \rightsquigarrow C^n \to N}$$

FIG. 4.4.2 – β_v-réduction

Lemme 4.4.3 (Fermeture de β_v-relation par la double substitution)
Supposons :

$$M \rightsquigarrow M',\ \vec{N} \rightsquigarrow \vec{N'},\ \vec{P} \rightsquigarrow \vec{P'}$$

alors

$$M\{\vec{N}\}\{\vec{P}\} \rightsquigarrow M'\{\vec{N'}\}\{\vec{P'}\}.$$

Il en découle la fermeture de la β_v-relation sous l'évaluation :

Théorème 4.4.4 (Fermeture de la β_v-relation sous l'évaluation)
Si

$$M \Rightarrow v \text{ et } M \overset{*}{\rightsquigarrow} M'$$

alors il existe v' tel que :

$$M' \Rightarrow v' \text{ et } v \overset{*}{\rightsquigarrow} v'.$$

β_v-relation entre les deux transformations : Les substitutions effectuées au vol par $@_\beta$ correspondent aux réductions des redex administratifs que génère la transformation non optimisée. La relation de β_v-relation capture exactement cette contraction des redex administratifs. La β_v-relation entre les deux transformations est donc un résultat naturel :

Théorème 4.4.5 (β_v-relation entre les deux transformations)
Si $K \overset{*}{\rightsquigarrow} K'$ alors $[\![M]\!][K] \overset{*}{\rightsquigarrow} [\![M]\!] \rhd K'$.

La preuve se fait par induction sur le terme CPS M.

Preuve de préservation sémantique pour la transformation CPS optimisée

La preuve de préservation sémantique pour la transformation optimisée (théorème 4.4.1), n'est pas une preuve par simulation. Elle est la combinaison de trois théorèmes :

I- La preuve de préservation sémantique pour la transformation non optimisée (théorème 4.3.4). On obtient alors :

$$[\![M]\!](\lambda^0.\ \kappa_0) \Rightarrow \Psi(v).$$

II- La β_v-relation entre les deux transformations (théorème 4.4.5) :

$$[\![M]\!](\lambda^0.\ \kappa_0) \overset{*}{\rightsquigarrow} [\![M]\!] \rhd (\lambda^0.\ \kappa_0).$$

III- La fermeture de la β_v-relation sous évaluation qui nous permet de conclure (théorème 4.4.4).

4.5 Autour de la transformation CPS

4.5.1 Intégration dans la chaîne de compilation

Dans notre chaîne de compilation, après notre transformation CPS optimisée, nous effectuons un retour en nML dont le principal rôle est de passer de deux sortes d'indice de de Bruijn à une seule sorte. Cette transformation prend en paramètre, en plus d'un terme CPS, deux environnements de traductions correspondant aux deux sortes de variables : Γ_x associe des indices aux variables x_i et Γ_κ associe des indices aux variables κ_i. La preuve de préservation sémantique de ce retour vers nML se fait en utilisant les sémantiques avec environnements. La sémantique avec environnements de CPS est de la forme :

$$e_x, e_\kappa \vdash M \Rightarrow v$$

où e_x est l'environnement des variables normales et e_κ l'environnement des variables de continuation.

Nous avons eu besoin de prouver la correspondance entre les sémantiques définies par substitution et avec environnements de CPS pour être complets dans notre certification.

4.5.2 Une meilleure exploitation des caractéristiques du style CPS

Notre transformation CPS optimisée traite un sous ensemble de ML plus large que ceux traités dans des travaux antérieurs. De plus, elle s'exerce sur une sémantique naturelle et utilise un double indiçage à la de Bruijn afin d'exclure les problèmes liés aux noms et à la substitution. Par ce biais, cette transformation exploite certaines caractéristiques des termes en style CPS. Cependant, nous pourrions en exploiter d'autres.

Lors d'une mise en style CPS, deux sortes de variables différentes sont ajoutées :

Les variables de continuation : Il s'agit du paramètre supplémentaire que prend chaque abstraction d'origine et qui correspond à l'abstraction de la continuation courante, celle qui récupérera le résultat du calcul et qui interagit comme un contexte. Dans les transformations décrites en section 4.1, il s'agit des variables notées k.

Les paramètres de continuation : Il s'agit des paramètres des continuations qui ont pour rôle d'abstraire le résultat qu'elles récupèrent afin de l'exploiter. Dans les transformations décrites en section 4.1, elles sont notées n ou m.

Par exemple, considérons la transformation d'une application $M\ N$,

$$\lambda k.\ [\![M]\!]\ [\lambda m.\ [\![N]\!]\ [\lambda n.\ m\ n\ k]].$$

Dans ce terme, k est une variable de continuation. Celle-ci abstrait la continuation courante qui récupérera le résultat de l'application, tandis que m et n sont des paramètres de continuation. Si l'on distingue ces différentes sortes de variables ainsi que leurs liaisons, des propriétés intéressantes apparaissent. Ainsi, une variable de continuation est linéaire et ne coexiste pas avec d'autres variables de continuation. De même, les paramètres de continuation sont linéaires, en revanche, ils peuvent se retrouver sous plusieurs liaisons de paramètres de continuation. L'exploitation de ces caractéristiques pourrait être intéressante à étudier. Notamment elle permettrait d'utiliser des substitutions plus adaptées et plus efficaces, tirant parti de la linéarité des variables de continuation par exemple. Nous avons commencé cette étude dans des travaux en commun avec Olivier Danvy.

5 Construction de fermetures minimales explicites

Les termes fonctionnels des langages fonctionnels sont des abstractions dont les corps ne sont pas forcément clos. Les langages impératifs disposent de fonctions dont les corps sont clos et dont les déclarations, par conséquent, peuvent être globales. Une des étapes vers la génération de code Cminor, est le passage des termes fonctionnels de nML aux fonctions globales de Cminor. Il s'agit de réorganiser un programme fonctionnel, constitué d'un terme, en un programme constitué d'une liste de fonctions globales et d'une fonction principale. A priori, on pourrait vouloir créer une fonction à chaque abstraction et la remplacer par son nom dans le terme. Ce dernier serait soit le corps d'une autre fonction soit la fonction principale qui n'a pas de paramètre formel. Chaque abstraction deviendrait alors une fonction globale qui serait représentée par un pointeur de code en Cminor.

Considérons par exemple l'expression :

$$(\lambda\ [x; y].\ x + y)\ [3; 4]$$

La stratégie de compilation esquissée ci-dessus entraînerait la déclaration globale de la fonction :

$$(f : \{\texttt{params}\ :\ [x; y]\ ;\ \texttt{body}\ :\ x + y\})$$

et l'appel dans le terme deviendrait : $f\ [3; 4]$.

Cela serait faire l'impasse sur une réalité : le corps d'une abstraction n'est pas forcément clos, il peut contenir des variables libres. Exemple :

$$\texttt{let}\ z = 5\ \texttt{in}\ (\lambda\ [x].\ x + z)\ [3].$$

La variable z dans le corps de l'abstraction est libre. Dès lors, globaliser naïvement l'abstraction conduit à déclarer une fonction dont le corps n'est pas clos. La globalisation des abstractions en fonctions nécessite une fermeture des corps d'abstraction, afin de faire disparaître les variables libres. Les valeurs de ces variables libres sont essentielles. Rappelons nous, déjà lors de l'évaluation d'une abstraction dans le langage εML et ses variantes, les valeurs fermetures contenaient une copie de l'environnement courant pour accéder aux valeurs des variables libres du corps de l'abstraction. Cependant, une copie totale de l'environnement n'est pas nécessaire, seules les valeurs des variables libres dans le corps de l'abstraction sont indispensables. Une valeur fermeture dont l'environnement ne concerne que les variables libres du corps d'abstraction est une fermeture *minimale*.

Il existe différentes manières de résoudre la fermeture des abstractions lors de la compilation de langages fonctionnels telles que :

La défonctionalisation : introduite par Reynolds [102], il s'agit de représenter l'espace des valeurs de fonctions par un type concret. Cette méthode consiste à transformer

une fonction de première classe en un constructeur appliqué aux variables libres de l'abstraction. Une fonction `apply` effectue l'application en filtrant sur les constructeurs correspondant aux fonctions. Le corps de la clause est le corps de la fonction correspondante. Par exemple, pour l'abstraction $\lambda\ x.\ x + z$, on crée le constructeur $F_0\ z$. L'application $(\lambda\ x.\ x + z)\ 3$ devient `apply` $(F_0\ z)\ 3$, avec :

$$\texttt{apply}\ f\ x = \texttt{match}\ f\ \texttt{with}\ \ldots | F_0\ z \rightarrow x + z | \ldots .$$

Dans cette première version, l'espace fonctionnel n'est pas partitionné. Pour plus d'efficacité, cet espace peut être partitionné soit par type [9], ou bien encore après une analyse de flot de contrôle [106]. La fermeture des abstraction est effectué par défonctionalisation dans le compilateur `MLton` [116]. La défonctionalisation peut être aussi utilisée dans la compilation de langages orientés objets, par exemple dans `GNU Eiffel`. Elle intervient dans la construction d'interpréteur par passage de continuation [26].

Les supercombinateurs : cette méthode a été implémentée dans le compilateur Glasgow Haskell [92]. L'utilisation de supercombinateurs combinée à la réduction de graphe se prête particulièrement bien à la compilation de langages paresseux. Plus précisément, la réduction de graphe [114] consiste à représenter les $\lambda-$termes par leurs arbres de syntaxe abstraite. La représentation sous forme de graphe apparaît lors des réductions. Une *réduction de graphe* est l'implantation des règles de réduction du $\lambda-$calcul sur les graphes représentant les $\lambda-$termes. Un *supercombinateur* est une abstraction dont la forme syntaxique close est spécifique :
$\lambda\ x_0.\ \ldots \lambda\ x_n.\ T$ où T n'est pas une abstraction, ce qui permet de transformer un supercombinateur en une fonction globale de corps T et de paramètres $x_0, \ldots, x_n$. Il est possible de transformer une abstraction quelconque en un supercombinateur. Cette transformation est appelée le $\lambda-$*lifting* [55]. Concrètement, il s'agit d'abstraire les variables libres dans le corps de l'abstraction en faisant des paramètres formels supplémentaires. $\lambda\ x.x + z$ devient $\lambda\ z.\ \lambda\ x.\ x + z$ et l'application $(\lambda\ x.\ x + z)\ 3$ devient $(\lambda\ z.\ \lambda\ x.\ x + z)\ z\ 3$. La réduction d'une application sur les graphes est alors effective lorsque tous les arguments sont présents et se fait de manière simultanée. Il est aussi possible de ne pas générer de supercombinateurs, pour cela, il faut utiliser une liste de combinateurs prédéfinis (par exemple les combinateurs S, K et I).

La construction de fermetures explicite (*closure conversion*) est le plus ancienne et la plus courante des méthodes. Elle a été introduite par Landin [63]. C'est aussi la méthode que nous implémentons dans notre étude et que nous détaillons dans le paragraphe suivant.

L'explicitation des fermetures Une abstraction devient une structure de données appelée *fermeture*. Une fermeture est la combinaison d'une fonction globale et d'un environnement contenant les valeurs des variables libres dans le corps d'abstraction d'origine. La fonction globale prend alors les mêmes paramètres formels que l'abstraction d'origine et, de manière transparente, elle prend un paramètre supplémentaire qui représente la fermeture elle-même. Les variables libres sont alors accédées via ce paramètre supplémentaire. Afin de clore le corps d'une abstraction, elle est remplacée par la paire d'une fonction globale et d'un environnement. Par exemple, $(\lambda\ x.\ x + z)$ devient (f, z) avec dans la liste de fonctions globales la fonction f définie comme suit : $(f : \{\texttt{params} = [clos; x];\ \texttt{body} = x + \texttt{field}\ 1\ clos\})$, où `field` 1 *clos* désigne le premier élément de la fermeture explicite minimale *clos*, ici z.

Cette méthode est employée dans tous les compilateurs de Scheme, et dans les compilateurs OCaml. L'explicitation des fermetures est aussi utilisée dans le compilateur SML of New Jersey, alliée à une mise en forme CPS. On parle alors de *closure-passing style* [4].

L'explicitation des fermetures est généralement effectuée vers des langages intermédiaires non typés. Cependant, expliciter les fermetures vers un langage typé [85] permet de transmettre des informations utiles aux phases plus basses du compilateur.

L'explicitation des fermetures peut aussi être employée dans la compilation des objets [42].

L'environnement d'une fermeture permet de récupérer les valeurs des variables libres dans le corps de l'abstraction concernée lors de l'évaluation de ce corps (lors de l'évaluation d'une application). L'environnement d'une fermeture peut être plus ou moins approximé. On distingue alors :

Les fermetures complètes La fermeture contient une copie complète de l'environnement des variables locales au moment de l'évaluation de l'abstraction (comme dans les sémantiques des langages intermédiaires définis jusqu'ici). Les fermetures de la CAM étaient des fermetures à environnement complet.

Les fermetures minimales Dans l'environnement courant à l'évaluation d'une abstraction, seules nous importent les variables libres dans son corps, afin de le clore. Les fermetures minimales restreignent leur environnement aux seules variables libres dans le corps de l'abstraction concernée. L'explicitation des fermetures des compilateurs OCaml et SML of New Jersey produit des fermetures minimales.

Les fermetures légères (*lightweight closures*) [115] sont une optimisation des environnements minimaux. Toutes les variables libres dans le corps ne sont pas dans l'environnement de fermeture, il y a une sélection parmi ces variables libres selon qu'elles sont ou pas présentes dans tous les sites d'appel de l'abstraction.

Dans ce chapitre, nous étudions l'explicitation des fermetures du langage nML (voir la section 3.4.2) avec le lieur d'abstraction récursif μ vers un langage intermédiaire organisé en fonctions globales closes et une fonction principale, le langage Fml. Nous avons choisi une représentation de fermetures minimales. La méthode employée peut se résumer comme suit (en gardant en tête la représentation d'une fermeture sous forme de bloc mémoire).

L'abstraction $\lambda(\vec{x}).\ t$, devient la paire

d'une fonction globale :

$$\{f : \texttt{params}\ clos; \vec{x}\ \ ; body : t'\}$$

où les variables libres de t, noté v_i deviennent des accès à des champs de la fermeture $clos$, $clos(i)$:

$$t' = t[v_1 \leftarrow \texttt{field}\ 1\ clos] \ldots [v_n \leftarrow \texttt{field}\ n\ clos].$$

d'une estampille locale : en lieu et place syntaxique de l'abstraction d'origine afin de récupérer les valeurs dynamiques des nouveaux champs de la fermeture, exactement comme les valeurs fermeture le faisaient jusqu'à présent.

$$f([\![v_1; \ldots; v_n]\!]).$$

Ainsi, la fonction globale matérialise son accès à la fermeture et la manipulation de la fermeture lui correspondant au travers d'un paramètre formel supplémentaire, ce qui permet de clore le corps de la fonction globale.

Exemple :

$$\texttt{let } z = 5 \texttt{ in let } y = \lambda[x].\ x + z \texttt{ in } y\ 4.$$

Nous créons donc la fonction globale :

$$f = \{\texttt{params } : [clos, x]\ ;\ \texttt{body } : x + \texttt{field } (1)\ clos\}.$$

En lieu et place de la définition syntaxique de l'abstraction, nous définissons la fermeture :

$$\texttt{let } z = 5 \texttt{ in let } y = f\ (z) \ldots.$$

La prise en charge du paramètre formel supplémentaire lors de l'application est implicite. L'application reste inchangée syntaxiquement. En effet, le passage de l'argument supplémentaire s'effectue lors de l'évaluation dynamique, puisque nous disposons déjà de la valeur de la fermeture (l'appelant). Nous évitons ainsi une évaluation inutile de l'argument supplémentaire.

Une conséquence directe de la prise en paramètre de la fermeture elle-même est de faire disparaître la distinction entre fonction récursive ou pas. En effet, les occurrences d'appels récursifs sont remplacées par un appel au paramètre supplémentaire représentant la fermeture. Ainsi :

$$\texttt{letrec } f\ x\ =\ x + f\ [x - 1] \texttt{ in } \ldots.$$

Entraîne la déclaration de la fonction globale :

$$f'\ =\ \{\texttt{params } : [clos; x]\ ;\ \texttt{body } : x + (\texttt{field } (0)\ clos)\ [x - 1]\}.$$

Les valeurs dynamiques leur correspondant se trouvent au niveau de la fermeture syntaxique. Nous choisissons de les expliciter syntaxiquement au niveau des fermetures afin de les manipuler et de pouvoir les traiter dans les prochaines phases.

Nous proposons de construire les fermetures au travers d'une traduction de nML vers Fml. La section suivante définit le langage Fml avant de décrire la construction de fermetures et d'esquisser la preuve de préservation sémantique.

5.1 Le langage Fml

Fml est un langage fonctionnel avec des fonctions déclarées globalement et définies de manière dynamique dans la syntaxe au travers de *fermetures minimales explicites*. Fermeture, parce qu'elle se réfère à la déclaration de la fonction par son nom et localise syntaxiquement l'environnement dynamique à considérer. Minimale, parce qu'elle ne considère dynamiquement que les valeurs associées aux variables libres. Explicite, parce qu'elle porte ces variables libres syntaxiquement.

5.1.1 Syntaxe

La grammaire suivante définit la syntaxe du langage Fml :

Termes :	$t ::= x$	variable
	$\mid$ `field` $n\ t$	$n$ième champ d'une fermeture
	$\mid f\ (t_1; \ldots; t_n)$	fermeture minimale
	$\mid t\ [t_0; \ldots; t_n]$	
	$\mid$ `let` $x\ =\ t_1$ `in` t_2	
	$\mid C\ (t_1; \ldots; t_n)$	
	$\mid$ `match` t `with` $\pi_0; \ldots; \pi_n$	
Motifs :	$\pi ::= x_1; \ldots; x_n \rightarrow t$	
Fonction :	$def\ := (f : \{\texttt{params}\ :\ x_0; \ldots; x_n\ ;\ \texttt{body}\ :\ t\})$	
programme :	$prog\ := \{\texttt{defs}\ :\ \vec{def}\ ;\ \texttt{main}\ :\ t\}$	

Organisation syntaxique Un programme p Fml est composé d'une liste de déclarations de fonctions p.`defs` et d'un terme principal p.`main`. Une déclaration de fonction f comporte une liste de noms de variables formant la liste de ses paramètres formels f.`params` et un terme, son corps, f.`body`.

Variables nommées : Fml n'est pas défini dans le formalisme de de Bruijn. Une variable y est identifiée nominalement et non relativement à sa position dans le terme par rapport au lieur la déclarant. De même, les paramètres formels des motifs sont nommés.

Fermetures minimales explicites : Les abstractions aussi bien simples que récursives ont disparues. Elles laissent place aux fermetures minimales explicites. Syntaxiquement, une fermeture $f\ (t_1; \ldots; t_n)$ localise la définition dynamique de la fonction globale f du programme. La fermeture correspondant à f sera passé comme premier argument aux appels de la fonction f. La liste de termes $t_1; \ldots; t_n$ permet de clore le corps de f. Elle correspond aux variables libres du corps de l'abstraction d'origine. Notons qu'il ne s'agit pas d'une liste de noms de variables. En effet, il peut s'agir aussi d'un accès à un champ d'une fermeture enrobante.
Les anciennes variables libres laissent place à des accès à des champs de fermeture : `field` $n\ t$ est un accès au $n$ième terme d'une fermeture.
Remarquons que toute information concernant le caractère récursif d'une abstraction disparaît. En effet, au niveau d'une fonction Fml, le paramètre supplémentaire représentant la fermeture peut aussi faire référence à la variable de récursion.

Conservation de l'expressivité : Fml, comme nML, offre la possibilité de lier localement et de filtrer sur un type concret. On y retrouve aussi les constructeurs appliqués et les applications n-aires. Le paramètre formel supplémentaire correspondant à la fermeture elle-même est passée implicitement et non syntaxiquement lors de l'application. Elle est considérée uniquement lors de l'évaluation dynamique.

L'addition sur les entiers naturels que nous suivons en exemple devient :

```
{ defs : [ (f0 ; {params : [clos ;x ;y] ;
                  body : match x with
                         | 0 -> y
                         | 1 -> C(1) [ clos [x ;y]]
                         end}) ;
           (f1 ; {params : [clos ;x] ;
                  body : clos(1)[x,C(1) [C(0)]]}) ] ;
  terme : let x0 = Clos f0  in let x1 = C(1)[C(1)[C(0)]] in
           (Clos f1 (x0)) [x1]
}
```

Il s'agit de la fermeture du programme nML :

```
let mu(1) [f,x,y].
       match x with
       | 0 -> y
       | 1(z) -> C(1) (f [x,z])
      end
in
 let z=(C(1) (C(1) C(0))) in
   (\ x. f [x, (C(1) (C(0))]) z
```

Apparaissent dans cet exemple deux abstractions auxquelles correspondent les deux fonctions globales $f0$ et $f1$. La première abstraction est d'arité 2 et correspond à la fonction $f0$ dont les deux paramètres d'origine ont été nommés par x, y. Cette fonction a son paramètre supplémentaire *clos*. L'abstraction d'origine n'a pas de variable libre, cependant l'appel récursif profite de la matérialisation de la fermeture et devient un appel à *clos*.

La deuxième abstraction, `(\ x. f [x, (C(1) (C(0))])`, est d'arité 1 et a une variable libre f. Cette abstraction correspond à la fonction globale $f1$, son paramètre formel a été nommé par x et comme pour la fonction $f0$, le paramètre formel supplémentaire correspondant à la considération de la fermeture *clos*. Ainsi, l'accès à la seule variable libre dans l'abstraction d'origine f devient $clos(1)$.

En plus des fonctions globales, le terme d'origine donne lieu à un terme Fml. Les variables y ont été nommées et les abstractions ont laissées place à des estampilles de fermetures, les fermetures minimales explicites. La première n'ayant pas de variable libre et ne porte que le nom de la fonction globale : `Clos f0`. La deuxième fermeture porte la variable libre $x0$.

5.1.2 Sémantique naturelle avec environnement

La sémantique d'un terme Fml est elle aussi donnée en sémantique naturelle en appel par valeur, suivant une stratégie d'évaluation faible avec environnement.

Les valeurs sémantiques de Fml sont de deux sortes :

Valeur de constructeur appliqué : $(C, v_1; \ldots; v_n)$, similaire aux valeurs de constructeurs appliqués de nML. C est le numéro du constructeur (les constructeurs sont

identifiés par leurs numéros d'apparition dans la définition du type concret d'origine, voir la section 2.4.4) et $v_1; \ldots; v_n$ les valeurs des arguments du constructeur.

Fermeture minimale : $(f, v_1; \ldots; v_n)$, les valeurs fermetures de Fml sont différentes de celles de nML. Il n'y a pas de fermeture récursive. Les fermetures sont minimales, elles ne comportent pas de copie de l'environnement courant, mais uniquement les valeurs des variables libres $v_1; \ldots; v_n$. Enfin, l'indication d'arité et le corps d'abstraction ont laissé place au nom de la fonction f.

Les valeurs sémantiques de Fml sont les suivantes :

Valeur : $t ::= (C, v_1; \ldots; v_n) \mid (f, v_1; \ldots; v_n)$

Un environnement e est une structure associant à un nom de variable x, une valeur v.

On peut récupérer la valeur associée à une variable : $e(x) = v$. Si e n'associe aucune valeur à x, alors $e(x) = \perp$.

On peut associer une valeur v à une variable x : $e[x \leftarrow v]$.

ε désigne l'environnement n'associant de valeur à aucune variable : l'environnement vide.

L'évaluation doit aussi se référer à une liste de déclarations de fonctions. Le jugement d'évaluation d'un terme Fml s'énonce comme suit :

$$S \ , \ e \vdash t \Rightarrow v,$$

qui se lit : dans l'environnement d'évaluation formé par la liste de déclarations de fonctions S et de l'environnement de variables e, le terme t s'évalue en la valeur v.

La sémantique de Fml est décrite par le jeu de règles d'inférence est donné sur la figure 5.1.1. La liste de déclarations de fonctions S est globale, elle ne varie pas au cours de l'évaluation d'un terme, elle correspond à la liste de fonctions globales du programme considéré. Détaillons ensemble ces règles d'évaluation. L'environnement e subit des modifications et se comporte comme les environnements rencontrés précédemment.
Une variable x s'évalue en v dans e, si v lui est associée dans e.
L'accès à un élément d'une fermeture `field` i t dans l'environnement e s'évalue en v_i si t dans le même environnement s'évalue en une fermeture $(f, v_1; \ldots; v_n)$ et que v_i est le $i$ème élément de $v_1; \ldots; v_n$. L'accès au champ 0 d'une fermeture n'a pas de sens en Fml, cela facilitera la traduction vers un accès dans le bloc Cminor correspondant. L'offset 0 y est réservé pour le pointeur code de la fonction.
Une fermeture f $(t_1; \ldots; t_n)$ dans (S, e) s'évalue en la valeur fermeture $(f, v_1; \ldots; v_n)$ si f a été déclarée dans S et que dans le même environnement les $t_1; \ldots; t_n$ s'évaluent en $v_1; \ldots; v_n$.
L'évaluation d'un constructeur appliqué est identique à celle de nML. De même pour la liaison locale et le filtrage.
Les indications de récursivité ayant disparues, il n'y a plus qu'une règle d'évaluation de l'application. Si t s'évalue en la fermeture $(f, w_1; \ldots; w_k)$, avec $(f, d) \in S$ et $t_0; \ldots; t_n$ s'évalue en $v_0; \ldots; v_n$ dans le même environnement, alors t $[t_0; \ldots; t_n]$ s'évalue en v, si dans l'environnement attribuant aux paramètres formels de d les valeurs des arguments $v_0; \ldots; v_n$ et la fermeture mise en jeu au paramètre *clos*, le corps de d s'évalue en v.

$$\frac{e(x) = v}{S, e \vdash x \Rightarrow v} \qquad \frac{S, e \vdash t \Rightarrow (f, v_1; \ldots; v_n) \quad 1 \leq i \leq n}{S, e \vdash \texttt{field}\ i\ t \Rightarrow v_i}$$

$$\frac{(f, d) \in S \quad S, e \vdash t_i \Rightarrow v_i \quad (\text{pour tout } i \in [1; n]\ )}{S, e \vdash f\ (t_1; \ldots; t_n) \Rightarrow (f, v_1; \ldots; v_n)}$$

$$\frac{S, e \vdash t_i \Rightarrow v_i \quad (\text{pour tout } i \in [1; n]\ )}{S, e \vdash C\ (t_1; \ldots; t_n) \Rightarrow (C, v_1; \ldots; v_n)} \qquad \frac{S, e \vdash t_1 \Rightarrow v_1 \quad S, e[x\ \leftarrow\ v_1] \vdash t_2 \Rightarrow v}{S, e \vdash \texttt{let}\ x\ =\ t_1\ \texttt{in}\ t_2 \Rightarrow v}$$

$$\frac{\begin{array}{c} S, e \vdash t \Rightarrow (f, w_1; \ldots; w_k) \quad (f, d) \in S \quad d.\texttt{params} = clos :: x_0; \ldots; x_n \\ S, e \vdash t_i \Rightarrow v_i \quad (\text{pour tout } i \in [0, n]\ ) \\ S, \varepsilon[clos\ \leftarrow\ (f, w_1; \ldots; w_k)] :: [x_0 \leftarrow v_0] \ldots [x_{(n-1)} \leftarrow v_{(n-1)}] \vdash d.\texttt{body} \Rightarrow v \end{array}}{S, e \vdash t\ [t_0; \ldots; t_n] \Rightarrow v}$$

$$\frac{\begin{array}{c} S, e \vdash t \Rightarrow (C, v_1; \ldots; v_k) \quad \pi_C = x_1; \ldots; x_k \rightarrow t_i \\ S, e[x_1\ \leftarrow\ v_1] \ldots [x_k \leftarrow v_k] \vdash t_i \Rightarrow v \end{array}}{S, e \vdash \texttt{match}\ t\ \texttt{with}\ \pi_0; \ldots; \pi_n \Rightarrow v}$$

FIG. 5.1.1 – Sémantique à grand pas avec environnement de Fml

La sémantique comporte aussi des vérifications concernant les noms de variables. Nous voulons que toutes les variables soient distinctes de `Root`, qui sera un paramètre formel spécial pour les fonctions Cminor. Dans notre formalisation Coq, ces vérifications apparaissent en prémisses des règles d'évaluation.

Une variable ne s'évalue que si elle diffère de `Root` : $x \neq \texttt{Root}$.

Les paramètres formels lors de l'évaluation du corps de fonction dans l'évaluation d'une application sont tous différents de `Root` : $x_i \neq \texttt{Root}$, pour tout i tel que : $0 \leq i \leq n$.

De même, lors de l'évaluation d'un filtrage.

Un programme p s'évalue en v, si dans l'environnement d'évaluation formé par sa liste de déclarations de fonctions et l'environnement vide ε, son terme principal $p.\texttt{main}$ s'évalue en v et que la liste de déclarations de fonctions $d.\texttt{defs}$ est bien formée. Une liste de déclarations de fonctions est bien formée si chaque nom de fonction est unique et diffère de `main` et `alloc_block`.

$$\frac{\texttt{well formed}\ \ p.\texttt{defs} \quad p.\texttt{defs}\ ,\ \varepsilon \vdash p.\texttt{main} \Rightarrow v}{\vdash p \Rightarrow v}$$

5.2 Calcul des fermetures minimales

Notre construction de fermetures s'effectue au travers d'une traduction de programme nML en programme Fml. Elle implique la construction d'une liste de déclarations de fonctions Fml, l'attribution à chaque fonction d'un nom et le nommage des variables, jusqu'ici

représentées par des indices de de Bruijn. Nous voulons nommer de manière unique chaque variable d'une part et chaque fonction d'autre part. Ces trois actes nécessitent généralement trois variables globales : la liste en construction, un générateur de noms de fonctions et un générateur de noms de variables.

Coq étant un langage purement fonctionnel, nous utiliserons une monade d'état pour représenter les traits impératifs de notre algorithme. Dans un premier temps, nous donnons une présentation algorithmique de la transformation pour plus de lisibilité. Nous décrirons ensuite la présentation monadique faisant partie de l'implantation. Enfin, nous donnons une présentation relationnelle utilisée dans la preuve.

5.2.1 Présentation algorithmique

Cette première présentation de notre traduction de terme nML en programme Fml nous permet d'expliciter notre transformation d'un point de vue algorithmique, sans se préoccuper des restrictions d'implantation.

La traduction d'une variable dépend de son caractère libre ou pas dans le corps d'une abstraction (simple ou récursive). Nous noterons, $FV(t)$ la liste des indices représentant les variables libres dans le terme t. La traduction se fait au travers d'un environnement de compilation Γ qui associe à un indice n, une information γ. Cette information est soit `Var` x s'il s'agit d'une variable liée, soit `Fld` n *clos* s'il s'agit d'une variable libre. Dans ce cas, *clos* fait référence à la fermeture dont $n!$ est une variable libre. Cette fermeture porte un nom, puisqu'elle est passée en paramètre à la fonction, afin de clore le corps de la fonction.
Un indice de de Bruijn étant une représentation locale d'une variable (position relative du lieur la définissant), les indices de l'environnement doivent être réactualisées en traversant chaque lieur. $\uparrow^n\ \Gamma$ augmente de n chaque indice présent dans Γ. $\uparrow\ \Gamma$ augmente chaque indice de Γ de 1.
`new_var` est le générateur de nom unique de variable. (`new_vars` n) génère n noms uniques de variables. `new_fun` est le générateur de nom unique de fonction.
S représente la liste de déclarations de fonctions en construction. Cette liste est initialisée en début de traduction d'un programme par la liste vide.

Traduction des variables : La traduction d'une variable $n!$ dépend de l'information qui lui est associée dans Γ. Si l'information est `Var` x, alors elle se traduit vers la variable x, si l'information est (`Fld` $p\ x$), alors elle se traduit vers l'accès au $(p+1)$ième élément de x, `field` $(p+1)\ x$. Nous conservons ainsi le premier champ de la fermeture pour le futur pointeur vers le code de la fonction, en prévision de la construction du bloc Cminor futur. Ainsi :

$$[\![n!]\!]_\Gamma \quad = \quad \begin{cases} x & \text{si } \Gamma(n) = \texttt{Var}\ x\,; \\ \texttt{field}\ (p+1)\ x & \text{si } \Gamma(n) = \texttt{Fld}\ p\ x \end{cases}$$

Fermeture des abstractions : Lors de la traduction d'une abstraction $\lambda^n.\ t$, nous allons créer une nouvelle fonction globale et construire une fermeture.
Pour cela, on commence par générer $(n+1)$ noms de variables uniques correspondant aux paramètres formels de l'abstraction : $x_0; \ldots; x_n :=$ `new_vars`$(n+1)$. De plus, on

génère un nom unique pour le paramètre supplémentaire représentant la fermeture : $clos :=$ `new_var`. Nous avons ainsi généré la liste des paramètres formels de la fonction globales en construction : $(clos, x_0; \ldots; x_n)$.
Afin de fermer le corps de la fonction en construction, nous calculons les variables libres dans le corps de l'abstraction $v_1; \ldots; v_k := FV_{(n+1)}(t)$. Ce calcul et la connaissance de la liste des paramètres formels de la fonction, nous permettent d'initialiser l'environnement de traduction qui fermera correctement le corps de l'abstraction. Nous créons ainsi, Γ' tel que :
- chaque variable libre v_i, pour $i \in [1; k]$, est associée à un futur champs de la fermeture désignée par le paramètre $clos$: `Fld` i $clos$.
- chaque paramètre formel de l'abstraction d'origine $i!$, pour $i \in [0; n]$ est associée à x_i.

Le corps de l'abstraction traduit dans cet environnement, nous fournit le corps clos de la fonction globale en construction : $[\![t]\!] \triangleright \Gamma'$. Il ne reste plus qu'à générer un nom unique pour la nouvelle fonction : $f :=$ `new_fun` et l'ajouter dans l'état courant.
Algorithmiquement :

$$
\begin{array}{rcl}
[\![\lambda^n.\ t]\!]_\Gamma & = & \text{Soit } x_0; \ldots; x_n := \texttt{new\_vars}(n+1); \\
& & \text{Soit } clos := \texttt{new\_var}; \\
& & \text{Soit } v_1; \ldots; v_k := FV_{(n+1)}(t); \\
& & \text{Soit } \Gamma_1 := (v_1, \texttt{Fld } 0\ clos), \ldots, (v_k, \texttt{Fld } (k-1)\ clos); \\
& & \text{Soit } \Gamma' := (n, \texttt{Var } x_n), \ldots, (0, \texttt{Var } x_0), \Gamma_1; \\
& & \text{Soit } f := \texttt{new\_fun}; \\
& & \text{Faire } S \leftarrow (f, \{(clos, x_0; \ldots; x_n); [\![t]\!]_{\Gamma'}\}); S; \\
& & \text{Retourner } f\ ([\![v_1!]\!]_\Gamma, \ldots, [\![v_k!]\!]_\Gamma)
\end{array}
$$

Concernant la fermeture d'une abstraction récursive $\mu^n.\ t$, la différence avec la fermeture d'une abstraction simple réside en le traitement de la variable de récursion 0!. Elle sera traduite par le paramètre formel désignant la fermeture $clos$.

Traduction des termes avec lieurs : La traduction d'une liaison locale `let` t_1 `in` t_2, commence par générer un nom unique x, traduit récursivement t_1 dans Γ et t_2 dans $((0, \texttt{Var } x), \uparrow \Gamma)$. En effet, les indices de de Bruijn étant une représentation relative des variables par rapport à la position des lieurs, il est nécessaire d'actualiser les indice de l'environnement de traduction. Soit :

$$
\begin{array}{rcl}
[\![\texttt{let } t_1 \texttt{ in } t_2]\!]_\Gamma & = & \text{Soit } x := \texttt{new\_var}; \\
& & \text{Retourner}(\texttt{let } x\ =\ [\![t_1]\!]_\Gamma \texttt{ in } [\![t_2]\!]_{(0,\texttt{Var } x);\uparrow\Gamma})
\end{array}
$$

La traduction d'une clause de filtrage $n \rightarrow t$ commence par générer n noms uniques. Puis celle-ci actualise l'environnement de traduction, en liftant les indices de Γ de n et en associant les n indices liés par le corps de la clause aux n noms de variables fraîchement générés. Enfin, elle produit la clause Fml de motif $x_1; \ldots; x_n$ et de corps la

traduction de t dans l'environnement actualisé. La traduction d'une clause de filtrage est donc :

$$\begin{array}{rcl} [\![n \to t]\!]_\Gamma & = & \text{Soit } x_1; \ldots; x_n := \texttt{new\_vars}\ n; \\ & & \text{Soit } \Gamma' := (0, \texttt{Var}\ x_1), \ldots, ((n-1), \texttt{Var}\ x_n), \uparrow^n \Gamma; \\ & & \text{Retourner } (x_1; \ldots; x_n \to [\![t]\!] \rhd \Gamma') \end{array}$$

Les autres termes : Les traductions de l'application et du constructeur appliqué se font récursivement sur les sous-termes dans l'environnement de traduction courant Γ. La traduction du filtrage se fait récursivement sur son terme discriminé et ses clauses.

La figure 5.2.1 donne l'algorithme dans son intégralité.

5.2.2 Présentation fonctionnelle

Comme indiqué précédemment, Coq est un langage purement fonctionnel. L'algorithme de la section 5.2.1 présente de forts traits impératifs, en particulier, l'utilisation de variables globales. Ces variables subissent des modifications tout le long de la transformation. Un moyen d'encoder ces effets de bord dans un langage purement fonctionnel est d'utiliser une monade d'état. La transformation pouvant échouer, nous utilisons une monade d'état composée avec la monade d'erreur.

Dans notre développement, les noms sont représentés par des entiers positifs, le type `ident` est un alias du type `positive` de Coq. Générer des noms uniques se résume alors à associer à chaque indice un positif différent. Pour cela, on transporte dans la monade d'état un positif qui sera attribué au prochain nom à générer. Ce positif sera alors incrémenté, afin de garantir que chaque nom généré est différent.

Cette monade se définit par le triplet monadique composé par son type, ses fonctions d'encapsulation et sa fonction de composition.

Commençons par définir son type :

```
Definition functions := list (ident *def).

Inductive res (A : Set) : Set :=
  | Error : res A
  | OK : A -> functions -> ident ->ident -> res A.

Definition mon (A : Set) : Set := functions ->ident -> ident -> res A.
```

Le type de la monade (`mon A`) est polymorphe. Elle prend en argument un état initial : la liste de déclarations de fonctions en construction, le générateur de noms de fonctions et le générateur de noms de variables. Si ces arguments sont présents, elle rend un objet de type

$$
\begin{array}{rcl}
[\![n!]\!]_\Gamma & = & \begin{cases} x & \text{si } \Gamma(n) = \texttt{Var}\ x\,; \\ \texttt{field}\ p+1\ x & \text{si } \Gamma(n) = \texttt{Fld}\ p\ x \end{cases} \\
[\![\lambda^n.\ t]\!]_\Gamma & = & \text{Soit } x_0;\ldots;x_n := \texttt{new\_vars}(n+1); \\
 & & \text{Soit } \mathit{clos} := \texttt{new\_var}; \\
 & & \text{Soit } v_1;\ldots;v_k := FV_{(n+1)}(t); \\
 & & \text{Soit } \Gamma_1 := (v_1, \texttt{Fld}\ 0\ \mathit{clos}), \ldots, (v_k, \texttt{Fld}\ (k-1)\ \mathit{clos}); \\
 & & \text{Soit } \Gamma' := (n, \texttt{Var}\ x_n), \ldots, (0, \texttt{Var}\ x_0), \Gamma_1; \\
 & & \text{Soit } f := \texttt{new\_fun}; \\
 & & \text{Faire } S \leftarrow (f, \{(\mathit{clos}, x_0;\ldots;x_n); [\![t]\!]_{\Gamma'}\}) :: S; \\
 & & \text{Retourner } f\ ([\![v_1!]\!]_\Gamma, \ldots, [\![v_k!]\!]_\Gamma) \\
 & & \\
[\![\mu^n.\ t]\!]_\Gamma & = & \text{Soit } x_0;\ldots;x_n := \texttt{new\_vars}(n+1); \\
 & & \text{Soit } \mathit{clos} := \texttt{new\_var}; \\
 & & \text{Soit } v_1;\ldots;v_k := FV_{(n+2)}(t); \\
 & & \text{Soit } \Gamma_1 := (v_1, \texttt{Fld}\ 0\ \mathit{clos}), \ldots, (v_k, \texttt{Fld}\ (k-1)\ \mathit{clos}); \\
 & & \text{Soit } \Gamma' := (0, \texttt{Var}\ \mathit{clos}), (n+1, \texttt{Var}\ x_n), \ldots, (1, \texttt{Var}\ x_0), \Gamma_1; \\
 & & \text{Soit } f := \texttt{new\_fun}; \\
 & & \text{Faire } S \leftarrow (f, \{(\mathit{clos}, x_0;\ldots;x_n); [\![t]\!]_{\Gamma'}\}) :: S; \\
 & & \text{Retourner } f\ ([\![v_1!]\!]_\Gamma, \ldots, [\![v_k!]\!]_\Gamma) \\
 & & \\
[\![\texttt{let}\ t_1\ \texttt{in}\ t_2]\!]_\Gamma & = & \text{Soit } x := \texttt{new\_var}; \\
 & & \text{Retourner}(\texttt{let}\ x\ =\ [\![t_1]\!]_\Gamma\ \texttt{in}\ [\![t_2]\!]_{(0,\texttt{Var}\ x)::\uparrow\Gamma}) \\
[\![t\ [t_0;\ldots;t_n]]\!]_\Gamma & = & \text{Retourner } ([\![t]\!]_\Gamma\ [[\![t_0;\ldots;t_n]\!]_\Gamma]) \\
[\![C\ (t_1;\ldots;t_n)]\!]_\Gamma & = & \text{Retourner } (C\ ([\![t_1;\ldots;t_n]\!]_\Gamma)) \\
[\![\texttt{match}\ t\ \texttt{with}\ \pi_0;\ldots;\pi_n]\!]_\Gamma & = & \text{Retourner } \texttt{match}\ [\![t]\!]_\Gamma\ \texttt{with}\ [\![\pi_0;\ldots;\pi_n]\!]_\Gamma \\
[\![n \to t]\!]_\Gamma & = & \text{Soit } x_1;\ldots;x_n := \texttt{new\_vars}n; \\
 & & \text{Soit } \Gamma' := (0, \texttt{Var}\ x_1), \ldots, ((n-1), \texttt{Var}\ x_n), \uparrow^n \Gamma; \\
 & & \text{Retourner } (x_1;\ldots;x_n \to [\![t]\!]_{\Gamma'})
\end{array}
$$

FIG. 5.2.1 – Algorithme du calcul de fermetures

`res`, où `res` se définit par deux constructeurs, l'un représentant le cas d'erreur (`Error`), l'autre représentant le cas sans erreur (`OK`). Il porte en argument les trois composantes de l'état final.

Les fonctions d'encapsulations de la monade sont au nombre de deux, une par constructeur :

```
Definition ret (A : Set) (a : A) : mon A :=  (s : functions)
          (f :ident)  (x :ident ) => OK a s f x.

Definition error (A : Set) : mon A := (s :functions)
          (f :ident) (x :ident) =>Error A.
```

`ret` est la fonction *unit* de la monade d'état. Elle permet d'encapsuler un objet de type `A` dans une monade `mon A`. `error` enrobe quant à elle, le cas erreur de la monade d'erreur dans `mon A`.

Enfin, sa fonction de composition *bind* est définie comme suit :

```
Definition bind (A B :Set) (f :mon A) (g :A -> mon B) : mon B :=
  (s : functions)(ff :ident)  (x :ident)
    match f s ff x with
    | Error => Error B
    | OK a s' f' x' => g a s' f' x'
    end.
```

`bind` permet d'appliquer une fonction `(g : A -> mon B)` à un objet de type `mon A` et retourne un objet de type `mon B`. Nous utilisons la notation *do* pour récupérer le résultat intermédiaire manipulé dans le `bind` en lui attribuant un nom. *do* $x \leftarrow a; b$ se lit comme `bind a (fun x -> b)`. On remarquera la similitude entre `bind` et le constructeur `let`. Elle permet de calculer $f\ x$ avec effet de bord et d'en stocker le résultat dans une variable qui sera liée dans la suite du calcul.

Ces trois composantes définissent la monade composée des monades d'état et d'erreur. Cependant, nous utilisons une définition enrichie, contenant les manipulations sur les trois composantes de l'état : les générations de noms et l'ajout de déclaration de fonctions. Ce sont ces manipulations qui changent l'état.

Ainsi, la génération d'un nom de variable frais se définit comme suit :

```
Definition ident_vars_add_var :=
 (s :functions)(f :ident)(x :ident) =>
  ret x s f (Psucc x).
```

L'ajout d'une définition de fonction dans l'état a pour définition :

```
Definition add_fundef (fundef :def) :=
  (s :functions)(f :ident)(x :ident) =>
   ret f ((f,fundef) ;s) (Psucc f) x.
```

La transformation s'énonce alors comme précédemment. L'environnement de traduction est identique à celui défini plus haut. Les effets de bord sont masqués par l'emploi du `do`. Les manipulations des variables globales sont remplacées par les manipulations des composantes de l'état de la monade.

Par exemple, la traduction d'une abstraction devient :

```
|Lamb n t =>
   let v := (FV (S n) nil t) in
   do env <- new_vars  (S n) ;
   do t' <- transf ( mk_ce (S n) v env) t ;
   let fdef := (mkdef  (env ;(names_params (S n) env)) t') in
   do fname <- add_fundef fdef ;
   match (transf_varl cenv v) with
   | None =>error Dterm
   | Some tl => ret (Clos fname tl)
   end
```

Lors de la traduction d'un programme Fml, l'état de la monade est initialisé. La liste de déclarations de fonctions est vide, le générateur de noms de fonctions est initialisé à 2. En effet, on réserve 0 pour la fonction `main` Cminor et 1 pour la fonction Cminor appelant l'allocation du garbage collector, `alloc_block`. Le générateur de noms de variables est lui initialisé à 1, on réserve 0 pour le paramètre formel supplémentaire Cminor faisant référence aux racines, `Root`.

Ce programme est alors traduit dans cet état initial. Si la traduction échoue, alors la traduction du programme entier échoue. Par contre, si elle réussit, on récupère le terme traduit d et la liste de déclarations de fonctions finale s et on retourne le programme dont le terme principal est d et la liste de déclarations de fonctions s.

```
Definition transf_program (p :term) :=
match (transf nil nil p nil 2 1) with
| OK d s _ _ => Some ( mkprog d s)
| _ => None
end.
```

Notons que la monade d'état a été déplié pour la transformation du programme, en effet, on veut récuppérer la listes de fonctions globales.

5.2.3 Présentation relationnelle de la construction de fermetures

Au niveau de l'implantation, la monade d'état permet de cacher élégamment les traits impératifs et la propagation des effets de bord. Néanmoins, elle n'allège aucunement la difficulté à raisonner sur un état en évolution. Nous voulons exhiber une preuve de préservation sémantique de notre construction de fermetures.

Or, la sémantique de Fml considère, de manière globale, la liste de déclarations de fonctions du programme considéré S et non des fragments de cette liste en construction.

Cependant, la traduction ne peut qu'ajouter des déclarations de fonctions dans l'état : elle ne peut ni en retirer ni en modifier. La liste de déclarations de fonctions lors de la traduction a donc un comportement *monotone*. Autrement dit, la liste de l'état initial est incluse dans celle de l'état final et si S_i, S_j sont des états intermédiaires de cette liste apparaissant chronologiquement l'un après l'autre, alors S_i est incluse dans S_j qui elle même est incluse dans la liste de l'état final.

Nous basant sur cette observation, nous utilisons une troisième représentation de notre construction de fermetures, sous forme d'une relation. Elle met en relation un terme nML, t, avec un terme Fml, t', s'appuyant sur un environnement de traduction identique à celui décrit plus haut Γ. Cette relation est paramétrée par une liste de déclarations de fonctions S. Elle sera notée :

$$S, \Gamma \vdash t \approx t'.$$

Elle peut s'interpréter, grossièrement, comme " t' est le terme correspondant à l'explicitation des fermetures de t", tel que :

- A chaque abstraction de t, correspond une fonction globale dans S et une estampille de fermeture minimale explicite qui sont issues de l'explicitation de la fermeture de l'abstraction considérée.
- Chaque variable de t correspond à une variable ou à un accès à un élément de fermeture selon l'information qui lui est associée dans Γ.

Plus concrètement, la relation se définit par un jeu de règles d'inférence déterministe.

Les variables : La correspondance des variables de nML est dirigée par l'information associée à son indice dans l'environnement de traduction :

$$\frac{\Gamma(n) = \texttt{Var}\ x}{S, \Gamma \vdash n! \approx x} \qquad \frac{\Gamma(n) = \texttt{Fld}\ \ pos\ x}{S, \Gamma \vdash n! \approx \texttt{field}\ (pos + 1)\ x}$$

Les abstractions : Comme dit plus haut, on ne construit pas de fonction globale, mais on vérifie que la bonne fonction a été construite dans S. Considérons l'abstraction d'origine :

$$\lambda^n.\ t.$$

Elle correspond à la fermeture explicite :

$$f\ (t'_1; \ldots; t'_k).$$

Pour cela, il faut que dans S, f soit associée à une déclaration de fonction d telle que sa liste de paramètres formels est constituée de $(n+2)$ noms uniques et différents de `Root`, soit : `fresh` $clos; x_0; \ldots; x_n$.

De plus, le corps de cette déclaration de fonction se doit d'être la fermeture de t dans un environnement où :

- les $n+1$ premiers indices de de Bruijn (les paramètres formels) sont associés aux $x_0; \ldots; x_n$: $(0, \texttt{Var}\ x_0), \ldots, (n, \texttt{Var}\ x_n)$,
- les variables libres de t, notées v_k, sont associées à un emplacement sur le futur bloc représentant la fermeture, laquelle est matérialisée par le paramètre formel $clos$:

$$(v_1, \texttt{Fld}\ 0\ clos), \ldots, (v_k, \texttt{Fld}\ (k-1)\ clos).$$

Enfin, $t'_1; \ldots; t'_k$ correspondent aux variables libres.

$$\frac{\begin{array}{c}(f,d) \in S \quad \texttt{fresh}\ d.\texttt{params} \quad d.\texttt{params} = clos; x_0; \ldots; x_n \\ FV_{n+1}(t) = v_1; \ldots; v_k \quad S, \Gamma \vdash v_i! \approx t'_i \quad (\text{pour tout } i \in [1;k]) \\ S, (0, \texttt{Var}\ x_0), \ldots, (n, \texttt{Var}\ x_n), (v_1, \texttt{Fld}\ 0\ clos), \ldots, (v_k, \texttt{Fld}\ (k-1)\ clos) \vdash t \approx d.\texttt{body}\end{array}}{S, \Gamma \vdash \lambda^n.\ t \approx f\ (t'_1; \ldots; t'_k)}$$

Concernant les abstractions récursives, il faut ajouter dans l'environnement de traduction une liaison pour la variable de récursion : $((n+1), \texttt{Var}\ clos)$.

$$\frac{\begin{array}{c}(f,d) \in S \quad \texttt{fresh}\ d.\texttt{params} \quad d.\texttt{params} = clos; x_0; \ldots; x_n \\ FV_{n+1}(t) = v_1; \ldots; v_k \quad S, \Gamma \vdash v_i! \approx t'_i \quad (\text{pour tout } i \in [1;k]) \\ S, (0, \texttt{Var}\ x_0), \ldots, ((n+1), \texttt{Var}\ clos), (v_1, \texttt{Fld}\ 0\ clos), \ldots, (v_k, \texttt{Fld}\ (k-1)\ clos) \vdash t \approx d.\texttt{body}\end{array}}{S, \Gamma \vdash \mu^n.\ t \approx f\ (t'_1; \ldots; t'_k)}$$

Les liaisons Concernant les liaisons, la relation ne génère pas de nom frais. Elle vérifie que les variables qui vont être liées sont fraîches ; c'est-à-dire qu'elles n'étaient pas dans l'environnement de traduction. On ajoute alors une association dans l'environnement de traduction que l'on réactualise. Ce qui donne pour la liaison locale :

$$\frac{S, \Gamma \vdash t_1 \approx t'_1 \quad x \notin \Gamma \quad S, (0, \texttt{Var}\ x), \uparrow \Gamma \vdash t_2 \approx t'_2}{S, \Gamma \vdash \texttt{let}\ t_1\ \texttt{in}\ t_2 \approx \texttt{let}\ x\ =\ t'_1\ \texttt{in}\ t'_2}$$

Et pour les clauses de filtrage :

$$\frac{\forall x_i, x_i \notin \Gamma \quad S, (0, \texttt{Var}\ x_0), \ldots, (n-1, \texttt{Var}\ x_{n-1}), \uparrow^n \Gamma \vdash t \approx t'}{S, \Gamma \vdash n \to t \approx x_0; \ldots; x_{n-1} \to t'}$$

Pour les autres termes, les termes se correspondent si les sous-termes se correspondent également.

La figure 5.2.2 donne la relation entre les termes de nML et les termes Fml dans son intégralité.

5.3 Préservation sémantique

La preuve de préservation sémantique de notre construction de fermetures se décompose en deux points :

1) préservation sémantique de la relation $\approx$,

2) correction de cette relation vis-à-vis de la traduction.

Nous esquisserons la preuve de préservation sémantique de la forme relationnelle. Nous montrons alors la correction de la relation vis-à-vis de la traduction. Cette dernière contient une preuve de la monotonicité de l'état. (Il aurait été impardonnable de se reposer sur une simple observation, dans le cadre de nos travaux). Enfin, nous pourrons exhiber la preuve de préservation sémantique de notre construction de fermetures.

$$\frac{\Gamma(n) = \texttt{Var}\ x}{S, \Gamma \vdash n! \approx x} \qquad \frac{\Gamma(n) = \texttt{Fld}\ \ pos\ x}{S, \Gamma \vdash n! \approx \texttt{field}\ (pos + 1)\ x}$$

$$\frac{\begin{array}{c}(f, d) \in S \quad \texttt{fresh}\ d.\texttt{params} \quad d.\texttt{params} = clos; x_0; \ldots; x_n \\ FV_{n+1}(t) = v_1; \ldots; v_k \quad S, \Gamma \vdash v_i! \approx t'_i \quad (\text{pour tout } i \in [1; k]) \\ S, (0, \texttt{Var}\ x_0), \ldots, (n, \texttt{Var}\ x_n), (v_1, \texttt{Fld}\ 0\ clos), \ldots, (v_k, \texttt{Fld}\ (k-1)\ clos) \vdash t \approx d.\texttt{body}\end{array}}{S, \Gamma \vdash \lambda^n.\ t \approx f\ (t'_1; \ldots; t'_k)}$$

$$\frac{\begin{array}{c}(f, d) \in S \quad \texttt{fresh}\ d.\texttt{params} \quad d.\texttt{params} = clos; x_0; \ldots; x_n \\ FV_{n+1}(t) = v_1; \ldots; v_k \quad S, \Gamma \vdash v_i! \approx t'_i \quad (\text{pour tout } i \in [1; k]) \\ S, (0, \texttt{Var}\ x_0), \ldots, ((n+1), \texttt{Var}\ clos), (v_1, \texttt{Fld}\ 0\ clos), \ldots, (v_k, \texttt{Fld}\ (k-1)\ clos) \vdash t \approx d.\texttt{body}\end{array}}{S, \Gamma \vdash \mu^n.\ t \approx f\ (t'_1; \ldots; t'_k)}$$

$$\frac{S, \Gamma \vdash t_1 \approx t'_1 \quad x \notin \Gamma \quad S, (0, \texttt{Var}\ x), \uparrow \Gamma \vdash t_2 \approx t'_2}{S, \Gamma \vdash \texttt{let}\ t_1\ \texttt{in}\ t_2 \approx \texttt{let}\ x\ =\ t'_1\ \texttt{in}\ t'_2}$$

$$\frac{S, \Gamma \vdash t \approx t' \quad S, \Gamma \vdash t_i \approx t'_i \quad (\text{pour tout } i \in [0; n])}{S, \Gamma \vdash t\ [t_0; \ldots; t_n] \approx t'\ [t'_0; \ldots; t'_n]}$$

$$\frac{S, \Gamma \vdash t_i \approx t'_i \quad (\text{pour tout } i \in [0; n])}{S, \Gamma \vdash t\ [t_0; \ldots; t_n] \approx t'\ [t'_0; \ldots; t'_n]} \qquad \frac{S, \Gamma \vdash t_i \approx t'_i \quad (\text{pour tout } i \in [1; n])}{S, \Gamma \vdash C\ (t_1; \ldots; t_n) \approx C\ (t'_1; \ldots; t'_n)}$$

$$\frac{S, \Gamma \vdash t \approx t' \quad S, \Gamma \vdash \pi_i \approx \pi'_i \quad (\text{pour tout } i \in [0; n])}{S, \Gamma \vdash \texttt{match}\ t\ \texttt{with}\ \pi_1; \ldots; \pi_n \approx \texttt{match}\ t'\ \texttt{with}\ \pi'_1; \ldots; \pi'_n}$$

$$\frac{\forall x_i, x_i \notin \Gamma \quad S, (0, \texttt{Var}\ x_0), \ldots, (n-1, \texttt{Var}\ x_{n-1}), \uparrow^n \Gamma \vdash t \approx t'}{S, \Gamma \vdash n \to t \approx x_1; \ldots; x_{n-1} \to t'}$$

FIG. 5.2.2 – Présentation relationnelle du calcul des fermetures

5.3.1 Préservation sémantique pour la présentation relationnelle

Nous voulons montrer une équivalence observationnelle entre l'évaluation d'un terme nML et celle du terme correspondant Fml selon la relation définie plus haut (voir **??**). Les sémantiques de ces deux langages sont définies à grand pas avec environnement. Les valeurs sémantiques de chacun de ces langages sont différentes, mais il nous faut pouvoir les comparer. De même, les environnements d'évaluation sont constitués de ces valeurs. De plus, la traduction des variables et leur nommage induit une répartition différente entre ces deux environnements. Une relation entre les environnements d'évaluation est donc nécessaire à la comparaison des évaluations.

Une évaluation en Fml considère dans son environnement d'évaluation la liste de déclarations de fonctions du programme tout entier. Les relations de correspondances entre environnements d'évaluation seront donc paramétrées par la liste de déclarations de fonctions correspondant à la liste de déclarations du programme en cours d'évaluation.

Nous commençons par décrire la relation de correspondance entre valeurs sémantiques des deux langages. Puis nous présentons la mise en correspondance entre les environnements d'évaluation. Enfin, nous esquissons la preuve de correction de la construction de fermeture.

Relation de correspondance entre valeurs

Parmi les valeurs sémantiques, nous trouvons des valeurs fermetures. Leur présence induit le besoin de paramétrer toute correspondance entre les valeurs nML et Fml par la liste de déclarations de fonctions Fml, d'une part pour vérifier les fonctions qui se cachent derrière les noms de fonctions apparaissant dans les valeurs sémantiques et d'autre part, pour traiter les corps des abstractions. Pour qu'une valeur fermeture nML corresponde à une valeur fermeture Fml, il faut, entre autre, que les corps d'abstractions se correspondent. Cette correspondance sera la relation définie plus haut (voir la section 5.2.3).

$$\frac{S \vdash v_i \sim w_i \quad (\text{pour tout } i \in [1;n])}{S \vdash (C, v_1; \ldots; v_n) \sim (C, w_1; \ldots; w_n)}$$

$$\frac{\begin{array}{c} S \vdash (f,d)\ \texttt{wf} \quad |d.\texttt{params}| = (n+2) \quad v_1; \ldots; v_k = FV(t) \\ S \vdash e(v_i) \sim w_i \quad (\text{pour tout } i \in [1;k]) \quad S, (\ \texttt{mkenv}\ v_1; \ldots; v_k\ d) \vdash t \approx d.\texttt{body} \end{array}}{S \vdash (n,t,e) \sim (f, w_1; \ldots; w_k)}$$

$$\frac{\begin{array}{c} S \vdash (f,d)\ \texttt{wf} \quad |d.\texttt{params}| = (n+2) \quad v_1; \ldots; v_k = FV(t) \\ S \vdash e(v_i) \sim w_i \quad (\text{pour tout } i \in [1;k]) \quad t, (\ \texttt{mkenv\_rec}\ v_1; \ldots; v_k\ d) \vdash S \approx d.\texttt{body} \end{array}}{S \vdash (n,t,e)_{rec} \sim (f, w_1; \ldots; w_k)}$$

La correspondance entre valeurs sémantiques est fortement inspirée par la relation exprimant la construction de fermeture. Les valeurs de constructeurs appliqués se correspondent si elles portent le même numéro et que les valeurs de leurs arguments se correspondent.

Une valeur fermeture simple nML, (n,t,e) correspond à une valeur fermeture Fml, $(f, w_1; \ldots; w_k)$ si f correspond à une fonction bien formée d'arité $(n+2)$ (un paramètre supplémentaire pour la prise en compte de la fermeture). De plus, les valeurs dans e des variables libres dans t, $v_1; \ldots; v_k$ correspondent aux valeurs de la fermeture minimale Fml. Enfin, le corps de l'abstraction t est en relation avec le corps de la fonction Fml d dans l'environnement de traduction fabriqué par `mkenv` qui traite les paramètres de la fonction et les variables libres dans le corps de fonction.

La mise en correspondance d'une fermeture nML récursive est identique à ceci près que l'environnement de traduction considéré lors de la mise en relation du corps d'abstraction avec le corps de la fonction Fml est fabriqué par `mkenv_rec`.

Correspondance étendue aux environnements

La relation de correspondance entre valeurs sémantiques permet non seulement de déterminer si les valeurs produites par les évaluations nML et Fml sont équivalentes mais aussi de mettre en correspondance les environnements d'évaluation. Les variables nML ayant plusieurs destins possibles lors de la construction des fermetures, les consultations dans les

environnements source et cible sont différentes. En effet, nous distinguons, lors de la traduction d'un corps d'abstraction, les variables libres des variables liées. Cette distinction se fait par les informations `Var` x ou `Fld` n x. Elles sont transmises lors du traitement de tout un terme au travers d'un environnement de traduction Γ. Une information est ajoutée pour chaque liaison et est retirée à la fin de la portée de chaque liaison.

Selon l'information associée à un indice n, nous pouvons déterminer le moyen de consulter la valeur Fml correspondant à la traduction $n!$. Si cette information est `Var` x, alors la valeur de $e(n)$ correspond à la valeur de $e'(x)$, où e est l'environnement d'évaluation nML et e' celui du programme correspondant Fml. Par contre, si l'information est `Fld` n x, alors $e'(x)$ est une valeur sémantique $(f, v_1; \ldots; v_m)$ où v_n correspond à la valeur nML $e(n)$.

Nous définissons la consultation dans un environnement d'évaluation Fml e' par rapport à une information de traduction γ, noté $e'(\gamma)$, comme suit :

- si γ = `Var` x alors $e'(\gamma) = e'(x)$,
- si γ = `Fld` i x alors $e'(\gamma) = v_i$ avec $e'(x) = (f, v_1; \ldots; v_m)$.

$$\frac{e'(x) = v}{e'(\texttt{Var}\ x) = v} \qquad \frac{e'(x) = (f, v_1; \ldots; v_m) \quad i \leq m}{e'(\texttt{Fld}\ i\ x) = v_i}$$

Définition 5.3.1 (Correspondance entre environnements)
Soit l'environnement e d'évaluation nML, e' celui de Fml. $S, \Gamma \vdash e \sim e'$ si pour tout $(n, \gamma) \in \Gamma$, $S \vdash e(n) \sim e'(\gamma)$.

Nous pouvons maintenant énoncer le théorème de préservation sémantique de la relation de construction de fermetures.

Théorème 5.3.2 (Préservation sémantique de $\approx$)
Si le programme nML p s'évalue en la valeur v et

$$p'.\texttt{defs}, \varepsilon \vdash p.\texttt{main} \approx p'.\texttt{main},$$

alors il existe une valeur v' telle que, p' s'évalue en v' et

$$p'.\texttt{defs} \vdash v \ \sim \ v'.$$

Nous prouvons ce théorème par simulation sur une évaluation nML. Cette simulation s'énonce par le lemme suivant :

Lemme 5.3.3 (Simulation)
Supposons $S, \Gamma \vdash t \approx t'$ et $S, \Gamma \vdash e \sim e'$. Si

$$e \vdash t \Rightarrow v,$$

alors il existe v' telle que

$$S, e' \vdash t' \Rightarrow v' \text{ et } S \vdash v \sim v'.$$

Ce lemme se prouve par induction sur l'évaluation nML. Le point crucial est la préservation de correspondance entre environnements lors d'une évaluation. Par exemple, lors de l'évaluation du sous-terme droit d'une liaison locale. En effet, si nous avons en hypothèse

$S, \Gamma \vdash e \approx e'$, alors nous devons pouvoir construire la correspondance entre environnement $S, (v_1; e) \vdash ((O, \texttt{Var}\ id); \uparrow \Gamma))) \approx e'[x \leftarrow v'_1]$.

Considérons la règle d'évaluation de la liaison locale nML :

$$\frac{e \vdash t_1 \Rightarrow v_1 \qquad (v_1; e) \vdash t_2 \Rightarrow v}{e \vdash \texttt{let}\ t_1\ \texttt{in}\ t_2 \Rightarrow v}$$

La relation sur la liaison locale se définit :

$$\frac{S, \Gamma \vdash t_1 \approx t'_1 \qquad x \notin \Gamma \qquad S, ((0, \texttt{Var}\ x), \uparrow \Gamma) \vdash t_2 \approx t'_2}{S, \Gamma \vdash \texttt{let}\ t_1\ \texttt{in}\ t_2 \approx \texttt{let}\ x\ =\ t'_1\ \texttt{in}\ t'_2}$$

Nous voulons montrer : $\exists v,\ S, e' \vdash \texttt{let}\ x\ =\ t'_1\ \texttt{in}\ t'_2 \Rightarrow v'$ et $S \vdash v \sim v'$. Pour cela, nous sommes amenés à reconstruire les prémisses de la règle d'évaluation de la liaison locale Fml :

$$\frac{S, e' \vdash t'_1 \Rightarrow v'_1 \qquad S, e'[x \leftarrow v'_1] \vdash t'_2 \Rightarrow v'}{S, e' \vdash \texttt{let}\ x\ =\ t'_1\ \texttt{in}\ t'_2 \Rightarrow v'}$$

Il nous faut exhiber : $S, (v_1; e) \vdash ((0, \texttt{Var}\ x); \uparrow \Gamma)) \approx e'[x \leftarrow v'_1]$, en utilisant les hypothèses à notre disposition. Ce qui se traduit par le lemme suivant :

Lemme 5.3.4
Si $S, \Gamma \vdash e \sim e'$, $x \notin \Gamma$ *et* $S \vdash v \sim v'$ *tel que* $S, e' \vdash t' \Rightarrow v'$ *alors*

$$S, (v; e) \vdash ((O, \texttt{Var}\ x); \uparrow \Gamma) \approx e'[x \leftarrow v'].$$

Supposons que l'on est la correspondance entre environnement suivante $S, \Gamma \vdash e \sim e'$ et l'équivalence entre les valeurs $S \vdash v \sim v'$ où v' vient d'une évaluation dans l'environnement e' et la liste de définitions de fonctions Fml S. Alors, l'environnement augmenté par la nouvelle liaison $(v; e)$ correspond à l'environnement augmenté $e'[x \leftarrow v']$ sous l'environnement de traduction $((O, \texttt{Var}\ x); \uparrow \Gamma)$ à condition que x n'apparaisse pas dans Γ.

5.3.2 Correction de l'implantation fonctionnelle de la transformation

Nous n'avons fait que la moitié du chemin. En effet, seule la correction de la transformation nous intéresse. C'est elle qui fera partie du front-end certifié. Dans ce qui suit nous montrons la correction de la relation par la transformation. Autrement dit, si un programme nML, p est traduit en un programme Fml p' alors $p \approx p'$.

Pour cela, nous utilisons le caractère monotone de la relation. Plus précisément, si la relation est paramétrée par une liste de déclarations S, alors les correspondances de cette relation sont conservées si on considère une liste S', telle que $S \subseteq S'$. Nous préciserons les invariants de la preuve. En effet, il faut certaines conditions sur les états intermédiaires de la traduction pour répondre à la relation. Enfin, nous pourrons énoncer la correction de la relation.

Monotonicité

Lors de la traduction d'un sous-terme, nous pouvons conclure des faits sur la liste intermédiaire de déclarations de fonctions issue de cette traduction. Cependant, le même sous-terme dans la relation sera considéré sous la liste de déclarations finale. Or, le caractère monotone de l'évolution de l'état implique que les faits conclus sur une liste intermédiaire seront présents dans les faits sur la liste finale. Nous montrons ici que la mise en relation entre un terme nML et un terme Fml sous des informations contenues dans S reste inchangée même si ces informations sont enrichies : $(S \subseteq S')$.

Théorème 5.3.5 (Monotonicité de la relation)
Si $S, \Gamma \vdash t \approx t'$ *et* $S \subseteq S'$ *alors* $S', \Gamma \vdash t \approx t'$.

La preuve se fait par induction sur la définition de la relation.

Ce théorème sera utilisé dans la preuve de la correction, notamment concernant le cas des sous-termes. En effet, lors du traitement de `let` t_1 `in` t_2, on doit pouvoir généraliser les conclusions sur l'état intermédiaire après la traduction de t_1 à l'état final issu de la traduction du terme entier.

Invariants de la preuve

Nous voulons montrer le théorème suivant :

Théorème 5.3.6 (Correction de la relation par la traduction)

$$[\![p]\!].\texttt{defs}, \varepsilon \vdash p \approx [\![p]\!].$$

Ce théorème se prouve par induction sur la relation. Avant d'énoncer le lemme de correction pour l'induction, nous nous attardons sur les invariants de la preuve. Ces invariants seront pris en hypothèses de la correction et reconstruits par les conclusions de la correction. Les invariants ont, ici, la charge de décrire un état correct de la monade lorsqu'il est considéré comme état initial de la traduction.

Considérons :

$$\texttt{trans}\ t\ S_i\ x_i\ f_i = \texttt{Ok}\ t'\ S_f\ x_f\ f_f$$

$S_f,\ x_f,\ f_f$ est un état correct de monade s'il satisfait les propriétés suivantes :

Unicité des noms de fonctions (H_1) : Les noms de fonctions de la liste de déclarations doivent être distincts deux à deux :
$\forall f_0,\ f_1, (f_0, d) \in S_i$ et $(f_1, d') \in S_i \rightarrow f_0 \neq f_1$.

Cohérence des noms de fonctions vis-à-vis de leur génération (H_2) : Ces noms doivent être correctes par rapport au générateur de noms de fonctions. Un nom de fonction f est correct par rapport au générateur de noms de fonction f_i si $f < f_i$:
$\forall f, f \geq f_i \rightarrow f \notin Dom(S_i)$.

Cohérence des noms de variables vis-à-vis de leur génération (H_3) : Enfin, l'environnement de traduction doit être correct par rapport au générateur de noms de

variables. Un environnement Γ est correct par rapport au générateur de noms de variables x_i si toute variable apparaissant dans Γ est correcte vis-à-vis de x_i :
$\forall x, x > x_i \rightarrow \Gamma(x) = \perp$.

En plus de montrer la correction de la relation par la traduction, cette preuve montre aussi la monotonicité de l'état lors de la traduction. Les conclusions de cette correction permettent, en plus de conclure la correction de reconstituer les invariants pour les applications récursives, mais aussi de comparer les états intermédiaires entre eux.
Ainsi, nous concluons que nous avons les propriétés suivantes :

Monotonicité des génération de noms (C_1) : les générateurs de noms des états issus d'une traduction sont plus grands ou les mêmes que ceux de l'état initial de cette traduction :

$$x_i \leq x_f \text{ et } f_i \leq f_f.$$

Monotonicité de la liste de déclarations en construction C_2 : De plus, la liste initiale de déclarations est incluse dans la liste finale :

$$S_i \subseteq S_f.$$

Cohérence des noms de fonctions (C_3) : La liste finale est correcte par rapport au générateur de noms de fonctions et ne comporte pas de répétition :
$\forall f, (f, d) \in S_i \rightarrow (f, d) \in S_f$, $\forall f,\ f \geq f_f \rightarrow f \notin Dom(S_f)$ et $\forall f_0\ f_1,\ (f_0, d) \in S_f$ et $(f_1, d') \in S_f \rightarrow f_0 \neq f_1$.

Lemme 5.3.7 (Correction de la relation par la traduction)
Considérons :

$$\texttt{trans}\ t\ x_i\ f_i\ S_i\ \Gamma = \texttt{Ok}\ t'\ x_f\ f_f\ S_f.$$

Supposons :

H_1 $\forall f_0,\ f_1, (f_0, d) \in S_i$ *et* $(f_1, d') \in S_i \rightarrow f_0 \neq f_1$,

H_2 $\forall f, f \geq f_i \rightarrow f \notin Dom(S_i)$,

H_3 $\forall x, x \geq x_i \rightarrow \Gamma(x) \neq \perp$.

Alors,

$$S_f, \Gamma \vdash t \approx t'$$

et

C_1 $x_i \leq x_f$, $f_i \leq f_f$,

C_2 $S_i \subseteq S_f$,

C_3 $\forall f, (f, d) \in S_i \rightarrow (f, d) \in S_f$.

5.3.3 Préservation sémantique de la construction de fermetures

Nous pouvons maintenant énoncer le théorème de préservation sémantique de la construction de fermetures :

Théorème 5.3.8 (Préservation sémantique de la construction de fermetures)
Si $\vdash p \Rightarrow v$ *alors il existe* v' *telle que* $\vdash [\![p]\!] \Rightarrow v'$ *et* $S \vdash v \sim v'$ *avec* $S = [\![p]\!].\texttt{defs}$.

La preuve se fait par combinaison des deux théorèmes 5.3.6 et 5.3.2.

5.4 Conclusions et perspectives

Dans ce chapitre, nous avons explicité les fermetures afin d'obtenir un programme organisé en fonctions globales. Cette explicitation s'est faite au travers d'une transformation de nML vers Fml. Au cours de la transformation d'un programme nML, nous construisons la liste de fonctions globales Fml. De plus, nous profitons de cette transformation afin de nommer les variables de manière unique. Cet algorithme a de forts traits impératifs. Or, Gallina est un langage purement fonctionnel. Pour pallier les traits impératifs de cette transformation, nous avons utilisé une monade d'état pour avoir une version fonctionnelle de l'algorithme. Au cours de la transformation, la liste de fonctions globales se construit au travers de l'état de la monade. Or, pour la preuve de préservation sémantique, nous avons besoin de raisonner sur l'évaluation des termes qui considère la liste finale de fonction (celle de l'état final). Pour les besoins de la preuve, nous avons défini une spécification relationnelle de l'explicitation des fermetures. Cette relation ne construit pas d'état mais vérifie que dans l'état final un terme nML correspond à un terme Fml.

Explicitation des fermetures pour les abstractions mutuellement récursives

Notre langage source (section 2.2) présente une lacune vis-à-vis du langage de l'assistant de preuve Coq. Une partie des développements menés dans l'assistant de preuve Coq, notamment ceux autour des langages de programmation et leurs spécifications, utilisent des structures de données mutuellement récursives. De telles structures s'implantent naturellement dans notre langage puisqu'il n'est pas typé. Cependant, les fonctions mutuellement récursives (que nous employons dans ce développement) ne font pas partie de εML. Leur étude s'inscrit naturellement dans le prolongement de nos travaux présentés dans ce manuscrit.

Une solution simple serait de transformer un ensemble de fonctions mutuellement récursives en une imbrication de fonctions récursives simples (lemme de Besik). Cette transformation est coûteuse en nombre de fermetures et augmente le nombre de paramètres formels de pratiquement toutes les fonctions. De plus, les applications doivent être transformées en conséquence.

La représentation d'abstractions mutuellement récursives dans un formalisme à la de Bruijn est assez simple. On peut par exemple utiliser un indiçage en couple, le premier élément de ce couple serait l'indice de de Bruijn correspondant au `letrec` et le deuxième un niveau de de Bruijn pour chaque fonction mutuellement récursive.

Les problèmes apparaissent plutôt lors de l'explicitation des fermetures et plus précisément concernant la représentation mémoire que l'on veut donner à de telles fermetures lors de la génération de code Cminor. Plusieurs choix s'offrent à nous.

Une première représentation serait une fermeture par fonction (mutuellement) récursive, les autres fermetures des autres fonctions mutuelles seraient pointées dans l'environnement de la fermeture. Chaque fermeture d'un ensemble de fonctions mutuellement récursives pointerait vers le code de fonction concernée. La figure 5.4.1 illustre cette représentation pour deux fonctions mutuellement récursives. Cette représentation serait coûteuse.

On pourrait raffiner en représentant un ensemble de fonctions mutuelles par une seule fermeture. Pour cela, on choisirait un représentant de l'ensemble pour le placer en tête de fermeture. Les autres fonctions de l'ensemble de fonctions mutuelles seraient placées dans les emplacements suivants. Ces emplacements pointeraient sur les codes de fonctions. Cette

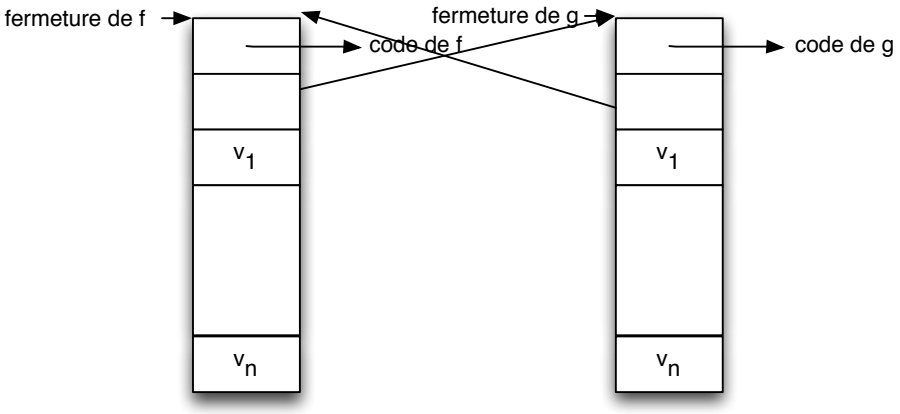


FIG. 5.4.1 – Représentation deux fermetures de fonctions mutuellement récursives

représentation a été inventée par A. Appel [2]. Par exemple les deux fonctions mutuellement récursives g et f ont en commun toutes les variables libres dans leur corps sauf g et f. Notons cet ensemble de variables $v_1; \ldots; v_n$. Sur la figure 5.4.2 on a représenté la fermeture de ces deux fonctions. La fermeture de f commence dans le premier emplacement, qui contient un pointeur vers le code de f. Le second emplacement marque le début de la fermeture de g, et contient un pointeur vers le code de g. Suivent ensuite les variables libres $v_1; \ldots; v_n$. Dans le corps de f, g devient `Field` 1 *clos* et dans le corps de g, f devient `Field` (-1) *clos*.

Ce choix de représentation est soumis à plusieurs critères. La représentation ne doit pas rendre inefficace le code généré et elle doit se prêter à la spécification du comportement d'un gestionnaire de mémoire sur la configuration mémoire d'un programme généré par notre compilation (voir la section 7.4). Nous envisageons d'approfondir notre réflexion sur le choix de la représentation des fonctions mutuellement récursives, afin de traiter un langage source plus expressif.

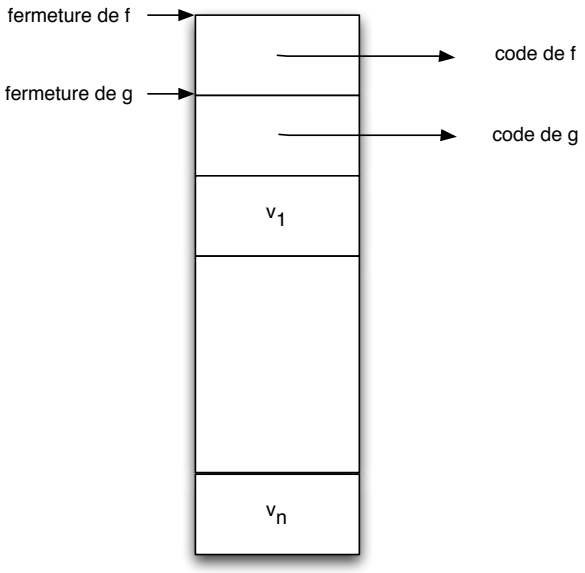


FIG. 5.4.2 – Représentation d'une fermeture unique pour 2 fonctions mutuellement récursives

6 Interaction avec un gestionnaire de mémoire

Lors de la compilation d'un langage, les données sont stockées en mémoire. L'espace mémoire peut ne pas suffire à l'enregistrement de toutes les données d'un programme lors de son exécution. Néanmoins, toutes les données allouées lors de la compilation d'un programme ne sont pas toutes utiles jusqu'à la fin de l'exécution du programme. Cet espace inutile peut être recyclé afin d'allouer de nouvelles données. Il est donc nécessaire de gérer l'espace mémoire. Dans les langages de bas niveau, comme `C` ou `C++`, le programmeur peut lui-même gérer la mémoire en désallouant les blocs dont il n'a plus l'utilité. On parle alors de désallocation explicite. Ce mécanisme n'est pas sûr, le programmeur pouvant introduire par mégarde des erreurs telles que libérer plusieurs fois le même pointeur ou bien libérer trop tôt un pointeur.

Dans les langages de haut niveau, les structures de données sont plus abstraites et la mémoire n'est pas directement manipulable. Dans la compilation de langages de haut niveau en général, de langages fonctionnels en particulier, la gestion de mémoire est dédiée à un *gestionnaire automatique de mémoire*. Il s'agit d'un programme complémentaire du compilateur dont la tâche est de gérer les allocations des données et la libération de l'espace mémoire devenu inutile. Dans l'optique de développer un compilateur pour miniML à la fois réaliste et sûr, nous avons mis en place et vérifié formellement une interaction avec un gestionnaire de mémoire qui serait développé en Cminor. Le présent chapitre décrit la mise en place de cette interaction et sa vérification formelle au travers de deux passes de compilation mettant en jeu deux langages purement fonctionnels.

6.1 Gestion automatique de mémoire

La gestion automatique de mémoire est un domaine à part entière de la compilation. Un état de l'art exhaustif est présenté dans [117].

6.1.1 Approches pour la gestion automatique de mémoire

Il existe principalement trois approches pour la gestion automatique de mémoire :

Le compte de références [19] A chaque objet, on associe un compte de références représentant le nombre de pointeurs sur cet objet dans le programme. Ce compte est mis à jour à chaque création d'un pointeur sur l'objet (le compteur est alors incrémenté de 1) et à l'élimination d'un pointeur (le compteur est alors décrémenté de 1). Bien entendu la mise à jour du compteur d'un objet o entraîne la mise à jour des compteurs des objets sur lesquels o réfère. Lorsque le compteur d'un objet est à 0 l'objet devient

libérable, son espace peut être récupéré. Le compte de références présente un problème majeur : sa mauvaise gestion des références cycliques. Si deux objets se réfèrent mutuellement sans avoir d'autre référence, les compteurs de chacun de ces objets sont à 1 tandis qu'ils sont inaccessibles par programme.

L'analyse statique de durée de vie des objets se combine avec une désallocation insérée par le compilateur. Lorsque l'analyse indique qu'un objet est mort, il peut être libéré, le compilateur insère alors dans le code généré une opération de désallocation explicite. Afin d'être plus efficace, l'analyse statique de durée de vie est généralement combinée avec une gestion par régions de la mémoire [112, 111]. La gestion de mémoire par régions permet de libérer très rapidement tous les objets alloués dans une même région, on parle de " désallocation en masse". L'analyse statique de durée de vie des objets présente toutefois un inconvénient, ces analyses restent imprécises.

L'utilisation d'un *Glaneur de Cellules* (Garbage Collector) ou GC. Le GC est appelé par le gestionnaire de mémoire lorsqu'une allocation réclame plus d'espace que de disponible. Le GC inspecte alors la mémoire et libère l'espace dont le programme n'a plus utilité. Afin de connaître l'espace utile au programme, ce dernier précise au GC les données dont il a encore besoin. Ces données sont appelées *racines*, il s'agit des variables globales et locales du programme. A partir des racines, le GC, en suivant les pointeurs, détermine les objets accessibles depuis les racines, c'est-à-dire tous les objets pour lesquels il exist un chemin de pointeurs depuis l'ensemble des racines. Le GC ne libère ni l'espace occupé par les racines ni celui occupé par les objets accessibles depuis des racines.

Dans notre développement, nous choisissons l'approche de gestion automatique de mémoire via un glaneur de cellules.

6.1.2 Algorithmes de GC

Il existe plusieurs variantes de GC.

Le marquage-balayage [80] (mark&sweep) : les objets accessibles sont marqués lors d'un parcours (profondeur ou largeur d'abord) du graphe des pointeurs du programme. La phase de marquage est suivie par une phase de balayage : on parcourt la mémoire afin de libérer les espaces correspondant aux objets non marqués devenus inutiles à l'exécution du programme. Le souci est que les espaces libérés constituent des espaces de taille restreinte disposés de manière non contiguë dans l'espace mémoire. On aimerait avoir un unique espace libre de la taille de tous les objets ayant été libérés. Considérons la configuration du tas décrite dans la figure 6.1.1.

La figure 6.1.2 schématise l'algorithme de marquage-balayage sur la configuration de la figure 6.1.1. Le GC commence par marquer les éléments accessibles depuis les racines, ici les deux premiers mots sont les racines. Sur la deuxième configuration, seuls 4 et le pointeur vers 5 ne sont pas marqués, ils ne sont pas accessibles. La deuxième phase les libère, on obtient alors la troisième configuration. La troisième configuration n'est pas optimale pour pouvoir allouer des données. En effet, même si la somme d'espace disponible suffit à allouer une donnée de taille n, on ne dispose pas forcément d'un espace contigu de taille n.

Le marquage-compactage (mark&compact) permet d'obtenir cela. Comme décrit sur la figure 6.1.3, après la phase de marquage, identique à celle du marquage-balayage, suit

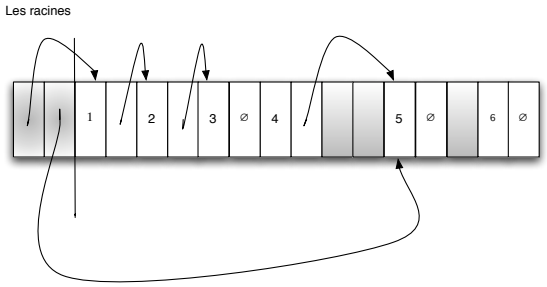


FIG. 6.1.1 – Exemple de configuration du tas

une phase de compactage qui regroupe les objets accessibles afin d'obtenir un espace libre contigu.

Le GC *à copie* (stop&copy) [37] libère un seul espace contigu en une seule phase. Le tas est scindé en deux espaces. Le premier est le tas effectif tandis que le second sert de zone de copie. Lorsque le GC se déclenche, il traverse la première zone et copie les objets accessibles dans la seconde zone. Les rôles de deux espaces sont alors intervertis. L'algorithme de GC à copie est décrit sur la figure 6.1.4. Notons toutefois, que le GC à copie présente un inconvénient : seule une moitié de l'espace du tas est utilisée pour l'enregistrement des données.

Les différents schémas peuvent se combiner dans les GC à génération. C'est le cas par exemple dans le système Objective Caml, sur le tas dit "mineur" le GC suit une stratégie à copie tandis que sur le tas dit "majeur" il suit une stratégie de marquage-balayage.

Dans le cadre de notre développement, le gestionnaire automatique de mémoire utilisera un GC. Le développement de ce GC, et plus généralement du gestionnaire de mémoire, n'est pas le sujet de notre étude. Des travaux en cours menés par Tahina Ramamanandro [98] et ceux de Andrew McCreight [81] sont dédiés à la conception et la certification de gestionnaire de mémoire. Nos travaux se concentrent sur la conception et la vérification formelle d'une interaction d'un tel gestionnaire de mémoire. Cette interaction intervient au niveau du code Cminor généré. Nous nous sommes intéressés à la détermination et la transmission au GC des racines via des langages intermédiaires fonctionnels.

6.1.3 Détermination des racines du GC

Concernant la troisième approche de gestion automatique de mémoire, il apparaît un trait commun à toutes les variantes de GC : la nécessité qu'il connaisse à son exécution l'ensemble des racines mémoire du programme. L'ensemble des racines mémoire d'un programme peut être déterminé de plusieurs manières :

Détermination des racines de manière exacte : avec la coopération du back-end. En effet, lors du processus de compilation, les données ont pu être stockées d'une manière différente à celle intuitive pour le programmeur : certains dans les registres du processeur, d'autres dans des emplacements de pile. Par ailleurs, sur la pile, elles sont

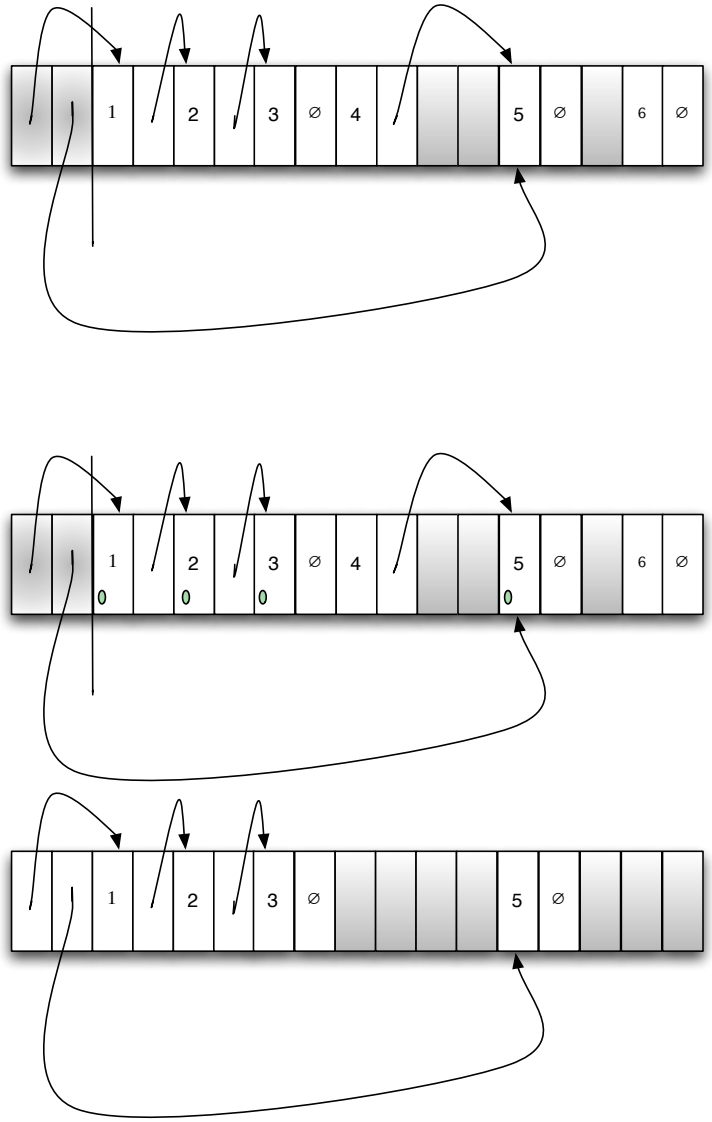


FIG. 6.1.2 – Algorithme de marquage-balayage

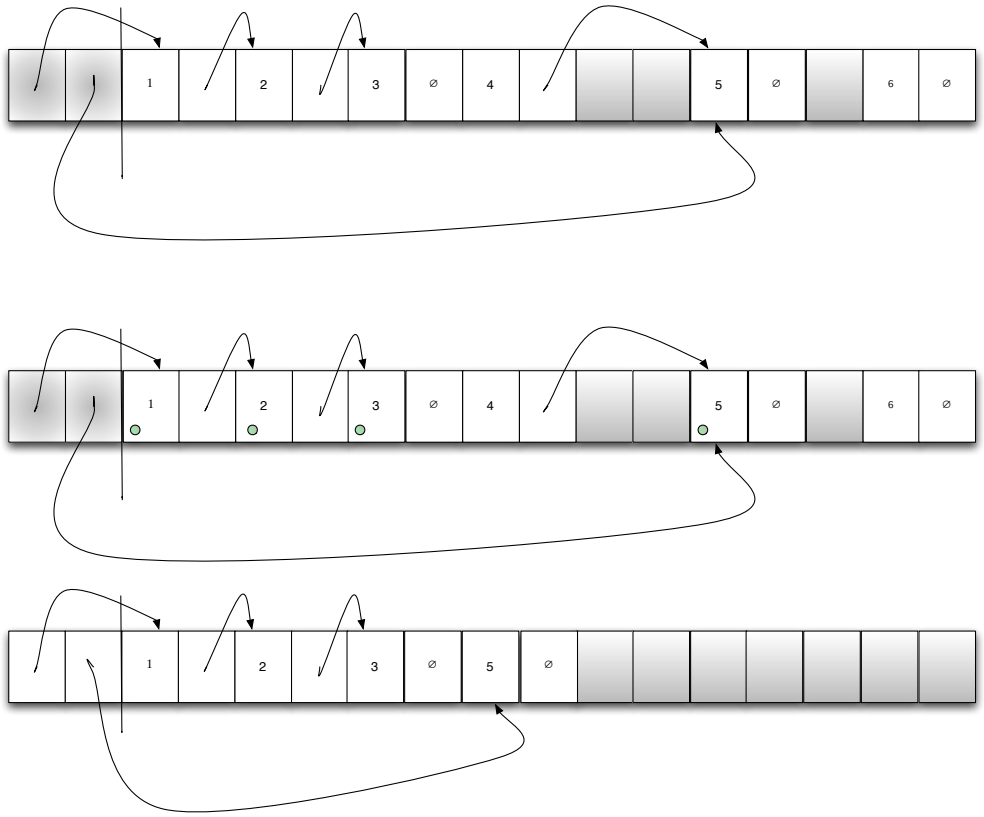


FIG. 6.1.3 – Algorithme de marquage-compactage

entrelacées d'autres composantes nécessaires à la compilation comme les adresses de retour des fonctions. La figure 6.1.5 confronte la vue d'un programmeur avec celle du code compilé. Par exemple, Peyton Jones, Ramsey et Reig proposent dans [93] un back-end généraliste muni d'un environnement d'exécution apte à coopérer avec l'environnement d'exécution du front end.

Détermination des racines de manière approchée : les GC *conservatifs*, comme décrit dans [14, 13], éliminent la coopération entre le programme et le gestionnaire automatique de mémoire. Le concept est de parcourir la pile et le tas en considérant tout ce qui ressemble à un pointeur sur le tas comme une racine. C'est bien entendu une sur-approximation trop forte. On peut raffiner en utilisant des heuristiques. Cette approche semble donc difficile à vérifier formellement. En particulier, certaines optimisations de compilation invalident les GC conservatifs : certaines racines ne sont pas détectées lors du parcours.

Détermination des racines par enregistrement explicite : via du code ajouté au programme soit manuellement soit automatiquement par le compilateur. Ce code addi-

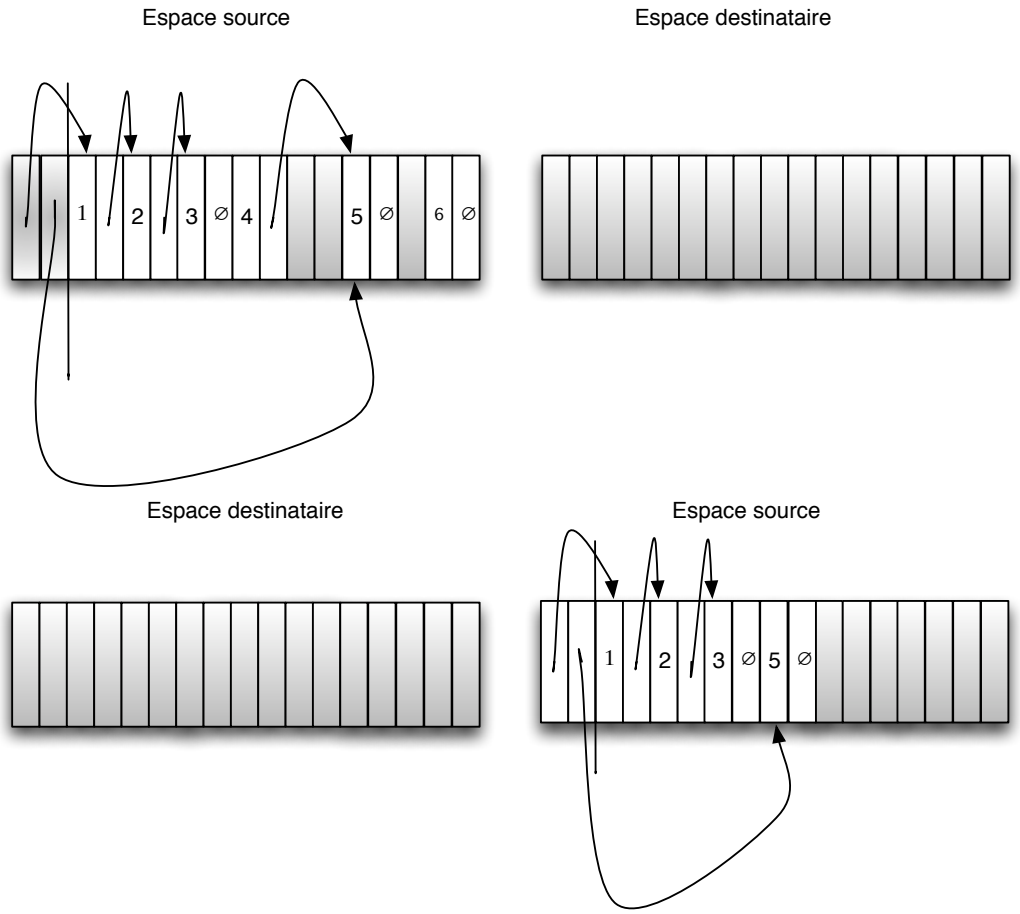


FIG. 6.1.4 – Algorithme de GC à copie

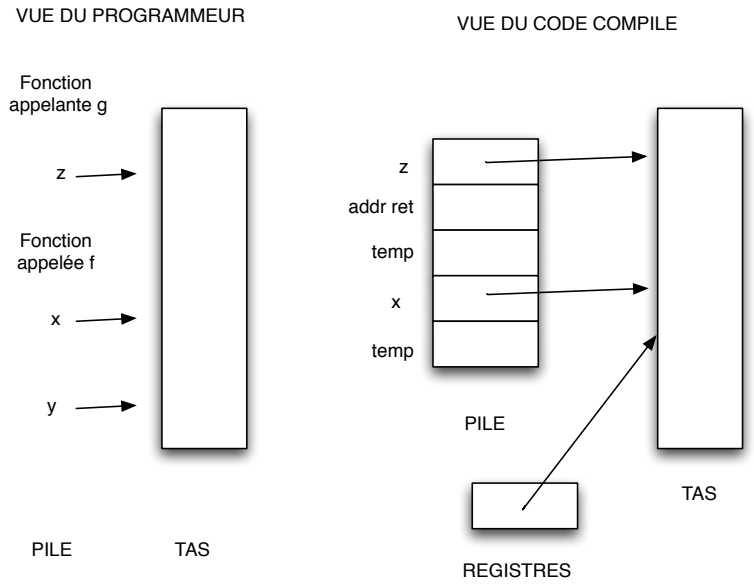


FIG. 6.1.5 – La vue du programmeur versus celle du code compilé

tionnel consiste à enregistrer les racines dans *une structure de données dédiée*. Cette structure de données dédiée constitue le vecteur de transmission entre le programme et le GC. Henderson dans [52] propose une automatisation de cette méthode, qui est connue maintenant sous le nom de *shadow stack*. La figure 6.1.6 illustre un enregistrement à la Henderson. La structure de données dédiée à l'enregistrement des racines est une liste simplement chaînée allouée sur la pile.

Cette méthode est plus sûre que l'utilisation d'un GC conservatif. De plus, elle ne demande aucune coopération avec le back-end.

Notre stratégie d'enregistrement des racines suit l'approche d'Henderson. La structure dédiée à l'enregistrement des racines est une liste simplement chaînée allouée sur les blocs de pile Cminor. En Cminor (voir la section 2.3), chaque appel de fonction crée un bloc de pile. Ce bloc est directement manipulable syntaxiquement. On tire avantage de ces blocs de pile Cminor afin de transmettre les racines au GC. Pour cela, les racines à transmettre lors de l'exécution d'une fonction sont enregistrées dans une liste simplement chaînée sur la pile courante de cet appel. Les racines des fonctions appelantes doivent aussi être transmises. Leur transmission se fait aussi par chaînage. La liste de racines sur le bloc de pile Cminor courant est chaînée à la liste de racines de la fonction appelante (et ceci d'appelés à appelants, comme une pseudo-pile d'appels).

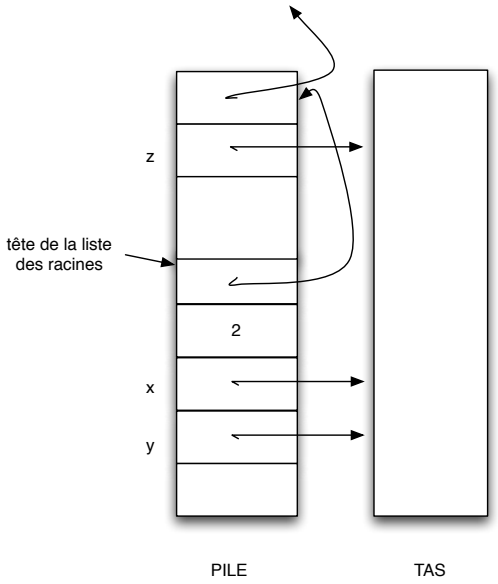


FIG. 6.1.6 – Illustration de l'enregistrement à la Henderson

6.1.4 Stratégies d'enregistrement explicites des racines

Comme dit plus haut, nous suivons une approche similaire à celle d'Henderson [52] : les racines seront enregistrées de manière explicite. Nous disposons de différentes manières d'enregistrer les racines.

Enregistrement à chaque appel potentiel au GC : Une première stratégie d'enregistrement des racines consiste à relever (calculer) les racines avant chaque *déclenchement potentiel du GC*. Un déclencheur potentiel de GC est une future allocation ou un appel de fonction. A chaque site de déclenchement potentiel de GC, on sauvegarde l'ensemble des racines courant. Puis on calcule les racines pour ce déclenchement de GC et on les place dans le nouvel ensemble de racines. Après évaluation du terme on restaure l'ensemble de racines précédant son exécution. Bien que produisant un ensemble de racines très précis, pas trop surestimé, cette approche est coûteuse, notamment en écritures inutiles. Considérons l'exemple suivant :

```
let z= t in
  let y = C(z,x) in
          f (a,b) ... z
```

Si l'on s'intéresse à la variable z, elle est enregistrée une première fois comme racine au niveau de l'allocation `C(z,x)`, et elle est enregistrée une deuxième fois au niveau de l'application de `f`.

A la déclaration d'une variable : Au lieu de calculer l'ensemble des racines à chaque potentiel déclenchement de GC, nous proposons de décider à chaque liaison de variable locale si cette variable sera racine d'au moins un sous-terme potentiellement déclencheur de GC dans sa portée lexicale.

L'ensemble des racines est mis à jour à chaque liaison de variable. Une variable qui sera enregistrée comme une racine, peut être distinguée des autres variables et être directement placée dans un élément de la structure d'enregistrement $root(i)$, si c'est le i-ème élément. En effet, elle est enregistrée durant toute sa portée lexicale.

Ce choix d'enregistrement ne demande pas une modification de la structure de données dédiée à l'enregistrement des racines à chaque déclencheur potentiel du GC. (Il n'y a pas de copie contrairement à la méthode précédente.) De plus, c'est ainsi que nous matérialiserons explicitement l'ensemble des racines dans notre chaîne de compilation. Les racines sont distinguées des autres variables.

Durant toute sa portée lexicale, une racine est associée à un seul emplacement dans cette structure. Cependant, l'ensemble des racines n'est pas optimal pour chaque potentiel déclenchement du GC. En effet, une racine peut ne plus être utile plus ou moins tôt dans sa portée lexicale. Ici, si elle est une future racine, elle reste stockée dans la structure dédiée à l'enregistrement des racines, jusqu'à la fin de sa portée lexicale.

6.1.5 Mise en œuvre de l'interaction avec un gestionnaire de mémoire

Dans ce chapitre, nous nous intéressons à mettre en oeuvre l'interaction avec un gestionnaire automatique de mémoire contenant un GC, avec détermination des racines par enregistrement explicite, au travers de langages intermédiaires dédiés. La figure 6.1.7, schématise grossièrement le code Cminor généré que l'on désire obtenir. Nous allons expliquer

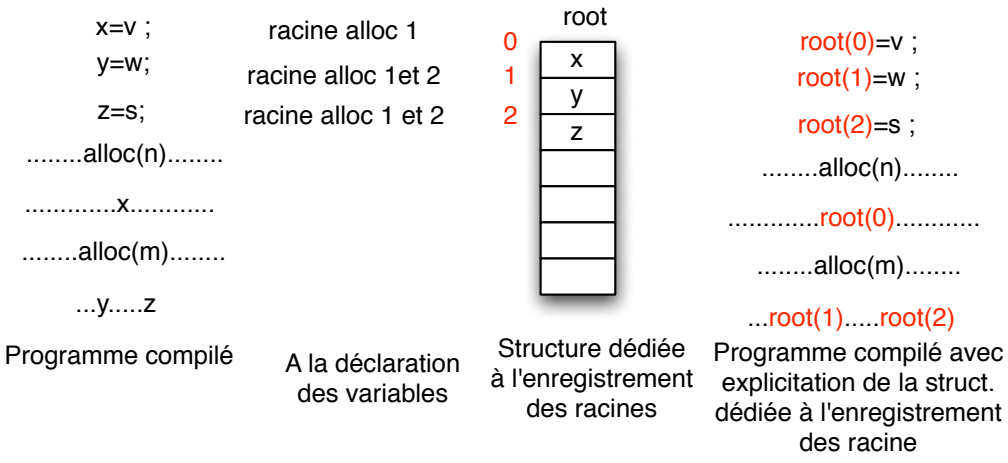


FIG. 6.1.7 – Squelette de l'interaction sur le code Cminor compilé

comment, au travers de deux passes de compilation, nous mettons en place cette interaction. Ces deux passes sont deux transformations consécutives vers deux langages intermédiaires. L'originalité est l'utilisation de deux langages intermédiaires purement fonctionnels pour le calcul et l'enregistrement explicite des racines. Le langage Fml du chapitre précédent 5 est un langage à la `ML` en style direct. Dans un premier temps, nous recherchons à mettre nos programme Fml dans une grammaire plus adéquate aux calculs des racines. Pour cela, nous utilisons la *forme intermédiaire monadique*, une forme faible de CPS. Puis, nous décrirons un langage intermédiaire dédié à l'enregistrement des racines, Fminor, dans lequel la structure de données dédiée à l'enregistrement des racines est explicite. Elle est manipulable via la syntaxe. Cette structure dédiée deviendra une liste simplement chaînée en Cminor qui sera sur les blocs de pile Cminor lors de la génération de code Cminor au chapitre 7. Les définitions syntaxique et sémantique de Fminor nous permettent de spécifier l'interaction avec un gestionnaire de mémoire. L'avantage est d'obtenir la vérification formelle de cette interaction, gratuitement, au travers de la propriété de préservation sémantique de la transformation en Fminor.

6.2 À la recherche de nos racines

Nous cherchons, ici, à détecter facilement l'ensemble des racines d'un calcul. Revenons sur la définition d'une racine : une racine est un pointeur qui doit survivre au garbage collector. Un pointeur r est une racine d'un calcul c, s'il est présent avant c, *ie.* s'il a été alloué avant le début du calcul c et qu'il est nécessaire après le calcul c. On voudrait avoir un langage intermédiaire, où tous les futurs pointeurs peuvent être détectés syntaxiquement.

6.2.1 A la recherche d'un langage propice au calcul des racines

Le langage Fml présenté au chapitre 5 n'est pas adéquat pour cette détection. Considérons le calcul $f\ [x]\ ::\ g\ [y]$, l'application du constructeur de liste `cons` aux arguments $f\ [x]$ et $g\ [y]$. Ce terme construit une nouvelle liste dont le premier élément est la résultante de l'application de la fonction f à la variable x et est mis en tête de la liste obtenue par l'application de la fonction g à la variable y.

Nous suivons une stratégie d'évaluation par appel par valeur avec évaluation des sous-termes de gauche à droite. Le premier sous-terme évalué est donc $f\ [x]$. Lors de l'évaluation du corps de la fonction f, il peut y avoir des allocations dans le tas. Ces allocations sont susceptibles de déclencher un GC. Dans le reste du calcul, la valeur associée à la variable y est nécessaire, elle est donc racine du calcul $f\ [x]$. Supposons que $f\ [x]$ s'évalue en la valeur v_f.

Il vient ensuite l'évaluation de $g\ [y]$ qui, pour les mêmes raisons, peut déclencher le GC. Dans la suite du calcul, la valeur v_f est nécessaire car elle est argument du constructeur `cons`. Elle est donc racine de $g\ [y]$. Supposons que $g\ [y]$ s'évalue en v_g.

Enfin, il faut allouer dans le tas un bloc pour représenter le constructeur `cons` appliqué à ses deux arguments. Bien entendu v_f et v_g sont des racines de cette allocation, puisqu'elles seront écrites dans le bloc alloué.

Il est difficile de prévoir statiquement, par simple analyse syntaxique les deux racines v_f

et v_g car elles n'apparaissent que lors de l'évaluation. Plus généralement, les calculs intermédiaires de Fml ne sont pas facilement manipulables syntaxiquement. Nous recherchons donc une forme ou un style de Fml où tous les calculs intermédiaires seraient facilement identifiables et manipulables par une analyse syntaxique.

Un premier candidat coûteux en nombre d'allocations : la forme CPS

Une racine peut être le fruit d'un calcul intermédiaire. Un bon moyen d'obtenir une forme de Fml où tous les calculs intermédiaires seraient distincts syntaxiquement et nommés serait de mettre nos programmes en style CPS. Comme nous l'avons vu au chapitre 4, l'une des propriétés importantes du style CPS est l'élémentarisation des calculs et le nommage de chaque calcul intermédiaire, via les paramètres de continuation. Notre exemple deviendrait :

$$\lambda k.\ f(x)(\lambda m.\ g(y)(\lambda n.\ k(m :: n))).$$

Ainsi, lors de l'allocation correspondant au constructeur appliqué :: les valeurs v_f et v_g sont portées par les variables m et n respectivement.

Mettre nos programmes en style CPS serait une manière d'obtenir une forme plus adéquate au calcul des racines de nos programmes. Cependant, il reste un inconvénient de taille. En effet, mettre un terme en style CPS implique une création d'un nombre important de fermetures. On ajoute au moins autant de fermetures que de calculs intermédiaires : chaque continuation nécessite la construction d'une fermeture. Or, chaque fermeture sera représentée par un bloc dans le tas et donc on ajouterait autant d'allocation dans le tas, ce qui serait contre-productif.

Un second candidat coûteux en taille de code : la forme A-normale

Nous aimerions garder l'élémentarisation des calculs intermédiaires et leur nommage, sans augmenter le nombre de fermetures. La forme A-normale (ANF) [40], aussi appelée forme monadique normale, est une forme dérivée du style CPS dont les propriétés communes sont exactement celles que nous désirons. Les calculs intermédiaires y sont élémentarisés et nommés par liaisons locales `let`. *A contrario* de la forme CPS, la forme A-normale n'introduit aucune augmentation du nombre de fermetures.

Cette élémentarisation des calculs se fait en hiérarchisant les termes selon la grammaire suivante :

Atomes :	$a ::= x$		variable
	$\mid$ `field` $n\ a$		champ d'une fermeture
Calculs :	$c\ = a$		atome
	$\mid a\ [a_0; \ldots; a_n]$		application $n-$aire
	$\mid f\ (a_1; \ldots; a_n)$		fermeture
	$\mid C\ (a_1; \ldots; a_n)$		constructeur appliqué
Termes :	$t ::= c$		calcul
	$\mid$ `let` $x = c$ `in` t		liaison locale
	$\mid$ `match` a `with` $\pi_0; \ldots; \pi_n$		filtrage

Motifs : $\pi ::= x_1; \ldots; x_n \rightarrow t$ clause de filtrage

Cette hiérarchisation repose sur les caractéristiques des termes :

Les atomes a **:** les variables et champs de fermeture ne correspondent pas à de futures allocations en mémoire. Les atomes constituent des consultations des environnements d'évaluation.

Les calculs c **:** en plus des atomes, ils xscontiennent les termes qui correspondront à de futures allocations dans le tas, ce sont les termes déclencheurs potentiels de GC directs. Les sous-termes des calculs sont des atomes, ce qui reflète l'élémentarisation des calculs. Toute écriture sur un bloc alloué correspondra à une valeur nommée par une variable.

Les termes t **:** en plus des calculs, contiennent les termes structurels. On remarque que c'est la seule catégorie syntaxique récursive. En effet, le corps des clauses sont des termes ainsi que les sous-termes droits des liaisons locales. Soulignons le calcul c apparaissant comme membre gauche de la liaison locale.

Ainsi, mis en forme A-normale, $f\ [x] :: g\ [y]$ devient :

$$\texttt{let}\ x_0 = f\ [x]\ \texttt{in let}\ x_1 = g\ [y]\ \texttt{in}\ x_0 :: x_1.$$

Cependant, la forme ANF présente un inconvénient majeur, la duplication de code. En effet, la mise en forme ANF d'un terme ne peut se faire par une simple transformation récursive. Considérons le terme $f\ [g\ [x]] :: g\ [y]$. La forme ANF du terme $f\ [g\ [x]]$ est $\texttt{let}\ x_0 = g\ [x]\ \texttt{in}\ f\ [x_0]$, et si l'on appliquait la transformation récursivement, on obtientrait :

$$\texttt{let}\ x_1 = (\texttt{let}\ x_0 = g\ [x]\ \texttt{in}\ f\ [x_0])\ \texttt{in let}\ x_2 = g\ [y]\ \texttt{in}\ x_1 :: x_2.$$

Ce terme n'est pas en forme ANF : le sous-terme gauche de la première liaison locale $\texttt{let}\ x_0 = g\ [x]\ \texttt{in}\ f\ [x_0]$ n'est pas un calcul. De même, lors de la transformation récursive de sous-termes en sous-termes nous pouvons rencontrer d'autres formes syntaxiques incorrectes vis-à-vis de la grammaire de la forme ANF :

- $\texttt{let}\ x = (\texttt{let}\ y = t\ \texttt{in}\ t')\ \texttt{in}\ \ldots$,
- $\texttt{match}\ (\texttt{let}\ x = t\ \texttt{in}\ t')\ \texttt{with}\ \ldots$,
- $\texttt{let}\ x = (\texttt{match}\ t\ \texttt{with}\ \ldots)\ \texttt{in}\ \ldots$ et
- $\texttt{match}\ (\texttt{match}\ t\ \texttt{with}\ \ldots)\ \texttt{with}\ \ldots$.

Après la transformation sous-termes par sous-termes, il faut appliquer une phase de renormalisation afin d'obtenir des termes en forme ANF. Cette renormalisation est basée sur les quatre règles suivantes :

`let-let :`

$$\begin{array}{rl} & \texttt{let}\ x_0 = (\texttt{let}\ x_1 = t_1\ \texttt{in}\ t_0)\ \texttt{in}\ t \\ \Rightarrow & \texttt{let}\ x_1 = t_1\ \texttt{in let}\ x_0 = t_0\ \texttt{in}\ t \end{array}$$

`match-let :`

$$\begin{array}{rl} & \texttt{match}\ (\texttt{let}\ x = t\ \texttt{in}\ t')\ \texttt{with}\ \ldots \\ \Rightarrow & \texttt{let}\ x = t\ \texttt{in match}\ t'\ \texttt{with}\ \ldots \end{array}$$

`let-match` :

$$\begin{array}{rl} & \texttt{let}\ x = (\texttt{match}\ t\ \texttt{with}\ \ldots \vec{|x_i} \to t_i | \ldots)\ \texttt{in}\ t' \\ \Rightarrow & \texttt{match}\ t\ \texttt{with}\ \ldots \vec{|x_i} \to\ \texttt{let}\ x = t_i\ \texttt{in}\ t' | \ldots \end{array}$$

`match-match`

$$\begin{array}{rl} & \texttt{match}\ (\texttt{match}\ t\ \texttt{with}\ \ldots \vec{|x_i} \to t_i | \ldots)\ \texttt{with}\ \ldots \vec{|y_j} \to t_j | \ldots \\ \Rightarrow & \texttt{match}\ t\ \texttt{with}\ \ldots \vec{|x_i} \to (\texttt{match}\ t_i\ \texttt{with}\ \ldots \vec{|y_j} \to t_j | \ldots) | \ldots \end{array}$$

Cette renormalisation est coûteuse ; en particulier les deux dernières règles dupliquent du code.

Un bon candidat : la forme monadique intermédiaire

Pour pallier ces difficultés, nous allons utiliser une *forme monadique intermédiaire* en lieu et place de la forme ANF. L'intuition est de considérer la grammaire apparaissant lors de la mise en forme ANF avant la renormalisation par les quatre règles (`let-let`, `match-let`, `let-match` et `match-match`). La distinction syntaxique entre les termes et les calculs n'est plus nécessaire. Cette forme ne distingue que les atomes des autres termes :

Atomes : $a ::= x \mid \texttt{field}\ n\ a$

Termes : $\begin{array}{rl} t & = a \\ & \mid a\ [a_0; \ldots; a_n] \\ & \mid f\ (a_1; \ldots; a_n) \\ & \mid C\ (a_1; \ldots; a_n) \\ & \mid \texttt{let}\ x = t\ \texttt{in}\ t \\ & \mid \texttt{match}\ a\ \texttt{with}\ \pi_0; \ldots; \pi_n \end{array}$

Motifs : $\pi ::= x_1; \ldots; x_n \to t$

En particulier, le sous-terme gauche d'un `let` peut être un terme, l'imbrication à gauche de liaisons locales n'est plus un obstacle. Les duplications de code dues aux règles `let-match` et `match-match` ont disparu. En effet, la première est une forme syntaxique licite tandis que la deuxième se récrit plus simplement : $\texttt{match}\ (\texttt{match}\ t\ \texttt{with}\ \vec{\pi})\ \texttt{with}\ \vec{\pi'}$ devient : $\texttt{let}\ x = (\texttt{match}\ t\ \texttt{with}\ \vec{\pi})\ \texttt{in}\ \texttt{match}\ x\ \texttt{with}\ \vec{\pi'}$.

L'exemple suivant illustre la différence entre un terme en forme A-normale et en forme intermédiaire monadique. Considérons le terme en style direct :

$$C(f\ [x] :: g\ [y]).$$

Alors sa forme monadique intermédiaire est :

$$\texttt{let}\ x_0 = (\texttt{let}\ x_1 = f\ [x]\ \texttt{in}\ \texttt{let}\ x_2 = g\ [y]\ \texttt{in}\ x_1 :: x_2)\ \texttt{in}\ C(x_0).$$

Tandis que la forme A-normale est :

$$\texttt{let}\ x_1 = f\ [x]\ \texttt{in let}\ x_2 = g\ [y]\ \texttt{in let}\ x_0 = x_1 :: x_2\ \texttt{in}\ C(x_0).$$

On peut remarquer que sur une forme A-normale, pour connaître la suite d'un calcul, il suffit de regarder le sous-terme droit de la liaison `let` correspondante. Ainsi, la continuation du calcul de $g\ [y]$ est $\texttt{let}\ x_0 = x_1 :: x_2\ \texttt{in}\ C(x_0)$. En revanche, la continuation d'un calcul sur la forme intermédiaire monadique est plus difficile à retrouver. En effet, la présence d'imbrication gauche de liaisons locales ne situe plus la continuation d'un calcul dans le membre droit de la liaison uniquement. Ainsi, en forme intermédiaire monadique la continuation du calcul de $g\ [y]$ ne se situe pas uniquement dans le membre droit du `let` liant le calcul, il faut ensuite revenir au sous-terme droit du `let` englobant pour accéder à la suite de la continuation.

6.2.2 Calcul des racines sur la forme intermédiaire monadique

Faisons le point. L'interaction avec un gestionnaire de mémoire se fait en deux étapes :
- le calcul des racines,
- l'enregistrement explicite des racines dans une structure de données dédiée, cet enregistrement est le sujet de la section 6.4.5.

Nous précisons ici comment déterminer et calculer les racines sur la forme intermédiaire monadique.

Les déclencheurs potentiels de GC

Les termes déclencheurs potentiels de GC directs sont les constructions de données : fermetures et constructeurs appliqués, et appels de fonction. Par extension, tout terme dont un sous-terme est un déclencheur potentiel de GC peut déclencher un GC. Dans la forme intermédiaire monadique, les termes susceptibles de déclencher un GC sont facilement identifiables.

En tout premier lieu, les structures de données, qui seront toutes représentées par des pointeurs sur des blocs du tas : les fermetures $f\ (a_1; \ldots; a_n)$ et les constructeurs appliqués $C(a_1; \ldots; a_n)$.

Ensuite, les applications de fonction $a\ [a_0; \ldots; a_n]$. En effet, lors de l'évaluation du corps de la fonction appelée, il peut y avoir déclenchement d'un GC.

Enfin, tout terme dont l'un des sous-termes est susceptible de déclencher un GC.

Soit `triggers` la fonction qui prend un argument un terme t et répond si oui (*true*) ou non (*false*) t est un déclencheur potentiel de GC.

$$\begin{aligned}
\texttt{triggers}\ a &= \mathit{false} \\
\texttt{triggers}\ (a\ [a_0; \ldots; a_n]) &= \mathit{true} \\
\texttt{triggers}\ (f\ (a_1; \ldots; a_n)) &= \mathit{true} \\
\texttt{triggers}\ (C\ (a_1; \ldots; a_n)) &= \mathit{true} \\
\texttt{triggers}\ (\texttt{let}\ x\ =\ t_1\ \texttt{in}\ t_2) &= \texttt{triggers}\ t_1 || \texttt{triggers}\ t_2 \\
\texttt{triggers}\ (\texttt{match}\ a\ \texttt{with}\ \pi_0; \ldots; \pi_n) &= \texttt{triggers}\ a || \texttt{triggers}\ \pi_0 || \ldots || \texttt{triggers}\ \pi_n
\end{aligned}$$

$$\texttt{triggers}\ |x_1;\ldots;x_n \to t \quad = \quad \texttt{triggers}\ t$$

Calcul de racines

Il y a différentes manières d'approximer une variable comme étant une racine. La première approximation est grossière. Il s'agit de considérer toute variable comme étant une racine durant sa portée lexicale. Cependant, une variable ne sera pas forcément racine d'un déclenchement potentiel de GC. C'est la cas si dans sa portée lexicale ne figure aucun déclencheur potentiel de GC. Par exemple, dans le terme

$$\texttt{let}\ x = C(a_1;\ldots;a_n)\ \texttt{in}\ x,$$

la variable x ne sera pas racine pour un appel de GC : dans le sous-terme droit x il n'y a pas de déclencheur potentiel de GC.

Nous raffinons cette approximation en considérant comme racine uniquement les variables dont la portée lexicale contient un potentiel déclencheur de GC selon la définition donnée plus haut (`triggers`). Plus précisément, lors de la déclaration d'une variable :

$$\texttt{let}\ x = t_1\ \texttt{in}\ t_2,$$

x sera enregistrée dans la structure de donnée dédiée comme étant une racine si elle appartient à l'ensemble des racines de t_2, que l'on note $R(t_2)$.

Les racines les plus évidentes à repérer dans un terme sont, bien entendu, les arguments des futures structures de données : $C(a_1;\ldots;a_n)$ et $f\ (a_1;\ldots;a_n)$. En effet, dans les deux cas, on va allouer un bloc dans le tas et les variables libres de $a_1;\ldots;a_n$ doivent survivre à un GC potentiellement déclenché par ces allocations. Les arguments d'une application ne sont pas des racines. Plus précisément, c'est à la fonction appelée de déterminer parmi ses paramètres lesquelles sont des racines dans son corps de fonction.

Dans les autres termes, nous appliquons récursivement ce calcul de racines. Au niveau de la liaison locale `let` notre calcul est plus fin. En effet, considérons le terme $\texttt{let}\ x = t_1\ \texttt{in}\ t_2$. Si t_1 n'est pas susceptible de déclencher un GC, il est inutile d'en préserver les racines. Les racines de $\texttt{let}\ x = t_1\ \texttt{in}\ t_2$ sont alors celle de t_2. Si t_1 est susceptible de déclencher un GC, alors les racines de t_1 font partie de celles de $\texttt{let}\ x = t_1\ \texttt{in}\ t_2$. De plus, toutes les variables libres dans t_2 doivent survivre au potentiel déclenchement de GC de t_1.

Notre calcul de racines sur une forme intermédiaire monadique se présente comme suit :

$$\begin{aligned}
R(a) &= \varepsilon \\
R(a\ [a_0;\ldots;a_n]) &= \varepsilon \\
R(f\ (a_1;\ldots;a_n)) &= FV(a_1) \cup \ldots \cup FV(a_n) \\
R(C\ (a_1;\ldots;a_n)) &= FV(a_1) \cup \ldots \cup FV(a_n) \\
R(\texttt{let}\ x\ =\ t_1\ \texttt{in}\ t_2) &= \begin{cases} R(t_1) \cup FV(t_2) \setminus \{x\} & \text{si } \texttt{triggers}\ t_1 = \texttt{true}\,; \\ R(t_2) \setminus \{x\} & \text{sinon} \end{cases} \\
R(\texttt{match}\ a\ \texttt{with}\ \pi_0;\ldots;\pi_n) &= R(\pi_0) \cup \ldots \cup R(\pi_n) \\
R(x_1;\ldots;x_n \to t) &= R(t) \setminus \{x_1,\ldots,x_n\}
\end{aligned}$$

Dans l'exemple suivant :

$$\texttt{let } x = t_1 \texttt{ in } C\ (x, y),$$

x est enregistrée comme une racine. En effet, $C\ (x, y)$ est un déclencheur potentiel de GC.

Tandis que, dans :

$$\texttt{let } x = t_1 \texttt{ in } a\ [x, y],$$

x n'est pas enregistrée comme une racine. En effet, la fonction vers laquelle s'évalue a détermine parmi ses paramètres s'il y a des racines.

6.3 Mise en forme intermédiaire monadique

Cette section donne une description de la première phase de l'interaction avec un GC : la mise en forme intermédiaire monadique. Dans notre chaîne de compilation, la forme intermédiaire monadique est implémentée sous la forme d'un langage intermédiaire appelé Mon. La mise en forme intermédiaire monadique d'un programme Fml est une traduction vers un programme Mon.

Nous commençons par décrire le langage Mon au travers de sa syntaxe et de sa sémantique. Puis nous décrivons la transformation avant d'étudier sa preuve de préservation sémantique.

6.3.1 La forme intermédiaire monadique

Le langage Mon est un langage fonctionnel, dont les fermetures ont été explicitées, portant dans la syntaxe les caractéristiques de la forme intermédiaire monadique.

Syntaxe

La syntaxe du langage Mon est fidèle à la grammaire de la forme intermédiaire monadique :

Atomes :
$$\begin{aligned} a ::= {} & x \\ \mid {} & \texttt{field } n\ a \end{aligned}$$

Termes :
$$\begin{aligned} t ::= {} & a \\ \mid {} & a\ [a_0; \ldots; a_n] \\ \mid {} & f\ (a_1; \ldots; a_n) \\ \mid {} & \texttt{let } x = t_1 \texttt{ in } t_2 \\ \mid {} & C\ (a_1; \ldots; a_n) \\ \mid {} & \texttt{match } a \texttt{ with } \pi_0; \ldots; \pi_n \end{aligned}$$

Motifs :
$$\pi ::= x_1; \ldots; x_n \rightarrow t$$

Fonctions :
$$def ::= f\ :\ \{\texttt{params}\ :\ x_0; \ldots; x_n\ ;\ \texttt{body}\ :\ t\}$$

Programme : $prog ::= \{\texttt{defs} : \vec{def}\ ;\ \texttt{main} : t\}$

Un programme Mon s'organise comme un programme Fml. Les termes Mon diffèrent quant à eux des termes Fml. Ils sont organisés en atomes (variables et accès aux éléments de fermetures) et en termes. Mise à part la liaison locale, $\texttt{let}\ x = t_1\ \texttt{in}\ t_2$ et les corps de choix de filtrage $x_1; \ldots; x_n \to t$, les sous termes de Fml ont laissé place à des atomes.

Sémantique naturelle à environnement

La sémantique de Mon est définie comme celle de Fml, à ceci près qu'elle distingue l'évaluation des atomes de celles des termes. Nous avons donc deux jugements :

Évaluation d'un atome $S, e \vdash a \Rightarrow v$, exprime que l'atome a s'évalue en v dans l'environnement d'évaluation formé par la liste de déclarations de fonctions S et l'environnement e.

Évaluation d'un terme $S, e \vdash t \Rightarrow v$, exprime que le terme t s'évalue en v dans l'environnement d'évaluation (S, e).

$$\frac{e(x) = v}{S, e \vdash x \Rightarrow v} \qquad \frac{S, e \vdash a \Rightarrow (f, v_1; \ldots; v_n) \quad 1 \leq i \leq n}{S, e \vdash \texttt{field}\ i\ a \Rightarrow v_i}$$

$$\frac{S, e \vdash a_i \Rightarrow v_i \quad 1 \leq i \leq n}{S, e \vdash f\ (a_1; \ldots; a_n) \Rightarrow (f, v_1; \ldots; v_n)} \qquad \frac{S, e \vdash a_i \Rightarrow v_i \quad 1 \leq i \leq n}{S, e \vdash C(a_1; \ldots; a_n) \Rightarrow (C, v_1; \ldots; v_n)}$$

$$\frac{S, e \vdash t_1 \Rightarrow v_1 \quad S, e[x \leftarrow v_1] \vdash t_2 \Rightarrow v}{S, e \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 \Rightarrow v}$$

$$\frac{\begin{array}{c} S, e \vdash a \Rightarrow (f,\ w_1; \ldots; w_k) \quad (f, d) \in S \quad d.\texttt{params} = x_0; \ldots; x_n \\ S, e \vdash a_i \Rightarrow v_i \quad 0 \leq i \leq n \quad S, \varepsilon[x_0 \leftarrow v_0] \ldots [x_n \leftarrow v_n] \vdash (d.\texttt{body}) \Rightarrow v \end{array}}{S, e \vdash a\ [a_1; \ldots; a_n] \Rightarrow v}$$

$$\frac{S, e \vdash a \Rightarrow (C, v_1; \ldots; v_k) \quad \pi_C = x_1; \ldots; x_k \to t \quad S, e[x_1 \leftarrow v_1] \ldots [x_k \leftarrow v_k] \vdash t \Rightarrow v}{S, e \vdash \texttt{match}\ a\ \texttt{in}\ \pi_0; \ldots; \pi_n \Rightarrow v}$$

Les règles d'évaluation définissant la sémantique de Mon sont très proches de celles de Fml. Les sous-termes ayant laissé place à des atomes, les jugements correspondants sont des jugements d'évaluation d'atomes.

Tout comme la sémantique de Fml, la sémantique de Mon comporte des vérifications concernant les noms de variables. On ne vérifie plus que les variables sont différentes de `Root`, on vérifie qu'elles ne font pas partie de *lid*. *lid* désigne une liste de noms de variables qui seront utilisées de manière spéciale lors de la génération de code Cminor.

Enfin, l'évaluation d'un programme se fait comme celle d'un programme Fml.

6.3.2 Algorithme

Mettre en forme intermédiaire monadique consiste, en gros, à décomposer les calculs de Fml en les rendant élémentaires tout en nommant leurs résultats. De manière pragmatique, il s'agit d'attribuer un nom à chaque sous-terme, qui ne soit pas déjà un atome, via une liaison locale et de remplacer le sous-terme par cette variable. Par exemple,

$$\texttt{cons}(f\ [x], g\ [y]) \text{ devient } \texttt{let}\ x_0 = f\ [x]\ \texttt{in let}\ x_1 = g[y]\ \texttt{in cons}(x_0, x_1).$$

Il y a deux possibilités de transformation d'un sous terme t apparaissant dans un terme T.

t est un atome : si t est une variable ou un accès à un élément de fermeture, t est déjà un atome et donc il n'est pas nécessaire de le transformer.

t est un terme complexe : sinon il nous faut remplacer t par un atome dans T, plus précisément par une variable, et élémentariser le calcul qu'il représente. Pour cela, nous lions t', qui se trouve être la forme intermédiaire monadique de t, à une nouvelle variable x de manière locale à T. Dans T, t est alors remplacé par x :

$$\texttt{let}\ x\ =\ t'\ \texttt{in}\ T[t \leftarrow x].$$

Élémentarisation des calculs : Nous appelons ce procédé *atomisation*. Il consiste à considérer un terme t du langage Mon, un nom de variable x et un terme Mon à trou K. Ce terme à trou K attend un atome pour devenir un terme Mon. De tels termes sont appelés *contextes d'atomisation* et sont représentés dans notre formalisation `Coq` par des fonctions $\texttt{fun}\ a \Rightarrow T$ de type $\texttt{atome}\ \rightarrow \texttt{terme}$.

Considérons le terme Fml : $S(f\ [x])$, et le sous-terme $f\ [x]$. A ce point du processus d'atomisation, le sous-terme Mon considéré est la forme intermédiaire monadique de $f(x)$, c'est-à-dire $f(x)$. Supposons x_0 le nom de variable choisi comme étant frais (nous reviendrons plus tard sur la génération de noms frais). Le contexte d'atomisation considéré est

$$\texttt{fun}\ a \Rightarrow S(a).$$

Dans le cas présent, $f\ [x]$ n'est pas un atome. L'atomisation crée donc une liaison locale de $f\ [x]$ à la variable x_0 et applique le contexte d'atomisation à cette variable :
$(\texttt{fun}\ a \Rightarrow S(a))\ x_0 = S(x_0)$. On obtient la forme intermédiaire monadique de $S(f\ [x])$:

$$\texttt{let}\ x_0 = f\ [x]\ \texttt{in}\ S(x_0).$$

D'un autre côté, si l'on considère $S(y)$, alors on ne crée pas de liaison locale mais on passe y en argument au contexte d'atomisation, $(\texttt{fun}\ a \Rightarrow S(a))\ y = S(y)$.

Dans notre développement, l'atomisation est définie via les fonctions `atomize` et `atomize_list` comme suit :

$$\begin{array}{rcl}
\texttt{atomize}\ t\ x\ K & = & \begin{cases} K\ t & \text{si } t \text{ est un atome ;} \\ \texttt{let}\ x\ =\ t\ \texttt{in}\ K\ x & \text{sinon} \end{cases} \\
\texttt{atomize_list}\ [t_1; \ldots; t_n]\ x\ K & = & \texttt{atomize}\ t_0\ x\ (\texttt{fun}\ a_0\ \Rightarrow \\
 & & \texttt{atomize}\ t_1\ (\texttt{nextname}\ x)(\texttt{fun}\ a_1 \Rightarrow \ldots \\
 & & K\ [a_0, a_1, \ldots, a_n]
\end{array}$$

La grande similitude avec une transformation CPS n'est pas anodine. Tout comme la forme A-normale, la forme intermédiaire monadique peut être vue comme une forme faible de CPS. L'argument K de `atomize` peut être vu comme le corps de la continuation et x le paramètre de cette continuation. Tout comme l'application intelligente $@_\beta$ lors de la transformation CPS (voir la section 4.1.3), `atomize` distingue son action selon que le terme considéré est un atome ou pas.

Les contextes d'atomisation Plus généralement, les contextes d'atomisation peuvent être vus comme la construction syntaxique correspondante où chaque atome a été abstrait. Ces abstractions sont des abstractions `Coq`. Ainsi, nous n'avons pas à définir de substitution dans le langage Mon. Nous définissons donc pour chaque construction du langage Mon un contexte d'atomisation :

$$\begin{aligned}
\texttt{Kfield}\ n &= \texttt{fun}\ a \Rightarrow\ \texttt{field}\ n\ a \\
\texttt{Kclos}\ f &= \texttt{fun}\ [a_1;\ldots;a_n] \Rightarrow f\ (a_1;\ldots;a_n) \\
\texttt{Kapp}\ a &= \texttt{fun}\ [a_0;\ldots;a_n] \Rightarrow a\ [a_0;\ldots;a_n] \\
\texttt{Kcstr}\ C &= \texttt{fun}\ [a_1;\ldots;a_n] \Rightarrow C\ (a_1;\ldots;a_n)
\end{aligned}$$

Génération de noms uniques Lors d'une atomisation, un nom de variable est pris en paramètre. Nous voulons une unicité des noms de variables dans notre programme Mon. Dans la transformation précédente, nous passions d'indices de de Bruijn à des variables nommées. Nous utilisions l'encodage des identifiants dans le type `Positive` de `Coq` pour garantir une génération unique des noms de variables et l'état de la monade pour propager les effets de bord. Dans la présente transformation, nous ajoutons de nouvelles variables. Le type `positive` est défini inductivement par trois constructeurs : `xH` (pour 1), `xI` p (représentant $2p+1$) et `xO` p (représentant $2p$). Utilisant la construction des `positive` de `Coq`, nous distinguons les variables présentes dans le langage source des variables ajoutées par atomisation. La variable Fml x sera traduite vers une représentation paire (`xO` x) que l'on note $\overline{x}$. Les nouvelles variables seront quant à elles des représentations impaires (`xI` x), qui seront notées $\underline{x}$. Leur génération est assurée par un entier pris en paramètre x, qui génère comme prochaine variable $\underline{x}$ et est alors incrémenté afin de créer la prochaine fois un nouveau nom frais.

Transformation d'un terme Nous notons $[\![t]\!]_x$, la mise en forme intermédiaire monadique du terme Fml t avec comme générateur de noms frais x. Cette transformation retourne un couple composé du terme t' forme intermédiaire monadique du terme t et un identifiant x' générateur de nom frais pour la transformation des termes suivants. La transformation est définie comme suit :

$$\begin{aligned}
[\![y]\!]_x &= (\overline{y}, x) \\
[\![\texttt{field}\ n\ t]\!]_x &= \texttt{atomize}\ t'\ \underline{x'}\ (\texttt{Kfield}\ n) \\
&\quad \text{avec} : (t', x') = [\![t]\!]_{x+1};
\end{aligned}$$

$$
\begin{array}{rcl}
[\![f\ (t_1;\ldots;t_n)]\!]_x & = & \texttt{atomize\_list}\ \ t'_1;\ldots;t'_n\underline{x'}\ (\texttt{Kclos}\ f) \\
 & & \text{avec : } (t'_1;\ldots;t'_n,x') = [\![t_1;\ldots;t_n]\!]_{x+1} \\
[\![t\ [t_0;\ldots;t_n]]\!]_x & = & (\texttt{atomize}\ \underline{x}\ t'\ (\texttt{fun}\ a \Rightarrow\ \texttt{atomize\_list} \\
 & & t'_0;\ldots;t'_n\ \underline{x'}\ (\texttt{Kapp}\ a)), x'') \\
 & & \text{avec : } (t',x') = [\![t]\!]_{x+1} \\
 & & \text{et } (t'_0;\ldots;t'_n,x'') = [\![t_0;\ldots;t_n]\!]_{x'} \\
[\![C\ (t_1;\ldots;t_n)]\!]_x & = & \texttt{atomize\_list}\ \ t'_1;\ldots;t'_n\ \underline{x'}\ (\texttt{Kcstr}\ C) \\
 & & \text{avec : } (t'_1;\ldots;t'_n,x') = [\![t_1;\ldots;t_n]\!]_{x+1} \\
[\![\texttt{let}\ y\ =\ t_1\ \texttt{in}\ t_2]\!]_x & = & (\texttt{let}\ \overline{y}\ =\ t'_1\ \texttt{in}\ t'_2, x'') \\
 & & \text{avec : } (t'_1,x') = [\![t_1]\!]_{x+1} \\
 & & \text{et } (t_2,x'') = [\![t_2]\!]_{x'} \\
[\![\texttt{match}\ t\ \texttt{with}\ \pi_0;\ldots;\pi_n]\!]_x & = & (\texttt{atomize}\ \ t' \\
 & & (\texttt{fun}\ a \texttt{=>}\ \texttt{match}\ a\ \texttt{with}\ \pi'_0;\ldots;\pi'_n)\ \underline{x}, x'') \\
 & & \text{avec :} (t',x') = [\![t]\!]_{x+1} \\
 & & \text{et } (\pi'_0;\ldots;\pi'_n,x'') = [\![\pi_0;\ldots;\pi_n]\!]_{x'}) \\
 & & \\
[\![x_0;\ldots;x_n \rightarrow t]\!]_x & = & (\overline{x_1},\ldots,\overline{x_n} \rightarrow t',x') \\
 & & \text{avec :} (t',x') = [\![t]\!]_{x+1}
\end{array}
$$

Une variable x est traduite par la variable $\underline{x}$. La transformation d'une liaison locale $\texttt{let}\ y\ =\ t_1\ \texttt{in}\ t_2$, avec comme générateur de noms x, produit la liaison locale Mon dont les sous termes sont les transformations de t_1 et t_2, et mettant en jeu la variable $\overline{y}$.
De même, lors de la transformation d'un motif de filtrage, les paramètres formels sont transformés convenablement et le corps est transformé avec comme générateur de nom x.
Concernant les autres structures syntaxiques, on applique les atomisations aux contextes correspondant des atomisations des sous-termes en donnant comme nom généré $\underline{x}$.

Transformation d'un programme La mise en forme intermédiaire monadique d'une fonction Fml commence par harmoniser les noms de ses paramètres formels par rapport à la nomenclature de noms de variables. Ainsi, si la liste des paramètres formels est $(x_1;\ldots;x_n)$, alors la liste de paramètres formels de la fonction Mon est $(\overline{x_1},\ldots,\overline{x_n})$. Le corps de la fonction est transformé avec comme générateur de noms unique 2.

$$[\![(f,d)]\!] \quad = \quad (f, \{\texttt{params}\ = \overline{d.\texttt{params}};\ \texttt{body}\ = [\![d.\texttt{body}]\!]_2\}$$

Enfin, la mise en forme intermédiaire monadique d'un programme Fml p produit un programme Mon p' tel que :
- la liste de déclarations de p' est la liste des traductions des déclarations de celle de p.
- le terme `main` de p' est la traduction de $p.\texttt{main}$.

6.3.3 Préservation sémantique

Nous avons montré la préservation de la sémantique des programmes Fml lors de leur de la mise en forme intermédiaire monadique :

Théorème 6.3.1 (Préservation sémantique de la transformation monadique) *Si le programme p Fml s'évalue en v, alors il existe une valeur v' telle que $[\![p]\!]$ s'évalue en v' et que v corresponde à v'.*

La preuve de ce théorème nécessite une simulation sur l'évaluation d'un terme Fml. Avant d'énoncer cette simulation, nous en décrivons les invariants.

Invariants

La simulation d'une évaluation d'un terme Fml en l'évaluation du terme Mon en forme intermédiaire monadique suppose une comparaison observationnelle entre les deux évaluations. Un jugement Fml est de la forme :

$$S, e \vdash t \Rightarrow v,$$

En parallèle, l'évaluation d'un terme Mon fait intervenir deux sortes de jugements, un pour les atomes et un pour les termes :

$$S', e' \vdash a \Rightarrow v \text{ et } S', e' \vdash t' \Rightarrow v'.$$

Plus précisément, une évaluation d'un terme t Fml sera simulé par l'évaluation de sa traduction $[\![t]\!]_x$. De plus, t est un sous-terme du programme p. Il en résulte $S' = [\![p.\texttt{defs}]\!]$.

En nous appuyant sur ces faits, nous voulons dégager une équivalence observationnelle entre les deux évaluations. Cette observation est matérialisée par les invariants suivants.

Relation de correspondance entre valeurs sémantiques Le cœur de la relation de correspondance entre les valeurs sémantiques Fml et Mon réside dans la correspondance entre fermetures, et plus précisément, entre les fonctions invoquées par ces fermetures. En effet, une fermeture Mon correspond à une fermeture Fml si, entre autres, la fonction à laquelle elle réfère est la traduction de celle que réfère la fermeture Fml. Or, le fait que $S' = [\![p.\texttt{defs}]\!]$ nous garantit cette propriété. Il suffit donc que le nom de la fonction apparaissant dans la fermeture Fml soit associé à une fonction dans S. Cette relation est propagée au niveau des valeurs portées par les fermetures et les arguments des constructeurs appliqués.

La relation de correspondance des valeurs sémantiques est donc paramétrée par S :

$$S \vdash v \sim v'$$

et se lit : la valeur Fml v correspond à la valeur Mon v' par rapport à la liste de déclarations de fonctions Fml S. Cette relation se définit par les deux règles suivantes :

$$\frac{(f,d) \in S \qquad S \vdash v_i \sim v'_i \qquad \text{pour tout } i \in [1,n]}{S \vdash (f,(v_1;\ldots;v_n)) \sim (f,v'_1;\ldots;v'_n))} \qquad \frac{S \vdash v_i \sim v'_i \qquad \text{pour tout } i \in [1,n]}{S \vdash C(v_1;\ldots;v_n) \sim C(v'_1;\ldots;v'_n)}$$

Relation de correspondance entre environnements locaux La relation de correspondance entre valeurs est à la base de la correspondance entre environnements locaux. Elle considère en plus, l'ajout de variables par la mise en forme intermédiaire monadique.

Pour rappel, la mise en forme intermédiaire monadique conserve les variables sources en les renommant selon la règle suivante : si y est une variable source, elle devient $\overline{y}$ dans Mon. Ainsi, les variables de e, environnement local de Fml, correspondent aux variables surlignées dans l'environnement simulant Mon. Autrement dit, la valeur correspondant à $e(y)$ se trouve stockée dans $e'(\overline{y})$.

Au cours de la transformation, et plus particulièrement des atomisations, de nouvelles variables sont générées. Cette génération est assurée par un générateur de noms uniques pris en paramètre de la transformation, ici, x. L'environnement simulant doit être cohérent vis-à-vis de ce générateur. En effet, le terme ne contient pas de variable ajoutée plus grande que la dernière générée. Si $x < y$ alors $e'(\underline{y}) = \bot$.

Les environnements locaux se correspondent, vis-à-vis d'un générateur de noms uniques selon ces deux propriétés.

Définition 6.3.2 (Correspondance entre environnements locaux)
$S \vdash e \sim_x e'$ si pour tout y, $S \vdash e(y) \sim e'(\overline{y})$ et pour tout x', $x < x'$, $e'(\underline{x'}) = \bot$.

Simulation

Nous sommes maintenant en mesure d'énoncer le lemme de simulation :

Lemme 6.3.3 (Simulation)
Supposons $S \vdash e \sim_x e'$; si

$$S, e \vdash t \Rightarrow v$$

alors il existe v' telle que :

$$[\![S]\!], e' \vdash [\![t]\!]_x \Rightarrow v' \text{ et } S \vdash v \sim v'.$$

Ce lemme se montre par induction sur l'évaluation $S, e \vdash t \Rightarrow v$.

Durant la preuve, nous obtiendrons les hypothèses d'induction sur les sous termes et les listes de sous termes de t selon la structure syntaxique de t. Par exemple, dans le cas d'une fermeture syntaxique Fml $(f, (t_1; \ldots; t_n))$, nous aurons en hypothèse la simulation de l'évaluation de la liste des sous termes qu'elle porte :

H : Supposons $S \vdash e \sim_x e'$, si

$$S, e \vdash t_i \Rightarrow v_i \text{ pour tout } i \in [1; n[$$

alors il existe $v'_1; \ldots; v'_n$ telle que :

$$[\![S]\!], e' \vdash [\![t_i]\!]_x \Rightarrow v'_i \text{ et } S \vdash v_i \sim v'_i \text{ pour tout } i \in [1, n[.$$

Or, lors de la mise en forme intermédiaire monadique

$$\begin{array}{rcl} [\![(f, t_1; \ldots; t_n)]\!]_x & = & \texttt{atomize\_list } t'_1; \ldots; t'_n \underline{x'} \ (\texttt{Kclos } f) \\ & & \text{avec } (t'_1; \ldots; t'_n, x') = [\![t_1; \ldots; t_n]\!]_{x+1} \end{array}$$

L'atomisation listée peut potentiellement transformer la liste de sous termes en une imbrication de liaisons locales de longueur inférieur ou égale à n. Par exemple $(f, (t_0; y; t_1))$ où t_0 et t_1 ne sont pas atomiques. La traduction donne :

$$\texttt{let } \underline{x} = t'_0 \texttt{ in let } \underline{x+1} = t'_1 \texttt{ in } f\ (\underline{x}, y, \underline{x+1}).$$

où $(t'_0, x') = [\![t_0]\!]_{x+1}$ et $(t'_1, x'') = [\![t_1]\!]_{x'}$. Lors de l'évaluation, le terme t'_0 est évalué dans l'environnement courant e', supposons en v_0. Puis, t'_1 est évalué dans $e'[\underline{x} \leftarrow v_0]$, supposons en v_1. Enfin y est évalué dans $e'[\underline{x} \leftarrow v_0][\underline{x+1} \leftarrow v_1]$.

Plus généralement, nous devons passer de l'hypothèse d'induction sur la liste d'évaluations des sous termes dans un même environnement H, à une évaluation potentiellement séquentielle de ces mêmes sous-termes au travers de liaisons locales imbriquées. De plus, l'ordre d'évaluation diffère. En effet, dans le terme d'origine l'évaluation des sous termes se faisant de gauche à droite, y était évalué avant t_1. En forme intermédiaire monadique, la transformation fait apparaître t_1 plus à gauche que y.

Cependant, on peut caractériser les environnements intermédiaires qui peuvent apparaître au travers du processus d'élémentarisation des calculs. Si l'on observe de plus près ces environnements intermédiaires, chaque sous terme sera évalué dans un environnement incluant e'. Nous définissons l'inclusion d'environnement comme suit :

Définition 6.3.4 (Inclusion d'environnements)
$e \subseteq e'$ si pour tout x tel que, $e(x) = v$ alors, $e'(x) = v$.

En effet, si l'on considère l'exemple précédent,

$$e' \subseteq e'[\underline{x} \leftarrow v_0]$$

et

$$e' \subseteq e'[\underline{x} \leftarrow v_0][\underline{x+1} \leftarrow v_1].$$

De plus, nous savons que les informations contenues dans e' suffisent à l'évaluation de ces sous-termes. Il est aisé de montrer que l'évaluation d'un terme ou d'un atome Mon dans un environnement e' est équivalente dans tout environnement incluant e'.

Théorème 6.3.5 (Évaluation et inclusion d'environnement)
Si $S, e \vdash a \Rightarrow v$ et $e \subseteq e'$ alors $S, e' \vdash a \Rightarrow v$.
Si $S, e \vdash t \Rightarrow v$ et $e \subseteq e'$ alors $S, e' \vdash t \Rightarrow v$.

Ces deux théorèmes se prouvent par induction sur les évaluations.

Le passage d'une évaluation de liste de sous-termes à une évaluation d'imbrication de liaisons locales se produit lors de l'évaluation de environnement d'atomisation appliqué. Plus exactement, c'est l'atomisation qui pour chaque sous terme va décider, selon qu'il soit déjà un atome ou pas, s'il y a création d'une liaison locale ou pas. C'est donc l'élémentarisation des calculs qui peut changer l'ordre d'évaluation des sous termes.

Nous savons quels sont les environnements d'atomisation apparaissant lors de la mise en forme intermédiaire monadique : il y en a un pour chaque structure syntaxique. Or, la preuve de simulation se fait par induction sur une évaluation Fml. La sémantique de Fml est

à grand pas, elle est donc dirigée par la syntaxe. De plus, elle compte une règle d'évaluation pour chaque structure syntaxique.

Nous avons donc intérêt à produire une règle d'évaluation hybride pour chaque environnement d'atomisation Mon. L'idée est de cacher dans ces règles le passage de l'évaluation d'une liste de sous-termes évaluées de gauche à droite dans le même environnement :

$$[\![S]\!], e' \vdash [\![t_i]\!]_x \Rightarrow v'_i \text{ pour tout } i \in [0; n[.$$

À l'évaluation d'une imbrication de liaisons locales d'au plus n liaisons suivi de l'évaluation des sous "atomes" qui sont, soit les sous terme d'origine soit des variables fraîchement liées.

Ces règles sont, en fait, des lemmes intermédiaires que nous présentons, ici, sous forme de règles d'inférence. Notons $\texttt{wf}\ e\ x$, si pour tout x' tel que $x \leq x'$, on a $e(\underline{x'}) = \bot$.

$$\frac{\texttt{wf}\ e\ x \quad S, e \vdash t \Rightarrow (f, v_1; \ldots; v_n) \quad 1 \leq i \leq n}{S, e \vdash (\texttt{Kfield}\ i\ t\ x) \Rightarrow v_i}$$

$$\frac{\texttt{wf}\ e\ x \quad (f, d) \in S \quad S, e \vdash t_i \Rightarrow v_i \quad 1 \leq i \leq n}{S, e \vdash (\texttt{Kclos}\ f\ t_1; \ldots; t_n\ x) \Rightarrow (f, v_1; \ldots; v_n)}$$

$$\frac{\texttt{wf}\ e\ x \quad S, e \vdash t_i \Rightarrow v_i \quad 1 \leq i \leq n}{S\ ,\ e \vdash (\texttt{Kcstr}\ i\ t_1; \ldots; t_n\ x) \Rightarrow (i, v_1; \ldots; v_n)}$$

$$\frac{\begin{array}{c}\texttt{wf}\ e\ x \quad \texttt{well\_formed}\ (f, d)\ S \quad d.\texttt{params} = (\overline{x_0}, \ldots, \overline{x_{(n+1)}}) \\ S, e \vdash t \Rightarrow (f\ , (w_1; \ldots; w_k)) \quad S, e \vdash t_i \Rightarrow v_i \quad 0 \leq i \leq n \\ S, \varepsilon[\overline{x_0}\ \leftarrow\ (f, w_1; \ldots; w_k)] \ldots [\overline{x_{(n+1)}} \leftarrow v_n] \vdash d.\texttt{body} \Rightarrow v\end{array}}{S, e \vdash (\texttt{Kapp}\ t\ t_0; \ldots; t_n\ x) \Rightarrow v}$$

La preuve de chacun de ces lemmes fait intervenir les résultats entre inclusion d'environnements et évaluation énoncés plus haut. Lors de la simulation, les lemmes d'évaluation des environnements d'atomisation appliqués permettent de conclure les simulations de chaque construction syntaxique.

6.4 Enregistrement effectif des racines, vers une génération de code Cminor

La forme intermédiaire monadique nous permet un calcul simple des racines pour un GC. Nous enregistrons de manière explicite nos racines dans une structure de données dédiée. Cette structure devient en Cminor une liste simplement chaînée sur les blocs de pile Cminor (lors de l'évaluation, les blocs de pile Cminor forme une pseudo-pile d'appel). Les blocs de pile Cminor sont directement manipulables syntaxiquement. Afin de vérifier formellement l'interaction avec un gestionnaire de mémoire et de se rapprocher du langage Cminor nous définissons le langage intermédiaire Fminor qui permet de manipuler syntaxiquement la structure dédiée à l'enregistrement des racines (l'ensemble des racines). Fminor est un langage intermédiaire purement fonctionnel à mi-chemin entre Mon et Cminor.

En plus d'expliciter syntaxiquement l'ensemble des racines, Fminor modélise le comportement du GC dans sa sémantique. Pour cela, sa sémantique à grand pas par appel par valeur se fait avec deux environnements : le premier pour les variables et le second pour les racines. De plus, on observe les environnements à la sortie de l'évaluation d'un terme. Ainsi, en spécifiant dans les règles de sémantique des termes déclencheurs potentiels de GC (constructions de données et appels de fonction) les environnements de fin d'évaluation par rapport aux environnements de début d'évaluation, nous pouvons spécifier le comportement d'un GC. Après évaluation d'un tel terme, d'une part, les variable sont inutiles à la suite de l'évaluation : l'environnement de variables de fin d'évaluation est donc vide. D'autre part, seules les racines sont utiles à la suite de l'évaluation : l'environnement de racine en fin d'évaluation et le même qu'à son début.

Dans un premier temps, nous donnons les intuitions qui justifient la définition de ce langage intermédiaire. Puis, nous le définissons formellement au travers de sa syntaxe et de sa sémantique. Cette sémantique nous permet de spécifier notre calcul de racines et l'enregistrement dans la structure de données dédiée de ces racines. Enfin, nous présentons la transformation de programme Mon et programme Fminor avant d'en esquisser la preuve de préservation sémantique. La sémantique de Fminor spécifiant l'interaction avec un gestionnaire de mémoire, la preuve de préservation sémantique de la compilation vers Fminor constitue la vérification formelle de cette interaction.

6.4.1 Interaction avec un GC via la structure de données dédiée à l'enregistrement des racines

Comme nous l'avons dit plus haut, notre enregistrement de racines se fait au niveau des liaisons de variables. Une variable x qui sera racine à un point donné dans sa portée lexicale sera stockée en pile durant toute sa portée lexicale. Ce mécanisme peut se matérialiser en dédoublant les lieurs afin de différencier les variables qui seront racines des autres. Ainsi, une liaison locale $\texttt{let}\ x = t_1\ \texttt{in}\ t_2$ deviendrait une liaison locale dans l'ensemble des racines si x est une racine de t_2. Soit :

$$\texttt{let root}\ x\ =\ [\![t_1]\!]\ \texttt{in}\ [\![t_2]\!] \text{ si } x \in R(t_2).$$

Mais, si x n'est pas racine de t_2 alors la liaison reste une liaison locale de variable :

$$\texttt{let var}\ x\ =\ [\![t_1]\!]\ \texttt{in}\ [\![t_2]\!] \text{ si } x \notin R(t_2).$$

De même, sur une clause de filtrage $x_1; \ldots; x_n \to t$, un paramètre formel de motif x_i deviendra un élément de la structure dédiée à l'enregistrement des racines, ou un paramètre formel nommé, selon qu'il est une racine de t ou non.

En Cminor, le bloc de pile Cminor courant est manipulable syntaxiquement. Un bloc de pile est alloué pour chaque appel de fonction. Chaque fonction indique la taille du bloc de pile Cminor nécessaire à l'exécution de son corps. Fminor explicitant l'ensemble des racines, nous pouvons garder le cardinal maximal de cet ensemble en vue de calculer la taille du bloc de pile Cminor lors de la génération de code Cminor. De plus, le nombre de racines avant chaque potentiel déclenchement doit être relevé. La génération de code Cminor aura donc besoin de connaître le nombre de racines courantes pour chaque potentiel déclenchement de GC, ainsi que le cardinal maximal de l'ensemble des racines pour le corps de chaque fonction. Pour cela, il suffit de prendre en paramètre un compteur *cpt* qui est incrémenté à

chaque fois que l'on lie une racine. La valeur de ce compteur est aussi le futur emplacement qui sera attribué à la prochaine racine. Ainsi, si la liaison locale `let` $x = t_1$ `in` t_2 est traduite avec comme nombre de racines dans l'ensemble de racine cpt et que x est racine de t_2, alors x devient l'élément numéro cpt de cet ensemble. Soit :

$$\texttt{let root } cpt \ = \ [\![t_1]\!]' \texttt{ in } [\![t_2]\!]'.$$

Nous notons ici la transformation des termes Mon en terme Fminor par $[\![]\!]'$. En fait, cette transformation nécessitera un environnement de traduction qui associe à une variable Mon une information qui indique que soit cette variable locale est encore une variable locale, soit qu'elle est devenue la $i$ème racine. De plus, afin d'attribuer le bon numéro à la prochaine racine, l'environnement est muni d'un entier (le compteur). Nous reviendrons plus en détail sur l'environnement de traduction dans la section 6.5.1. Nous faisons de même pour chaque liaisons de paramètres formels dans une clause de filtrage. Ce compteur donne la taille de le nombre de racines enregistrées actuellement.

Enfin, à chaque allocation dans le tas, le code Cminor doit transmettre la taille courante de la pile, c'est-à-dire le nombre de racines qu'elle contient. Ces allocations se produisent au niveau des structures de données, et donc chaque structure de données va porter syntaxiquement cette information nécessaire.

6.4.2 Test de compatibilité avec le modèle mémoire

Enfin, ayant sous la main le nombre maximal de racines pour chaque corps de fonction, nous pouvons calculer la taille du futur bloc de pile Cminor nécessaire à la future fonction Cminor. Nous pouvons donc vérifier si cette taille est correcte vis-à-vis du modèle mémoire de Cminor.

Le modèle mémoire de Cminor manipule des blocs dont la taille ne peut excéder le plus grand entier signé que l'on note ici `max_bloc`. Une pile sera de taille correcte si son nombre d'éléments mots mémoire est inférieur à `max_bloc`. Autrement dit, si $(cpt+2)\times 4 <$ `max_bloc`. Nous définissons :

$$|cpt|_p = (cpt + 2) \times 4 < \texttt{max\_bloc}.$$

Ainsi, nous pouvons rejeter des programmes susceptibles de produire des blocs de pile Cminor de taille incorrecte, à chaque ajout d'une racine de la structure dédiée. Si dans la liaison locale `let` $x = t_1$ `in` t_2, x est racine de t_2, et que le nombre de racines dans l'ensemble des racines est cpt, le terme sera traduit comme la liaison de la cptème racine uniquement si un bloc de pile Cminor de taille $(cpt + 2) \times 4$ est inférieur à la taille de bloc maximal. Soit :

$$\texttt{let root } cpt \ = \ [\![t_1]\!]' \texttt{ in } [\![t_2]\!]' \text{ si } x \in R(t_2) \text{ et } |cpt|_p.$$

Encore une fois nous reviendrons plus amplement sur la transformation à la section 6.5.1. Cependant, la possibilité de rejeter des programmes lors de cette transformation en fait une transformation partielle, qui dans notre développement s'écrit dans la monade d'erreur (voir la section 2.4.4). Le même test est effectué à chaque liaison de paramètre formel de clause de filtrage.

Au passage, nous pouvons effectuer le même genre de tests sur la taille des futurs blocs de tas. Il s'agit des blocs qui représenteront les structures de données. Le bloc qui représentera la

fermeture $(f, a_1; \ldots; a_n)$ portera un pointeur vers chaque représentant des a_i, et un pointeur vers le code de la fonction associé à f. C'est-à-dire, qu'il sera d'une taille de $n+1$ mots mémoire. De même pour le constructeur appliqué $C(a_1; \ldots; a_n)$. Nous définissons :

$$|n|_t = (n+1) \times 4 < \texttt{max\_bloc}.$$

6.4.3 Des fonctions globales plus proches de celles de Cminor

Les fonctions globales de Cminor contiennent des informations utiles à leur exécution telles que la liste des variables locales apparaissant dans leur corps, afin d'initialiser l'environnement local d'évaluation, et la taille du bloc de pile Cminor nécessaire à leur exécution.

Le plus grand emplacement associé à une racine dans l'ensemble des racines est une sur-approximation de la taille du bloc de pile Cminor nécessaire à l'évaluation du corps de fonction.

De même, calculer les variables de la future fonction Cminor revient à relever l'ensemble des variables apparaissant dans le corps.

6.4.4 Spécification de l'interaction avec un gestionnaire de mémoire via la sémantique

Dans l'optique de prouver la préservation sémantique lors de la génération de code Cminor, et de vérifier formellement l'interaction avec le gestionnaire de mémoire encodé dans cette génération de code, nous devons nous rapprocher de l'approche sémantique de Cminor.

Dans les sémantiques de langages fonctionnels présentées jusqu'ici, une variable était stockée dans l'environnement local jusqu'à la fin de sa portée lexicale. Concernant les futures racines, elles seront stockées sur la structure dédiée à l'enregistrement des racines jusqu'à la fin de leur portée lexicale.

Après l'évaluation d'un potentiel déclencheur de GC, les variables de l'environnement local, c'est-à-dire les valeurs des variables non racines peuvent avoir été effacées par le GC. D'ailleurs, ces valeurs par définition sont inutiles à la suite du calcul. Un moyen de garantir leur inutilité est d'écraser l'environnement de variables locales à la sortie de l'évaluation d'un déclencheur potentiel de GC. Or, ce comportement ne se reflète ni dans la sémantique de Cminor ni dans celle de Mon.

Nous aimerions pouvoir parler des environnements avant et après une évaluation. Ainsi, nous pourrions caractériser l'évaluation d'un déclencheur potentiel de GC par :

– la préservation du contenu de l'ensemble des racines ;
– l'écrasement de l'environnement de variables locales.

Notre jugement d'évaluation sera donc de la forme

$$e_{\texttt{var}}, e_{\texttt{root}} \vdash t \Rightarrow v, e'_{\texttt{var}}, e'_{\texttt{root}},$$

où $e_{\texttt{var}}, e_{\texttt{root}}$ sont les environnements au début de l'évaluation de t et $e'_{\texttt{var}}, e'_{\texttt{root}}$ les environnements à la fin de l'évaluation. En particulier, si t est un déclencheur potentiel de GC, on aura $e'_{\texttt{var}} = \varepsilon$ et $e'_{\texttt{root}} = e_{\texttt{root}}$, reflétant le fait que seules les variables de l'ensemble des racines sont préservées par le GC. Un tel comportement d'évaluation nous permettra de vérifier formellement notre calcul de racines et notre interaction avec un GC.

6.4.5 Le langage Fminor

Le langage intermédiaire Fminor est un langage purement fonctionnel répondant à l'ensemble des observations ci-dessus.

Syntaxe

La syntaxe de Fminor est donnée par la grammaire suivante :

Atomes :
$$\begin{array}{lll} a ::= & \texttt{var}(x) & \\ \mid & \texttt{field}\ n\ a & \\ \mid & \texttt{root}(n) & n\text{-ième racine courante} \end{array}$$

Termes :
$$\begin{array}{lll} t ::= & a & \\ \mid & a\ [a_0; \ldots; a_n]^z & \\ \mid & f\ (a_1; \ldots; a_n)^z & \\ \mid & \texttt{let var}\ x = t_1\ \texttt{in}\ t_2 & \text{lie } t_1 \text{ à } \texttt{var}(x) \text{ dans } t_2 \\ \mid & \texttt{let root}\ n\ =\ t_1\ \texttt{in}\ t_2 & \text{lie } t_1 \text{ à } \texttt{root}(n) \text{ dans } t_2 \\ \mid & C\ (a_1; \ldots; a_n)^z & \\ \mid & \texttt{match}\ a\ \texttt{with}\ \pi_0; \ldots; \pi_n & \end{array}$$

Motifs :
$$\pi ::= p_1; \ldots; p_n \rightarrow t$$

Paramètres de motif :
$$\begin{array}{lll} p ::= & \texttt{var}(x) & \\ \mid & \texttt{root}(n) & \end{array}$$

Fonctions :
$$def ::= f, \{\texttt{params}\ :\ x_0; \ldots; x_n\ ;\ \texttt{rootsize}\ :\ int\ ;\ \texttt{vars}\ : y_1; \ldots; y_m\ ;\ \texttt{body}\ :\ t\}$$

Programme :
$$prog ::= \{\texttt{defs}\ :\ \vec{def}\ ;\ \texttt{main}:\ t\}$$

Un programme p Fminor s'organise en une liste de déclarations de fonctions $p.\texttt{defs}$ et un terme principal $p.\texttt{main}$.
Une déclaration de fonction f Fminor est plus riche que celle d'une fonction Mon et très proche de celle d'une fonction Cminor. En plus de la liste de paramètres $f.\texttt{params}$ et du corps $f.\texttt{body}$, la déclaration de fonction comporte une indication du nombre maximal de racines durant l'évaluation du corps de fonction lors d'une application, $f.\texttt{rootsize}$. De plus, la déclaration de fonction, liste les variables locales au corps de fonction, $f.\texttt{vars}$.
La matérialisation de la structure de données dédiée à l'enregistrement des racines se concrétise syntaxiquement par l'apparition, parmi les atomes, d'un accès à un élément de l'ensemble des racines, $\texttt{root}(n)$ accède à la n-ième racine courante. Parmi les termes, il est possible de stocker un calcul comme racine de la fonction courante, $\texttt{let root}\ n\ =\ t_1\ \texttt{in}\ t_2$ place comme $n$ème racine le terme t_1 dans t_2. Enfin, les motifs de filtrage reflètent aussi

cette matérialisation de la structure dédiée. Un paramètre formel p d'un motif est soit une variable $\texttt{var}(x)$ soit un emplacement dans l'ensemble des racines $\texttt{root}(n)$.

Sémantique naturelle avec environnements

La sémantique est définie selon la même stratégie que les langages fonctionnels précédemment présentés. De plus, elle reflète d'assez près l'évaluation du programme Cminor, qui sera produite par la traduction décrite au chapitre 7. Elle suit l'évolution de la structure dédiée à l'enregistrement des racines au cours de l'évaluation au travers d'un environnement des racines et en retournant un environnement modifié par l'évaluation. La sémantique de Fminor s'exprime au travers de deux jugements :

Jugement d'évaluation d'un atome : $S, e_{\texttt{var}}, e_{\texttt{root}} \vdash a \Rightarrow v$,
dans l'environnement formé de la liste de déclarations de fonctions S, l'environnement de variables $e_{\texttt{var}}$ et l'environnement des racines $e_{\texttt{root}}$, l'atome a s'évalue en la valeur v.

Jugement d'évaluation d'un terme : $S, e_{\texttt{var}}, e_{\texttt{root}} \vdash t \Rightarrow v, e'_{\texttt{var}}, e'_{\texttt{root}}$,
dans l'environnement initial formé de S, $e_{\texttt{var}}$ et $e_{\texttt{root}}$ le terme t s'évalue en la valeur v en modifiant l'environnement de variables en $e'_{\texttt{var}}$ et l'environnement des racines en $e'_{\texttt{root}}$.

Les valeurs sémantiques sont de la même forme que celle de Mon. Un environnement de variables se manipule comme ceux de Mon. Un environnement de racines $e_{\texttt{root}}$ associe à un emplacement, désigné par un entier n, une valeur v. Tout comme un environnement de variables, on peut :

- consulter la valeur associée à un emplacement n, $e_{\texttt{root}}(n) = v$. Si $e_{\texttt{root}}$ n'associe aucune valeur à l'emplacement n, alors $e_{\texttt{root}}(n) = \bot$;
- associer à un emplacement n une valeur v dans $e_{\texttt{root}}$, $e_{\texttt{root}}[n \leftarrow v]$;
- l'environnement vide ε n'associe aucune valeur à tous les emplacements.

Les paramètres d'une clause de filtrage sont soit des variables $\texttt{var}(x)$ soit des racines $\texttt{root}(n)$. Selon le cas, la valeur v associée à un paramètre de clause p lors de l'évaluation d'un filtrage sera stockée soit dans l'environnement de variable $e_{\texttt{var}}$ si c'est une variable $p = \texttt{var}(x)$ soit dans l'environnement des racines $e_{\texttt{root}}$ si c'est une racine $p = \texttt{root}(n)$. Notons l'ajout dans les environnements d'une liaison d'un paramètre de clause de filtrage p à une valeur v par :

$$(e_{\texttt{var}}, e_{\texttt{root}})[p \leftarrow v] \quad = \quad \begin{cases} (e_{\texttt{var}}[x \leftarrow v], e_{\texttt{root}}) & \text{si } p = \texttt{var}(x) \,; \\ (e_{\texttt{var}}, e_{\texttt{root}}[n \leftarrow v]) & \text{si } p = \texttt{root}(n) \end{cases}$$

Lors de l'évaluation d'un filtrage, le sujet doit s'évaluer en un constructeur appliqué $(C, v_1; \ldots; v_n)$. La liste de valeurs ainsi obtenue est alors répartie dans les environnements afin d'être liée aux paramètres de la C-ième clause $p_1; \ldots; p_n$. Soit :

$$(e_{\texttt{var}}, e_{\texttt{root}})[(p_1; \ldots; p_n) \leftarrow (v_1; \ldots; v_n)] \quad = \quad (e_{\texttt{var}}, e_{\texttt{root}})[p_1 \leftarrow v_1] \ldots [p_n \leftarrow v_n]$$

Lors de l'évaluation d'un terme ou d'un atome, la liste de définitions de fonctions S est globale, elle ne subit aucune modification. L'ensemble des règles d'évaluation de Fminor est listé sur la figure 6.4.5.

$$\frac{e_{\texttt{var}}(x) \;=\; v}{S, e_{\texttt{var}}, e_{\texttt{root}} \vdash \texttt{var}(x) \Rightarrow v} \qquad \frac{e_{\texttt{root}}(n) \;=\; v}{S, e_{\texttt{var}}, e_{\texttt{root}} \vdash \texttt{root}(n) \Rightarrow v}$$

$$\frac{S, e_{\texttt{var}}, e_{\texttt{root}} \vdash a \Rightarrow (f, v_1; \ldots; v_n) \qquad 1 \leq i \leq n}{S, e_{\texttt{var}}, e_{\texttt{root}} \vdash \texttt{field}\ i\ a \Rightarrow v_i}$$

$$\frac{S, e_{\texttt{var}}, e_{\texttt{root}} \vdash t_1 \Rightarrow v', e'_{\texttt{var}}, e'_{\texttt{root}} \qquad S, e'_{\texttt{var}}[x \;\leftarrow\; v'], e'_{\texttt{root}} \vdash t_2 \Rightarrow v, e''_{\texttt{var}}, e''_{\texttt{root}}}{S, e_{\texttt{var}}, e_{\texttt{root}} \vdash \texttt{let var}\ x \;=\; t_1\ \texttt{in}\ t_2 \Rightarrow v, e''_{\texttt{var}}, e''_{\texttt{root}}}$$

$$\frac{S, e_{\texttt{var}}, e_{\texttt{root}} \vdash t_1 \Rightarrow v', e'_{\texttt{var}}, e'_{\texttt{root}} \qquad S, e'_{\texttt{var}}, e'_{\texttt{root}}[n \;\leftarrow\; v'] \vdash t_2 \Rightarrow v, e''_{\texttt{var}}, e''_{\texttt{root}}}{S, e_{\texttt{var}}, e_{\texttt{root}} \vdash \texttt{let root}\ n \;=\; t_1\ \texttt{in}\ t_2 \Rightarrow v, e''_{\texttt{var}}, e''_{\texttt{root}}}$$

$$\frac{\begin{array}{c} S, e_{\texttt{var}}, e_{\texttt{root}} \vdash a \Rightarrow (C,\ v_1; \ldots; v_k) \qquad \pi_C \;=\; p_1; \ldots; p_k \rightarrow t \\ (e_{\texttt{var}}, e_{\texttt{root}})[p_1; \ldots; p_k \leftarrow v_1; \ldots; v_k] = (e'_{\texttt{var}}, e'_{\texttt{root}}) \qquad S, e'_{\texttt{var}}, e'_{\texttt{root}} \vdash t \Rightarrow v, e''_{\texttt{var}}, e''_{\texttt{root}} \end{array}}{S, e_{\texttt{var}}, e_{\texttt{root}} \vdash \texttt{match}\ a\ \texttt{with}\ \pi_0; \ldots; \pi_n \Rightarrow v, e''_{\texttt{var}}, e''_{\texttt{root}}}$$

$$\frac{(f, d) \in S \qquad |n|_t \qquad |z|_p \qquad S, \varepsilon, e_{\texttt{root}} \vdash a_i \Rightarrow v_i \qquad 1 \leq i \leq n}{S, e_{\texttt{var}}, e_{\texttt{root}} \vdash f\ (a_1; \ldots; a_n)^z \Rightarrow (f,\ v_1; \ldots; v_n), \varepsilon, e_{\texttt{root}}}$$

$$\frac{|n|_t \qquad |z|_p \qquad S, \varepsilon, e_{\texttt{root}} \vdash a_i \Rightarrow v_i \qquad 1 \leq i \leq n}{S, e_{\texttt{var}}, e_{\texttt{root}} \vdash C\ (a_1; \ldots; a_n)^z \Rightarrow (C, v_1; \ldots; v_n), \varepsilon, e_{\texttt{root}}}$$

$$\frac{\begin{array}{c} |z|_p \qquad S, e_{\texttt{var}}, e_{\texttt{root}} \vdash a \Rightarrow (f, v_1; \ldots; v_k) \qquad (f, d) \in S \qquad d.\texttt{params} = x_1; \ldots; x_n \\ S, e_{\texttt{var}}, e_{\texttt{root}} \vdash a_i \Rightarrow v_i \qquad 1 \leq i \leq n \qquad S, \varepsilon[x_0 \;\leftarrow\; v_0] \ldots [x_n \leftarrow v_n], \varepsilon \vdash (\texttt{body}\ d) \Rightarrow v \end{array}}{S, e_{\texttt{var}}, e_{\texttt{root}} \vdash a\ [a_1; \ldots; a_n]^z \Rightarrow v, \varepsilon, e_{\texttt{root}}}$$

FIG. 6.4.1 – Sémantique à grand pas en appel par valeur avec environnement de Fminor

Une variable s'évalue comme la valeur qui lui est associée dans l'environnement de variable et une racine s'évalue comme la valeur qui lui est associée dans l'environnement de racines.

Évaluation des lieurs

Dans les environnements initiaux $e_{\texttt{var}}$ et $e_{\texttt{root}}$, une liaison locale $\texttt{let var}\ x \;=\; t_1\ \texttt{in}\ t_2$ s'évalue en v en produisant les environnements finaux $e''_{\texttt{var}}$ et $e''_{\texttt{root}}$ si :

- t_1 dans les environnements initiaux s'évalue en v' en produisant les environnements intermédiaires $e'_{\texttt{var}}$ et $e'_{\texttt{root}}$.
- t_2 dans les environnements intermédiaires enrichies de la nouvelle liaison locale $e'_{\texttt{var}}[x \;\leftarrow\; v']$

et $e'_{\texttt{root}}$, s'évalue en v et produit les environnements finaux.

De même, dans les environnements initiaux $e_{\texttt{var}}$ et $e_{\texttt{root}}$, l'enregistrement de la racine dans l'ensemble de racines locales `let root` $n = t_1$ `in` t_2 s'évalue en v en produisant les environnements finaux $e''_{\texttt{var}}$ et $e''_{\texttt{root}}$, si :

t_1 dans les environnements initiaux s'évalue en v' en produisant les environnements intermédiaires $e'_{\texttt{var}}$ et $e_{\texttt{root}}$.

t_2 dans les environnements intermédiaires enrichies du nouvel enregistrement de racine locale $e'_{\texttt{root}}[n \leftarrow v']$ et $e'_{\texttt{var}}$, s'évalue en v et produit les environnements finaux.

Un filtrage `match` a `with` $\pi_1; \ldots; \pi_n$ s'évalue, dans les environnements initiaux, comme le corps Cème choix $\pi_C = p_1; \ldots; p_k \rightarrow t$ dans les environnements produits par $(e_{\texttt{var}}, e_{\texttt{root}})[v_1; \ldots; v_k \leftarrow p_1; \ldots; p_k] = (e'_{\texttt{var}}, e'_{\texttt{root}})$.

Spécification de l'interaction avec un gestionnaire de mémoire Elle est faite à travers des règles d'évaluation des déclencheurs directs de GC. L'évaluation d'un terme déclencheur direct de GC (fermeture, constructeur appliqué et application) est de la forme :

$$S, e_{\texttt{var}}, e_{\texttt{root}} \vdash t \Rightarrow v, \varepsilon, e_{\texttt{root}}.$$

Ce jugement reflète le comportement post GC :

Les racines sont préservées, on retourne l'environnement de racines initial.

Les variables ne sont pas préservées par le GC : leurs valeurs peuvent devenir invalides après déclenchement du GC. Pour garantir que les valeurs des variables ne seront pas utilisées dans la suite du calcul, on vide donc l'environnement de variables, en le remplaçant par l'environnement vide ε.

Les évaluations des déclencheurs directs de GC, comportent aussi des vérifications relatives aux tailles des futurs blocs mémoires Cminor. Ils portent syntaxiquement le nombre d'éléments présents dans l'ensemble des racines z. L'évaluation n'est effective que si z correspond à une taille de bloc de pile Cminor correcte :$|z|_p$. De plus, les structures de données produiront de futurs blocs de tas :

- $(f, a_1; \ldots; a_n)^z$ produit un bloc de tas de $n+1$ éléments de 4 mots mémoire, si $|n|_t$ alors l'évaluation est effective.
- $C\ (a_1; \ldots; a_n)^z$ produit un bloc de tas de $n+1$ éléments de 4 mots mémoire, si $|n|_t$ alors l'évaluation est effective.

Ces tests sont ajoutés aux prémisses des règles d'évaluation.

Une fermeture minimale explicite, si elle concerne une fonction f définie dans S et que ses éléments $a_1; \ldots; a_n$ s'évaluent en $v_1; \ldots; v_n$, s'évalue en $(f, v_1; \ldots; v_n)$.
De même, un constructeur appliqué $C\ (a_1; \ldots; a_n)^z$ s'évalue en $(C, v_1; \ldots; v_n)$, si $a_1; \ldots; a_n$ s'évalue en $v_1; \ldots; v_n$ dans l'environnement de variables vide et l'environnement des racines courant.
Une application $a\ [a_1; \ldots; a_n]^z$ s'évalue comme le corps de la fonction mise en jeu d (celle indiquée par la valeur fermeture d'évaluation de a) dans l'environnement de variables associant aux paramètres de d les valeurs $v_1; \ldots; v_n$, où $v_1; \ldots; v_n$ sont les évaluations de $a_1; \ldots; a_n$ dans l'environnement de variables vide et l'environnement de racines courant. Pour que l'évaluation réussisse, il faut :

- $(f, d) \in S$,

– $|d.\texttt{rootsize}|_p$,
– $\forall x, x \in d.\texttt{params} \rightarrow x \notin lid$.

Les évaluations en Fminor effectuent aussi les mêmes vérifications que les évaluations de Mon concernant les noms de variables. Tout les noms liés par `let`, au niveau des paramètre formels de motif et des paramètres formels d'une fonction sont disjoints de la liste *lid* de noms de variables spéciaux.

6.5 Génération de code Fminor

6.5.1 Algorithme

La traduction de programme Mon en programme Fminor ne fait pas que calculer les racines futures, elle effectue aussi des vérifications concernant les tailles des futurs blocs mémoire par les tests définis à la section 6.4.2.

Traduction d'un terme Lors de la traduction d'un terme Mon en un terme Fminor, la traduction d'une racine du terme diffère de celle d'une variable non racine. Cette information est connue au moment de la liaison de la variable et est utilisée lorsque l'on traduit cette même variable. La traduction se fait donc au travers d'un environnement. Cet environnement Γ est composé :

d'une fonction associant à une variable x une information γ.

$$\gamma ::= \texttt{var}(x) | \texttt{root}(n).$$

– Si x n'est pas une racine alors $\Gamma(x) = \texttt{var}(x)$,
– Si x est une racine alors $\Gamma(x) = \texttt{root}(n)$, indique que x deviendra le $n$ième élément de ensemble des racines,
– Si Γ n'associe aucune information à une variable x, alors $\Gamma(x) = \perp$.

d'un entier : un compteur représentant le nombre de racine couramment enregistrées sur la structure de données dédiée. Ce compte de racines est noté $|\Gamma|$. C'est aussi le prochain emplacement que Γ attribuera à une racine.

La fonction `add` qui prend en argument un identifiant x, un environnement de traduction Γ et un ensemble de variable R (typiquement l'ensemble des racines dans lequel la variable est liée) effectue l'ajout d'un élément dans un environnement de traduction Γ de la manière suivante :

$$\texttt{add}\ x\ \Gamma\ R \quad = \quad \begin{cases} (\texttt{root}(|\Gamma|)\ ,\ \{\Gamma[x \leftarrow \texttt{root}(|\Gamma|)]\ ,\ (|\Gamma|+1))\} & \text{si } x \in R \text{ et } ||\Gamma|+1|_p\,; \\ (\texttt{var}(x)\ ,\ \{\Gamma[x \leftarrow \texttt{var}(x)]\ ,\ |\Gamma|\}) & \text{si } x \notin R \end{cases}$$

Si x n'appartient pas à R alors x sera associé à $\texttt{var}(x)$ dans Γ et le compte de racines ne change pas. Sinon (si x appartient à R), on vérifie que le compte de racines incrémenté de 1 est compatible avec une une taille correcte pour un futur bloc : $||\Gamma|+1|_p$. Si le test échoue, la traduction échoue. Sinon, x est associée à $\texttt{root}(|\Gamma|)$ et le compte de racines sera incrémenté de 1. L'addition d'un élément dans l'environnement retourne le nouvel environnement et le paramètre formel Fminor correspondant à la nature de la variable. $\texttt{root}(|\Gamma|)$ si la variable

deviendra le ($|\Gamma|$)ème élément de l'ensemble des racines, $\texttt{var}(x)$ si elle sera la variable x Fminor.

Ensuite, nous définissons $\texttt{adds}\ x_1;\ldots;x_n\ \Gamma\ R$, qui effectue (n) `add` séquentiellement et retourne la liste de paramètres formels de motif et l'environnement modifié.

Nous sommes maintenant armés pour décrire la traduction d'un terme Mon en un terme Fminor :

$$
\begin{array}{rcl}
[\![x]\!]_\Gamma & = & \begin{cases} \texttt{var}(x) & \text{si } \Gamma(x) = \texttt{var}(x)\,; \\ \texttt{root}(n) & \text{si } \Gamma(x) = \texttt{root}(n) \end{cases} \\
[\![\texttt{field}\ n\ a]\!]_\Gamma & = & \texttt{field}\ n\ [\![a]\!]_\Gamma \\
[\![a\ [a_0;\ldots;a_n]]\!]_\Gamma & = & [\![a]\!]_\Gamma\ [[\![a_0]\!]_\Gamma;\ldots;[\![a_n]\!]_\Gamma]^{|\Gamma|} \\
[\![f\ (a_1;\ldots;a_n)]\!]_\Gamma & = & f\ ([\![a_1]\!]_\Gamma;\ldots;[\![a_n]\!]_\Gamma)^{|\Gamma|}\text{si } |n|_t \\
[\![C\ (a_1;\ldots;a_n)]\!]_\Gamma & = & C\ ([\![a_1]\!]_\Gamma;\ldots;[\![a_n]\!]_\Gamma)^{|\Gamma|}\text{si } |n|_t \\
[\![\texttt{let}\ x\ =\ t_1\ \texttt{in}\ t_2]\!]_\Gamma & = & \begin{cases} \texttt{let var}\ x\ =\ [\![t_1]\!]_\Gamma\ \texttt{in}\ [\![t_2]\!]_{\Gamma'} & \text{si } \texttt{add}\ x\ \Gamma\ R(t_2) = (\texttt{var}(x),\Gamma')\,; \\ \texttt{let root}\ n\ =\ [\![t_1]\!]_\Gamma\ \texttt{in}\ [\![t_2]\!]_{\Gamma'} & \text{si } \texttt{add}\ x\ \Gamma\ R(t_2) = (\texttt{root}(n),\Gamma') \end{cases} \\
[\![\texttt{match}\ a\ \texttt{with}\ \pi_0;\ldots;\pi_n]\!]_\Gamma & = & \texttt{match}\ [\![a]\!]_\Gamma\ \texttt{with}\ [\![\pi_0;\ldots;\pi_n]\!]_\Gamma \\
[\![x_1;\ldots;x_n \to t]\!]_\Gamma & = & p_1;\ldots;p_n \to [\![t]\!]_{\Gamma'} \\
 & & \text{avec } (p_1;\ldots;p_n,\Gamma') = \texttt{adds}\ (x_1;\ldots;x_n)\ \Gamma\ R(t)
\end{array}
$$

La traduction d'une variable x dépend de l'information γ qui lui est associée dans Γ. Si γ est $\texttt{var}(x)$ alors elle est traduite vers la variable Fminor x . Si $\gamma = \texttt{root}(n)$ alors elle est traduite par $\texttt{root}(n)$. La traduction de l'accès à un élément de fermeture `field` n a produit l'accès au $n$ième élément de la traduction de a.
La traduction d'un appel de fonction $a\ [a_0;\ldots;a_n]$ se fait récursivement sur les atomes le constituant et collecte le compte de racines : $[\![a]\!]_\Gamma\ [[\![a_0]\!]_\Gamma;\ldots;[\![a_n]\!]_\Gamma]^{|\Gamma|}$.
La traduction des structures de données se fait aussi récursivement sur les atomes les constituants et le compte de racines courantes. Néanmoins, lors de la traduction d'une structure de donnée, si la taille du futur bloc de tas la représentant n'est pas correcte, la traduction échoue.
Lors de la traduction d'une liaison locale $\texttt{let}\ x\ =\ t_1\ \texttt{in}\ t_2$ dans l'environnement de traduction Γ, t_1 est traduit dans Γ. Selon que x est une racine ou pas de t_2, la liaison locale sera traduite différemment.

x **n'est pas racine de** t_2 **:** la liaison locale Mon sera traduite par la liaison `let var` liant x à $[\![t_1]\!]_\Gamma$ dans la traduction de t_2 dans l'environnement actualisé.

x **est une racine de** t_2 **:** la liaison locale devient une liaison `let root` de $[\![t_1]\!]_\Gamma$, à la position indiquée par l'actualisation de l'environnement dans lequel t_2 est traduit.

La traduction d'un filtrage se fait récursivement sur l'atome qu'il discrimine et ses motifs. La traduction d'un motif $x_1;\ldots;x_n \to t$ est telle que pour pour tout x_i, selon que x_i est une racine de t ou non, `adds` retourne le paramètre formel de filtrage correspondant, dans la liste $p_1;\ldots;p_n$. Le corps du motif est alors traduit dans l'environnement produit par `adds`.

Traduction d'une fonction Les fonctions Mon ont deux champs : le premier répertorie ses paramètres formels, `params` et le second contient un terme qui est son corps, `body`.

Les fonctions Fminor contiennent, en plus de ces deux champs, des informations plus proches de celles contenues par les fonctions Cminor. Le champ `vars` contient la liste des futures variables locales de la fonction Cminor, et le champ `rootsize` le nombre de racines maximal au cours de l'exécution du corps de la fonction (sur-approximation de la taille du bloc de pile Cminor nécessaire à l'exécution du corps de la fonction).

Lors de la traduction d'une fonction Mon en une fonction Fminor, nous conservons les paramètres formels. Cependant, certains paramètres formels peuvent être des racines dans le corps de la fonction.

Afin de prendre en considération ce fait, chaque traduction de corps de fonction Mon se retrouvera sous terme droite d'une imbrication de liaisons locales de tels paramètres formels dans l'ensemble des racines.

Considérons la fonction Mon :

$$\{\texttt{params } = x_0; \ldots; x_n; \texttt{ body } = t\}$$

Lors de la traduction de t, on initialise l'environnement de traduction avec $x_0; \ldots; x_n$. Selon que $x_i \in R(t)$ ou pas, on lui associe une information de traduction γ. Pour le premier paramètre formel, x_0 :

- si $x_0 \notin R(t)$, alors $\gamma = \texttt{var}(x_0)$,
- si $x_0 \in R(t)$, alors $\gamma = \texttt{Root}(0)$ et $|\Gamma_0| = 1$.

On parcourt ainsi la liste de paramètres formels comme pour une clause de filtrage. De plus, cette initialisation de l'environnement de traduction, produit une liste qui associe à chaque x_i racine de t l'emplacement que l'environnement lui a associé.

Notons `init_env`, la fonction qui produit l'environnement de traduction initial de la traduction d'un corps de fonction t à partir de la liste de paramètres formels $x_0; \ldots; x_n$ et de l'ensemble des racines de t, $R(t)$. En plus de l'environnement initial Γ, `init_env` fournit la liste l qui pour chaque paramètre formel racine dans le corps de fonction associe l'emplacement attribué par Γ. Soit,

$$\texttt{init\_env } (x_0; \ldots; x_n)\ R(t) = (\Gamma, [\![x_0; \ldots; x_n]\!]).$$

On peut alors traduire le corps de la fonction Mon dans Γ.

Il reste alors à enregistrer les paramètres formels racines dans le corps de la fonction dans la structure de données dédiée. Pour cela, on parcourt la liste $[\![x_0; \ldots; x_n]\!]$ de gauche à droite et on séquence autant de liaisons locales de racines que de couples de $[\![x_0; \ldots; x_n]\!]$. Pour chaque couple de $[\![x_0; \ldots; x_n]\!]$ de la forme (x_j, n) on crée la liaison $\texttt{let root } n = x_j \texttt{ in } \ldots$ Par exemple, si $[\![x_0; \ldots; x_n]\!] = (x_j, 0), (x_k, 1)$ alors le corps de la fonction Fminor est :

$$\texttt{let root } 0 = x_j \texttt{ in let root } 1 = x_k \texttt{ in } [\![t]\!]_\Gamma.$$

Notons `let root seq` l'imbrication des liaisons locales de racines selon une liste fournit par `env_init` $[\![x_0; \ldots; x_n]\!]$ dont le dernier sous-terme droit est t. Le corps de la fonction Fminor générée est donc :

$$\texttt{let root seq } [\![x_0; \ldots; x_n]\!] \texttt{ in } [\![t]\!]_\Gamma.$$

A partir de la traduction du corps de la fonction Mon, $[\![t]\!]_\Gamma$, on peut statiquement obtenir la liste des variables locales de la future fonction Cminor.

Calculer les variables locales à un terme Fminor revient à répertorier les variables apparaissant dans le terme et non liées au niveau d'un motif de filtrage. En effet, un paramètre formel d'une clause de filtrage deviendra une lecture à un offset sur la valeur du sujet filtré. On note $V(t)$ l'ensemble des variables locales dans le terme t.

Tout comme pour la liste des futures variables locales de la fonction Cminor, on peut déterminer statiquement le compte de racines nécessaire à l'exécution de la fonction Cminor qui sera générée.

Plus précisément, on peut surestimer cette taille en récupérant, syntaxiquement, le plus grand emplacement dans l'ensemble des racines liées dans le terme. De plus, lors de ce calcul, on peut effectuer des vérifications de cohérence entre les emplacements attribués et les emplacements utilisés. Ainsi, si le dernier emplacement attribué est n : `let root` $n =$ t_0 `in` t_1, alors toute occurrence de `root`(m) dans t_1 est telle que $m \leq n$.

On note `root_space` la fonction qui surestime la taille de l'ensemble des racines maximale nécessaire à l'exécution du corps de fonction t et qui vérifie la cohérence de la matérialisation de l'ensemble des racines. Nous pouvons maintenant définir la traduction d'une fonction Mon en une fonction Fminor :

$$
\begin{array}{rcl}
[\![d]\!] & = & \texttt{let } (\Gamma, l) := \texttt{init\_env } d.\texttt{params } R(d.\texttt{body}) \texttt{ in} \\
 & & \quad \texttt{let } t := [\![d.\texttt{body}]\!]_\Gamma \texttt{ in} \\
 & & \quad\quad \texttt{let } q := \texttt{root\_space } t \texttt{ in si } |q|_p \\
 & & \quad\quad \{\texttt{params} := d.\texttt{params} ; \\
 & & \quad\quad \texttt{rootsize} := q; \\
 & & \quad\quad \texttt{vars} := V(t) \setminus \{d.\texttt{params}\}; \\
 & & \quad\quad \texttt{body} := \texttt{let root seq } l \texttt{ in } t\}
\end{array}
$$

Traduction d'un programme Enfin, nous pouvons traduire un programme Mon en un programme Fminor. Si p est un programme Mon, alors sa traduction Fminor a pour liste de fonctions $[\![p.\texttt{defs}]\!]$ et pour fonction principale la traduction de $p.\texttt{main}$ dans l'environnement de traduction vide. La traduction échoue si la taille maximale potentielle du futur bloc de pile Cminor n'est pas une taille possible pour un futur bloc mémoire.

6.5.2 Propriété syntaxique : des programmes bien bornés

Une propriété importante des programmes Fminor générés concerne la cohérence des emplacements apparaissant dans les corps des fonctions vis-à-vis des nombres maximaux de racines de ces mêmes fonctions. Autrement dit, si d est une fonction Fminor générée, alors toute occurrence `root`(n) dans $d.\texttt{body}$ est un emplacement valable dans l'ensemble de racines, ie. $n \leq d.\texttt{rootsize}$.

Afin de définir cette propriété, nous avons besoin de définir une relation entre ce nombre et un terme. La relation syntaxique $\|t\| \leq z$ exprime que les emplacements apparaissant

dans le terme t sont bornés par z. Cette relation se définit par le jeu de règles d'inférence suivant :

$$\|\texttt{var}(x)\| \leq z \qquad \frac{n \leq z}{\|\texttt{root}(n)\| \leq z} \qquad \frac{\|a\| \leq z}{\|\texttt{field}\ n\ a\| \leq z}$$

$$\frac{z' \leq z \quad \|a\| \leq z \quad \|a_i\| \leq z \text{ pour tout } i \in [0,n]}{\|a[a_0;\ldots;a_n]^{z'}\| \leq z}$$

$$\frac{z' \leq z \quad \|a_i\| \leq z \text{ pour tout } i \in [1,n]}{\|f\ (a_1;\ldots;a_n)^{z'}\| \leq z} \qquad \frac{z' \leq z \quad \|a_i\| \leq z \text{ pour tout } i \in [1,n]}{\|C\ (a_1;\ldots;a_n)^{z'}\| \leq z}$$

$$\frac{\|t_1\| \leq z \quad \|t_2\| \leq z}{\|\texttt{let var}\ x = t_1\ \texttt{in}\ t_2\| \leq z} \qquad \frac{n \leq z \quad \|t_1\| \leq z \quad \|t_2\| \leq z}{\|\texttt{let root}\ n = t_1\ \texttt{in}\ t_2\| \leq z}$$

$$\frac{\|a\| \leq z \quad \|\pi_i\| \leq z \text{ pour tout } i \in [0,n]}{\|\texttt{match}\ a\ \texttt{with}\ \pi_0;\ldots;\pi_n\| \leq z}$$

$$\frac{\text{pour tout } i \in [1,n] \;\; p_i = \texttt{root}(k),\; k \leq z \quad \|t\| \leq z}{\|p_1;\ldots;p_n \to t\| \leq z}$$

Une fonction est bien bornée si tous les emplacements dans la structure de données dédiée à l'enregistrement des racines de son corps sont bornés par la taille que son exécution nécessite.

$$\frac{\|d.\texttt{body}\| \leq d.\texttt{rootsize}}{\|d\|}$$

Enfin, un programme est bien borné si toute ses fonctions le sont.

$$\frac{\text{pour tout } (f,d) \in p.\texttt{defs} \quad \|d\| \quad \|p.\texttt{main}\|}{\|p\|}$$

Nous avons validé notre surestimation du nombre maximal de racines d'une fonction et notre matérialisation de la structure de données dédiée à l'enregistrement des racines par notre génération de code Fminor.

Théorème 6.5.1 (Programme bien borné)
Un programmes Fminor généré est bien borné : $\|[\![p]\!]\|$.

Ce résultat joue un rôle crucial dans la preuve de préservation sémantique de génération de code Cminor. Comme nous le verrons au prochain chapitre, ces bornes permettent de conditionner les accès dans le futur bloc de pile Cminor courant.

6.5.3 Préservation sémantique

Nous avons montré la préservation sémantique de la matérialisation de la structure dédiée à l'enregistrement des racines. La préservation sémantique de cette transformation forme aussi la vérification formelle de l'interaction avec un gestionnaire de mémoire :

Théorème 6.5.2
Si le programme Mon p s'évalue en v, alors il existe v' telle que : $[\![p]\!] \Rightarrow v'$ *et* $p.\texttt{defs}, [\![p]\!].\texttt{defs} \vdash v \sim v'$.

La preuve nécessite une simulation sur l'évaluation d'un terme Mon. Cette simulation définit une équivalence observationnelle. Nous avons donc besoin de construire un cadre d'observation d'équivalence au travers d'une correspondance entre valeurs sémantiques, s'étendant aux environnements d'évaluation. Une évaluation Fminor, en plus de fournir une valeur d'évaluation à un terme en consultant les environnements, mime l'évolution de la future pile Cminor autrement dit la gestion des racines lors de l'évaluation d'un programme. La preuve de préservation sémantique de la traduction d'un programme Mon en un programme Fminor vérifie aussi la bonne gestion des racines en vue de l'interaction avec le GC.

Nous commençons par définir les invariants de la simulation, et montrons les constructions et préservation par rapport aux diverses modifications des environnements lors des évaluations.

Les invariants de la simulation

Correspondance des listes de fonctions Lors de l'évaluation d'un programme Mon, tout comme lors de l'évaluation d'un programme Fminor, le terme principal a besoin de consulter la liste de fonctions globales du programme lors de son évaluation. Lors de la simulation, nous avons besoin d'une correspondance entre la liste de fonctions globales Mon S et la liste de fonction globales Fminor S'. En effet, il faut qu'à chaque nom de fonction f de S de définition d, S' associe la traduction de la définition de d.

Définition 6.5.3 (Correspondance des environnements globaux) *La liste de fonctions globales Mon S correspond à la liste de fonctions globales Fminor S' si* $\forall f,\ S(f) = d$ *implique* $S'(f) = [\![d]\!]$. *Nous notons cette correspondance :* $S \sim S'$.

La correspondance entre les listes de fonctions globales se construit au début de la preuve de préservation sémantique de la traduction d'un programme Mon p, si celui ci s'évalue. En effet, s'il s'évalue alors sa liste de fonctions globales associe à chaque nom de fonction une unique définition. Et par construction $p.\texttt{defs} \sim [\![p.\texttt{defs}]\!]$.

Correspondance entre valeurs sémantiques Les valeurs sémantiques des deux langages sont celles représentant les constructeurs appliqués et les fermetures. Les fermetures se référant aux fonctions globales, la correspondance entre valeurs sémantiques est donc paramétrée par les deux listes de fonctions globales qui sont en correspondance. En effet, la fermeture Mon $(f, v_1; \ldots; v_n)$ correspond à la fermeture Fminor $(f, w_1; \ldots; w_n)$, si dans la liste de fonctions globales Mon S, f est associé à d et dans S', f est associé à $[\![d]\!]$. Or $S \sim S'$, et il suffit que chacune des listes associe une définition à f. De plus, on propage

cette correspondance : à la liste $v_1; \ldots; v_n$ correspond la liste $w_1; \ldots; w_n$. De même, la valeur sémantique d'un constructeur appliqué correspond à la valeur sémantique du constructeur appliqué de même numéro et dont les valeurs des arguments se correspondent. La relation de correspondance entre valeurs sémantiques se définit par les règles d'inférence suivantes :

$$\frac{S, S' \vdash v_j \sim w_j \qquad 1 \leq j \leq n}{S, S' \vdash (i, v_1; \ldots; v_n) \sim (i, w_1; \ldots; w_n)}$$

$$\frac{f \in dom(S) \qquad f \in dom(S') \qquad S, S' \vdash v_j \sim w_j \qquad 1 \leq j \leq n}{S, S' \vdash (f, v_1; \ldots; v_n) \sim (f, w_1; \ldots; w_n)}$$

Dans la suite, nous ne préciserons plus les listes de fonctions globales : on note $v \sim v'$ pour
$S, S' \vdash v \sim v'$.

Correspondance entre environnements d'évaluation Nous avons besoin d'étendre cette relation de correspondance aux environnements d'évaluation. Si x dans l'environnement Mon e est associé à la valeur v, selon que x est racine ou non du terme en cours d'évaluation, la traduction de x sera placé dans l'environnement de variables ou l'environnement de racines. Autrement dit, si $\Gamma(x) = \texttt{var}(x)$, alors x est traduit par la même variable et est stockée dans l'environnement de variable ; tandis que si $\Gamma(x) = \texttt{root}(n)$ alors x devient le n-ième emplacement de l'ensemble des racines. On définit l'accès à une valeur v dans les environnements d'évaluation Fminor $(e_{\texttt{var}}, e_{\texttt{root}})$ par rapport à une information de traduction γ comme suit :

$$(e_{\texttt{var}}, e_{\texttt{root}})(\Gamma(x)) \quad = \quad \begin{cases} e_{\texttt{var}}(x) & \text{si } \Gamma(x) = \texttt{var}(x)\,; \\ e_{\texttt{root}}(n) & \text{si } \Gamma(x) = \texttt{root}(n) \text{ et } n < |\Gamma|\,; \end{cases}$$

Seules les variables libres du terme en cours d'évaluation nous intéressent ; en effet, les autres valeurs ne seront pas consultées. Nous pouvons maintenant définir la relation de correspondance entre environnements d'évaluation :

Définition 6.5.4 (Correspondance des environnements)
Soit F un ensemble de variables, e l'environnement d'évaluation Mon,
si $\forall x \in F$, $e(x) \sim (e_{\texttt{var}}, e_{\texttt{root}})(\Gamma(x))$, alors e correspond à $(e_{\texttt{var}}, e_{\texttt{root}})$ par rapport à Γ.
On note cette correspondance : $F, \Gamma, S, S' \vdash e \approx (e_{\texttt{var}}, e_{\texttt{root}})$.

Pour plus de lisibilité, nous ne précisons plus les listes de fonctions globales. Nous noterons : $F, \Gamma \vdash e \approx (e_{\texttt{var}}, e_{\texttt{root}})$ pour $F, \Gamma, S, S' \vdash e \approx (e_{\texttt{var}}, e_{\texttt{root}})$.

Cohérence de l'environnement de traduction La correspondance entre les environnements d'évaluation est dépendante de l'environnement de traduction : c'est cet environnement qui dirige la répartition dans l'environnement Fminor des racines et qui dessine la matérialisation de la structure dédiée à l'enregistrement des racines. Il doit donc répondre à une certaine cohérence :

- un nombre de racines correct pour le modèle mémoire de Cminor ;
- chaque emplacement attribué doit être inférieur au nombre de racines courant ;

– une attribution unique de chaque emplacement.
Nous définissons ainsi le prédicat $\texttt{wf}\ \Gamma$, " l'environnement de traduction Γ est bien formé"

Définition 6.5.5 (Bonne formation de l'environnement de traduction)
Γ est bien formé si :

$|(|\Gamma|)|_p$ *: la taille* $|\Gamma|$ *est valide pour un bloc de pile Cminor,*

$\forall x$ *si* $\Gamma(x) = \texttt{root}(n)$ *alors* $n < |\Gamma|$,

$\forall x\ y$ *si* $\Gamma(x) = \texttt{root}(n)$ *et* $\Gamma(y) = \texttt{root}(n)$, *alors* $x = y$.

Enfin, si le terme en cours de traduction t est un déclencheur potentiel de GC, l'environnement de traduction Γ doit associer aux racines de t un emplacement dans l'ensemble des racines.

Définition 6.5.6 (Cohérence de l'environnement de traduction)
L'environnement de traduction Γ *est cohérent par rapport à l'ensemble* R, *noté* $\Gamma \models R$, *si* $\forall x \in R,\ \exists i,\ \Gamma(x) \neq \texttt{root}(i)$.

Caractérisation des environnements de sortie Fminor En plus de conclure à l'existence d'une valeur Fminor observationnellement équivalente, la simulation conclut l'existence des environnements de sortie d'évaluation Fminor. Ce environnement doit préserver les racines du terme en cours d'exécution, c'est-à-dire que l'environnement de racines de sortie leur associe la même valeur que l'environnement de racines initial. Soit :

Définition 6.5.7 (Préservation des racines)
L'environnement de racines $e'_{\texttt{root}}$ *préserve les racines indiquées par l'environnement de traduction* Γ *de l'environnement de racines* $e_{\texttt{root}}$ *si*
$\forall x, \Gamma(x) = \texttt{root}(n)$ *implique* $e_{\texttt{root}}(n) = e'_{\texttt{root}}(n)$.
Nous noterons la préservation des racines par : $e_{\texttt{root}} \preceq_\Gamma e'_{\texttt{root}}$.

De plus, si le terme n'est pas un déclencheur potentiel de GC, on veut que les variables libres de la traduction de t selon son environnement de traduction Γ aient les mêmes valeurs dans l'environnement de variables initial et de sortie.

Définition 6.5.8 (Préservation des variables libres)
L'environnement de variables Fminor $e'_{\texttt{var}}$ *préserve les variables libres indiquées par* Γ *de l'environnement de variables* $e_{\texttt{var}}$ *si,* $\forall x, \Gamma(x) = \texttt{var}(x)$ *implique* $e_{\texttt{var}}(x) = e'_{\texttt{var}}(x)$.
Nous noterons la préservation des variables libres par : $e_{\texttt{var}} \preceq_\Gamma e'_{\texttt{var}}$.

Simulation Nous pouvons maintenant énoncer le lemme de simulation sur l'évaluation d'un terme Mon :

Lemme 6.5.9 (Simulation)
Sous les hypothèses :

H_1 : *correspondance entre les listes de déclarations de fonctions :* $S \sim S'$,

H_2 : *bonne formation de l'environnement de traduction :* $\texttt{wf}\ \Gamma$;

H_3 : *la correspondance des environnements sur les variables libres de* t : $FV(t), \Gamma \vdash e \approx (e_{\texttt{var}}, e_{\texttt{root}})$;

H_4 : *cohérence de l'environnement de traduction :* $\texttt{triggers}\ t \Rightarrow \Gamma \models R(t)$.

Si

$$S, e \vdash t \Rightarrow v,$$

alors il existe v', $e'_{\texttt{var}}$ *et* $e'_{\texttt{root}}$ *tels que*

$$S', e_{\texttt{var}}, e_{\texttt{root}} \vdash [\![t]\!]_\Gamma \Rightarrow v', e'_{\texttt{var}}, e'_{\texttt{root}}.$$

De plus,

C_1 : *les valeurs sémantique se correspondent :* $v \sim v'$;

C_2 : *les racines sont préservées :* $e_{\texttt{root}} \preceq_\Gamma e'_{\texttt{root}}$;

C_3 : *si* t *n'est pas un déclencheur de GC, alors les variables locales sont préservées :* $\neg\texttt{trigger}\ t$ *implique* $e_{\texttt{var}} \preceq_\Gamma e'_{\texttt{var}}$.

La preuve se fait par induction sur une évaluation Fminor. Les points clés de la preuve sont les constructions des invariants au cours des évaluations, décrites ci-dessous.

Construction des invariants lors de l'évaluation d'une liaison locale

Lors de l'évaluation d'une liaison locale $\texttt{let}\ x = t_1\ \texttt{in}\ t_2$, nous avons en hypothèses :

H_1 : $\texttt{wf}\ \Gamma$;

H_2 : $FV(\texttt{let}\ x = t_1\ \texttt{in}\ t_2), \Gamma \vdash e \approx (e_{\texttt{var}}, e_{\texttt{root}})$;

H_3 : $\texttt{triggers}\ (\texttt{let}\ x = t_1\ \texttt{in}\ t_2) \Rightarrow \Gamma \models R(\texttt{let}\ x = t_1\ \texttt{in}\ t_2)$.

Invariants de l'évaluation du sous-terme gauche Pour utiliser l'hypothèse d'induction sur l'évaluation de t_1, il nous faut restreindre ces hypothèses aux variables libres de t_1 et les racines de t_1.

Lemme 6.5.10 *Sous* H_2, *nous avons* $FV(t_1), \Gamma \vdash e \approx (e_{\texttt{var}}, e_{\texttt{root}})$.

Cette preuve utilise le fait que l'ensemble des variables libres de t_1 est un sous ensemble de l'ensemble des variables libres de toute la liaison locale.

Lemme 6.5.11 *Sous* H_3, *nous avons :* $\texttt{triggers}\ (t_1) \Rightarrow \Gamma \models R(t_1)$.

En effet, si le sous-terme gauche est un déclencheur potentiel de GC, alors toute la liaison est un déclencheur potentiel de GC. Γ est donc cohérent par rapport à l'ensemble des racines de la liaison locale. Or, l'ensemble des racines de t_1 est un sous ensemble des racines de toute la liaison, puisque t_1 peut déclencher un GC.

Nous pouvons donc appliquer la simulation à l'évaluation de t_1 et nous obtenons l'existence d'une valeur v'_1, d'un environnement de variables Fminor $e'_{\texttt{var}}$ et d'un environnement de racine $e'_{\texttt{root}}$ tels que :
C_1 : $v_1 \sim v'_1$,
C_2 : $e'_{\texttt{root}}$ tel que $e_{\texttt{root}} \preceq e'_{\texttt{root}}$,
C_3 : $\neg\texttt{triggers}\ t_1 \Rightarrow e_{\texttt{var}} \preceq_\Gamma e'_{\texttt{var}}$.

Invariants de l'évaluation du sous-terme droit Pour pouvoir appliquer la simulation sur l'évaluation de t_2, nous devons construire les invariants pour les environnements intermédiaires $e[x \leftarrow n]$, Γ' où $\texttt{add}\ x\ R(t_2)\ \Gamma = (p, \Gamma')$, et en fonction de p, $(e'_{\texttt{var}}[x \leftarrow v'], e'_{\texttt{root}})$ si $x \notin R(t_2)$ ou $(e'_{\texttt{var}}, e'_{\texttt{root}}[n \leftarrow v'])$ si $x \in R(t_2)$ et $p = \texttt{root}(n)$.

Lemme 6.5.12 *Si* $\texttt{wf}\ \Gamma$ *et* $\texttt{add}\ x\ R(t_2)\ \Gamma = (p, \Gamma')$, *alors* $\texttt{wf}\ \Gamma'$.

En effet, si x n'est pas une racine de t_2 alors la taille de l'ensemble des racines ne change pas et les informations concernant les racines n'ont pas été modifiées. Si x est une racine de t_2 alors la fonction `add` augmente la taille de l'ensemble de 1, et teste si cette taille est correcte pour un futur bloc de pile Cminor. Si ce n'était pas le cas `add` aurait échoué. De même, l'emplacement associé à x est $|\Gamma|$ trivialement inférieur à $|\Gamma'|$. Enfin, aucune autre variable n'est associée au même emplacement, puisque $\texttt{wf}\ \Gamma$.

De même,

Lemme 6.5.13 *Sous* H_3, *si* $\texttt{add}\ x\ R(t_2)\ \Gamma = (p, \Gamma')$ *alors* $\texttt{triggers}\ t_2 \Rightarrow \Gamma' \models R(t_2)$.

La preuve repose sur le fait que si $\texttt{triggers}\ t_2$, alors toute la liaison locale est un déclencheur potentiel d'un GC et donc $\Gamma \models R(t_2)$, étant donné que $R(t_2) \subseteq R(\texttt{let}\ x = t_1\ \texttt{in}\ t_2)$.

Enfin, il faut construire la correspondance des environnements. Rappelons que :
- si x est une racine de t_2, alors l'environnement de traduction actualisé Γ' est : $\{\Gamma[x \leftarrow \texttt{root}(|\Gamma|)], |\Gamma| + 1\}$,
- sinon, $\Gamma' = \Gamma[x \leftarrow \texttt{var}(x)]$.

Lemme 6.5.14
Sous les invariants initiaux H_1, H_2 *et* H_3 *et les conclusions de la première simulation :* C_1, C_2 *et* C_3, *si* $\Gamma(x) = \bot$,
- *soit* $x \in R(t_2)$ *alors* $FV(t_2), \Gamma' \vdash e[x \leftarrow v_1] \approx (e'_{\texttt{var}}, e'_{\texttt{root}}[|\Gamma| \leftarrow v'_1])$,
- *soit* $x \notin R(t_2)$ *alors* $FV(t_2), \Gamma' \vdash e[x \leftarrow v_1] \approx (e'_{\texttt{var}}[x \leftarrow v'_1], e'_{\texttt{root}})$.

Nous sommes alors en mesure d'utiliser le lemme de simulation sur l'évaluation du sous-terme droit et de conclure la simulation de l'évaluation de la liaison locale.

Simulation de l'appel de fonction

Lors de l'évaluation de l'appel de fonction $a\ [a_1; \ldots; a_n]$ en Mon, nous construisons l'environnement d'évaluation du corps de la fonction mise en jeu, à partir de l'environnement vide, en ajoutant une liaison pour chaque paramètre formel à la valeur d'évaluation de l'argument correspondant : $\varepsilon[x_0 \leftarrow v_0] \ldots [x_n \leftarrow x_n]$ si $x_1; \ldots; x_n$ sont les paramètres formels et $v_1; \ldots; v_n$ les valeurs des arguments.

En Fminor, on procède de la même manière. En effet, si d est la fonction mise en jeu, avec $d.\texttt{params} = x_1; \ldots; x_n$, alors $d.\texttt{body}$ s'évalue dans l'environnement de racines vide et l'environnement de variables $\varepsilon[x_0 \leftarrow w_0] \ldots [x_n \leftarrow w_n]$.

D'ailleurs, lors de la traduction d'une fonction d, le corps est traduit dans un environnement Γ renseignant les paramètres formels selon qu'ils soient racines ou pas et on calcule une liste $(r_0, 0), \ldots, (r_k, k)$ telle que $r_i \in d.\texttt{params}$ et $r_i \in R(d.\texttt{body})$. Le corps de fonction Fminor généré est alors : $\texttt{let root } 0 = r_0 \texttt{ in } \ldots \texttt{let root } k = r_k \texttt{ in } [\![d.\texttt{body}]\!]_\Gamma$,
noté $\texttt{let root seq } (r_0, 0), \ldots, (r_k, k) \texttt{ in } [\![d.\texttt{body}]\!]_\Gamma$.

Or, lors de l'application de la simulation à l'évaluation de l'application, on a l'hypothèse d'induction sur l'évaluation du corps de fonction. En utilisant un lemme similaire à celui énoncé plus haut pour la liaison locale, on obtient la préservation des invariants pour l'ajout d'un paramètre formel selon qu'il soit racine ou pas du corps de fonction. Par séquentialisation, nous obtenons la construction des invariants pour l'évaluation du corps de fonction. Entre autres nous obtenons :
$S,\ e_{\texttt{var}},\ e_{\texttt{root}} \vdash d.\texttt{body} \Rightarrow v', e'_{\texttt{var}}, e'_{\texttt{root}}$. Cependant, le corps de la fonction générée est
$\texttt{let root seq}(r_0, 0), \ldots, (r_k, k) \texttt{ in } [\![d.\texttt{body}]\!]_\Gamma$. Nous utilisons le résultat intermédiaire suivant pour obtenir cette évaluation :

Lemme 6.5.15 *Supposons*

$$S,\ e_{\texttt{var}},\ e_{\texttt{root}} \vdash t \Rightarrow v, e'_{\texttt{var}}, e'_{\texttt{root}}$$

et

$$e_{\texttt{root}} = s[0 \leftarrow e(r_0)] \ldots [k \leftarrow e(r_k)]$$

alors

$$S,\ e_{\texttt{var}},\ s \vdash \texttt{let root seq}(r_0, 0), \ldots, (r_k, k) \texttt{ in } t \Rightarrow v, e'_{\texttt{var}}, e'_{\texttt{root}}.$$

6.6 Conclusion et perspectives

Nous avons explicité l'interaction avec un gestionnaire automatique de mémoire au travers de deux langages intermédiaires fonctionnels. Nous avons choisi une transmission des racines à la Henderson, il s'agit de transmettre les racines au GC via une structure de données dédiée. L'interaction avec le gestionnaire de mémoire réside, pour le compilateur en le calcul des racines et leur transmission via la structure de données dédiée. Le premier langage intermédiaire est la forme monadique intermédiaire. Dans cette forme tous les calculs sont élémentarisés en calculs intermédiaires et tout calcul intermédiaire est nommé. Cette forme est propice au calcul des racines, toutes les racines, même celles correspondant à des calculs intermédiaires sont détectables. Le second langage intermédiaire est Fminor, il explicite la structure de données dédiée à l'enregistrement des racines. Pour cela, la syntaxe manipule concrètement l'ensemble des racines : on peut invoquer un élément de l'ensemble ou bien lier localement un élément (à la manière d'une variable). La sémantique de Fminor décrit le comportement des termes du langage et spécifie cette interaction. Pour cela, les règles d'évaluation d'un terme potentiellement déclencheur de GC, les structures de données et les applications, ont un traitement des environnements de sortie caractéristiques, modélisant le comportement du GC : elles vident l'environnement des variables et retournent l'environnement de racines initial, ainsi si l'évaluation suivante réussit, cela assure que le calcul des racines est correcte.

Une optimisation envisageable

Notre mode d'enregistrement des racines se fait au niveau de la liaison d'une variable et une variable considérée comme une racine reste sur la structure dédiée jusqu'à la fin de sa portée lexicale. Une optimisation possible de nos travaux serait qu'une racine reste sur la structure de données dédiée jusqu'à la fin de sa *durée de vie* (dernière occurrence). Une méthode serait le *black-holing*.

Considérons l'exemple suivant :

$$\texttt{let}\ x = t_1\ \texttt{in}\ \texttt{let}\ y = f\ [x, z]\ \texttt{in}\ t.$$

S'il n'y a aucune occurrence de x dans t, x reste dans l'ensemble des racines jusqu'à la fin de sa portée lexicale inutilement. Pour chaque déclenchement potentiel de GC contenu dans t, cela fait une racine supplémentaire ainsi que tous ses accessibles conservées inutilement. Après la dernière occurrence de x, on pourrait plutôt la remplacer dans la structure de donnée dédiée à l'enregistrement des racines par un pointeur sur `NULL` (d'où la métaphore du trou noir). Cela implique que le GC n'aurait plus à considérer x comme une racine et surtout ne conserverait plus inutilement des objets accessibles depuis x. La méthode du "black-holing" constitue donc un raffinement de notre mode d'enregistrement : l'ensemble des racines est statiquement estimé de manière plus fine.

Notre développement est facilement adaptable à cette optimisation, il suffirait d'avoir une valeur par défaut ou bien un tag indiquant la fin de vie d'une racine. Le calcul de vie d'une variable ne serait qu'une simple analyse syntaxique sur la forme intermédiaire monadique.

7 Génération de code `Cminor`

Les langages fonctionnels sont des langages de haut niveau où l'esprit d'abstraction prime. Ces langages sont particulièrement adaptés aux formalisations et aux raisonnements.

Les langages impératifs de bas niveau, comme `C` et plus particulièrement Cminor (voir la section 2.3), sont des langages plus concrets et proches de l'utilisation de la machine. Ils permettent de manipuler plus explicitement les composants de la machine, tout en étant plus abstraits que les langages assembleurs.

Les structures de données complexes dans les langages impératifs sont stockées en mémoire. C'est le cas en Cminor (voir la section 2.3.2). La mémoire évolue le long de l'évaluation d'un programme. On peut créer des éléments (par allocation), en ajouter ou modifier (par écriture), ou bien encore en effacer (par libération). La mémoire est structurée en plusieurs zones. Chacune de ces zones se distingue par le genre de données qu'on y stocke :

- Les données globales sont stockées dans une partie allouée une fois pour toutes. Elle ne varie par de taille durant l'exécution d'un programme. Parmi ces données, on trouve le code des fonctions.
- Le tas est une partie de la mémoire dédiée aux structures de données du programme. C'est une zone que le GC peut manipuler.
- La pile, en Cminor, est telle que chaque appel de fonction induit l'allocation d'un bloc de pile. De plus, chaque déclaration de fonction indique la taille de pile nécessaire. Enfin, l'allocation d'un bloc de pile se fait dynamiquement. La pile courante est manipulable directement dans la syntaxe de Cminor via l'opérateur `addrstack`. Toutefois il ne faut pas confondre les blocs de pile Cminor avec la pile d'appel.

La capacité à manipuler l'état mémoire forme la principale distinction entre un langage fonctionnel pur de haut niveau et un langage impératif de bas niveau. La compilation de εML en Cminor se présente donc comme la concrétisation d'un langage mathématique d'abstraction vers un langage presque machine manipulant la représentation d'objets en mémoire. Notre compilation est guidée par trois problématiques liées à cette concrétisation :

- la globalisation des fonctions,
- la représentation uniforme des données,
- l'interaction avec le gestionnaire de mémoire.

Tout le long de la chaîne de compilation décrite dans les chapitres précédents, nous avons considéré ces problématiques. La numérotation des constructeurs (décrite dans la section 2.4.4), nous donne une représentation plus efficace des constructeurs permettant l'utilisation de table de sauts et de recherche dichotomique. La décurryfication (chapitre 3) nous permet d'économiser le nombre de fermetures et donc d'allocations dans le tas. L'explicitation des fermetures (chapitre 5) nous a rapproché d'une représentation uniforme des données en créant des fermetures minimales. De plus, les fonctions ainsi fermées ont pu être

globalisées. Le calcul des racines et l'interaction avec le gestionnaire de mémoire via la pile ont été explicités via les passes décrites au chapitre 6.

Le présent chapitre est la dernière étape de ces efforts : la génération du code Cminor issu de ce processus de compilation.

Dans ce chapitre, nous commençons par décrire la représentation uniformes des données. Puis nous décrivons concrètement l'interaction avec le gestionnaire de mémoire et l'encodage de cette interaction dans le code produit. Nous présentons alors la transformation de programme Fminor (voir la section 6.4.5) en programme Cminor. Avant d'exhiber la preuve de préservation sémantique de cette transformation, nous définissons une axiomatisation et spécification du comportement du gestionnaire de mémoire.

7.1 Représentation uniforme des données

La représentation uniforme des données consiste à représenter les structures de données fonctionnelles (les fermetures et les constructeurs appliqués) par des pointeurs Cminor. Plus précisément, il s'agit de représenter les structures de données abstraites de miniML au travers d'objets stockés en mémoire et accessibles via des pointeurs Cminor. Ces pointeurs référencent des blocs du tas. Pour chaque structure de données, on alloue un bloc dans le tas et on y place les éléments la constituant :

Représentation d'un constructeur appliqué : Un constructeur appliqué $C(v_1; \ldots; v_n)$ sera représenté par un pointeur $(b, 0)$. b, représenté sur la figure 7.1.1, est un bloc du tas de $n+1$ mots mémoire. Le premier mot, $b[0]$, contient l'entier C qui identifie le constructeur C. Les n autres mots, $b[4], \ldots, b[4 * n]$, contiennent des pointeurs vers les représentations des $v_1; \ldots; v_n$.

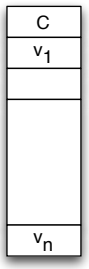


FIG. 7.1.1 – Bloc mémoire contenant un constructeur appliqué

Il existe d'autres manières de représenter un constructeur. Par exemple, le compilateur OCaml ne procède pas à une représentation uniforme des données : un constructeur constant, c'est-à-dire d'arité nulle, est représenté par l'entier correspondant à son numéro. Dans notre représentation, un constructeur constant est représenté par un pointeur vers un bloc d'un mot, ce mot contenant l'entier correspondant au numéro du constructeur.

Représentation d'une fermeture : Dans la littérature, il existe diverses manières de représenter une fermeture. Nous avons choisi de les représenter en un bloc, comme dans les compilateurs OCaml ou bien encore Standard ML of New Jersey. Dans notre code Cminor généré, une fermeture $(f, v_1; \ldots; v_n)$ sera représentée par un pointeur $(b, 0)$. b, représenté sur la figure 7.1.2 est un bloc du tas de $n+1$ mots mémoire. Le premier mot, $b[0]$, contient un pointeur vers le code de la fonction f. Les n autres mots, $b[4], \ldots, b[4*n]$, contiennent des pointeurs vers les représentations des $v_1; \ldots; v_n$.

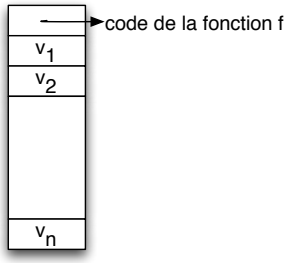


FIG. 7.1.2 – Bloc mémoire contenant une fermeture

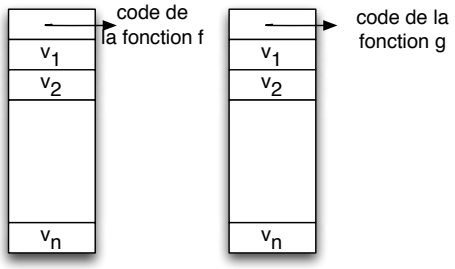


FIG. 7.1.3 – Représentation d'une fermeture en un bloc

Une autre représentation possible aurait été la représentation en deux blocs : un bloc contenant un pointeur vers le code de la fonction et un pointeur vers le second bloc contenant l'environnement. Cette représentation permet un partage d'environnement entre plusieurs fermetures. La figure 7.1.4 montre la représentation de deux fermetures de fonctions ayant le même environnement, dans la représentation en un bloc et la représentation en deux blocs. Dans une représentation en un bloc, l'environnement ne peut être mis en commun et donc, chaque fermeture a une copie de cet environnement comme le montre la figure 7.1.3.

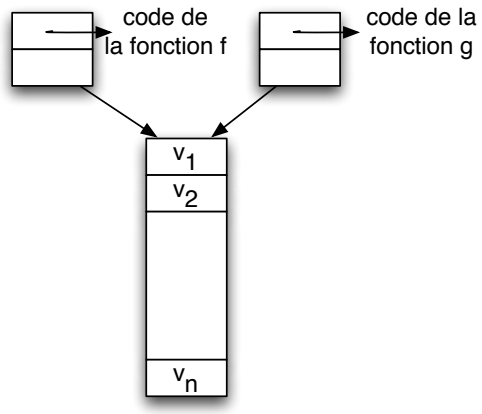


FIG. 7.1.4 – Représentation d'une fermeture en deux blocs

Enfin, comme dans la CAM [23], on pourrait représenter une fermeture par une liste chaînée dont la tête contient un pointeur vers le code de la fonction et un pointeur vers le premier élément de l'environnement. Les éléments de l'environnement sont chaînés les uns aux autres. L'idée est de partager des portions de l'environnement entre plusieurs fermetures comme le décrit la figure 7.1.5.

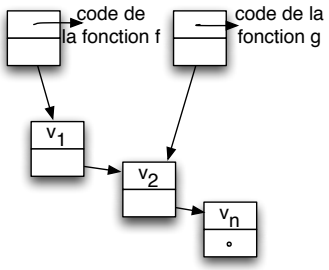


FIG. 7.1.5 – Représentation d'une fermeture en liste chaînée

Le gain engendré par le partage reste faible vis-à-vis du surcoût induit sur les accès aux valeurs $v_1; \ldots; v_n$. C'est pourquoi nos fermetures sont représentées en un bloc.

Pour chaque structure de données, on allouera donc un bloc dans le tas que l'on rempliera par une suite d'écritures. Le langage source étant purement fonctionnel, un mot mémoire initialisé par une écriture ne changera jamais par la suite :

– on n'écrit à un emplacement sur un bloc de tas qu'au moment de construire une

nouvelle structure de données, c'est-à-dire après l'allocation du bloc,
– on ne réécrit jamais sur un tel emplacement.

7.2 Interaction avec le gestionnaire de mémoire

Comme décrit dans le chapitre 6, nous avons choisi d'interagir avec un gestionnaire de mémoire automatique à GC. Cet interaction réside d'une part, dans le calcul des racines et leur transmission. Nous utilisons les blocs de pile Cminor comme vecteur d'enregistrement. De plus, le chaînage des blocs de pile consécutifs peut simuler la pile d'appels. Les racines de chaque fonction sont enregistrées à leurs déclarations sur la pile courante Cminor de la fonction. Nous avions commencé la mise en place de cette transmission au chapitre 6.

D'autre part, l'allocation de mémoire pour la représentation des structures de données se fait avecla fonction d'allocation du gestionnaire de mémoire `"alloc_block"`.

7.2.1 Allocation dans le tas

Les structures de données nécessitent des allocations dans le tas. Ces allocations se feront via la fonction `"alloc_block"` du gestionnaire de mémoire. Cette fonction Cminor prend en paramètre la taille du bloc à allouer et retourne un pointeur sur bloc fraîchement alloué. `"alloc_block"` est l'interface avec le gestionnaire de mémoire dont nous disposons. `"alloc_block"` est susceptible de déclencher un GC si jamais il n'y a pas assez d'espace disponible dans le tas. Nous détaillerons plus loin la spécification de l'effet d'un GC (voir la section 7.4) sur le modèle mémoire Cminor (décrit en section 2.3.2).

7.2.2 Interaction avec le GC, transmission des racines

Les principaux acteurs de l'interaction entre un programme Cminor généré par compilation d'un programme Fminor et le GC du gestionnaire de mémoire sont les blocs de pile Cminor. Pour chaque appel de fonction, Cminor alloue dynamiquement un bloc de pile. La taille du bloc de pile alloué pour un appel à la fonction f est de f.`stacksize`.

La transmission des racines via les blocs de pile Cminor a commencé à être mise en place au travers du langage Fminor. Comme nous l'avons vu au chapitre 6, si une variable est racine, elle est enregistrée à sa déclaration dans l'ensemble des racines. Cet ensemble de racines propre à un appel de fonction à un moment donné devient en Cminor une portion de la pile Cminor correspondant au même appel.

Toutefois, si on considère que dans le corps de la fonction courante f_0, avec pour bloc de pile Cminor courante sp_0, nous exécutons l'appel à la fonction f_1, Cminor alloue un nouveau bloc de pile sp_1. Or, parmi les racines d'un GC qui se déclencherait durant l'exécution du corps de f_1, certaines sont sur le bloc de pile Cminor de l'appelant sp_0. Plus généralement, un GC dans un appel de fonction doit avoir accès aux racines stockées sur les blocs de pile Cminor des fonctions appelantes successives. Pour cela, on chaîne ces blocs de pile d'appelant à appelé, un peu comme une simulation de la pile d'appel. La figure 7.2.1 schématise le chaînage de trois blocs de pile. Le bloc `NULL` désigne le premier bloc de pile alloué à l'exécution d'un programme, le bloc sp_0 est le bloc de pile d'une fonction et le bloc sp_1 correspond au bloc de pile Cminor courant d'un appel de fonction contenue dans la première fonction. Le premier emplacement sur un bloc de pile Cminor, dans notre trans-

formation, est dédié au chaînage des blocs de pile Cminor, il pointe sur le bloc de pile de la fonction appelante. Le deuxième emplacement indique le nombre de racines présentes sur le bloc lui-même.

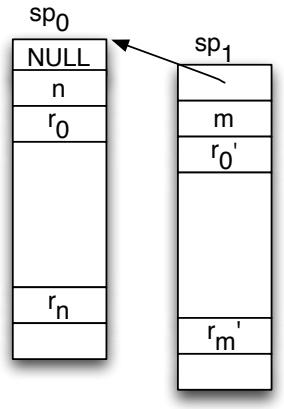


FIG. 7.2.1 – Chaînage des blocs de pile

Pour implémenter ce chaînage des blocs de pile Cminor reconstituant une pseudo-pile d'appel, nous ajoutons à chaque déclaration de fonction un paramètre supplémentaire `Roots` :

- Chaque fonction transmet un pointeur sur son bloc de pile Cminor à chaque fonction qu'elle appelle. Pour cela les appels de fonction ont, en tête des arguments, un argument supplémentaire qui est systématiquement : `addrstack`.
- Chaque fonction effectue le chaînage entre son bloc de pile Cminor et celui de la fonction appelante. Ainsi d'appelant en appelé la pseudo-pile d'appels se trame. Pour cela, le début de chaque corps de fonction commence par l'écriture dans le premier emplacement de son bloc de pile Cminor courant du pointeur désigné par le paramètre supplémentaire `Roots` : `addrstack` ← `Roots`.

Un bloc de pile Cminor issu de notre génération de code Cminor se constitue donc de la manière suivante :

Chaînage de la pseudo pile d'appel : Dans son premier emplacement, un bloc de pile stocke le pointeur vers le bloc de pile précédent.

Le nombre de racines avant un site potentiel de GC : En Fminor, les déclencheurs potentiels directes de GC portent syntaxiquement le nombre de racines stockées sur la pile. Cette information est nécessaire au GC. Avant chaque appel à l'allocateur du gestionnaire de mémoire, on stocke sur le deuxième emplacement de la pile courante ce nombre de racines. Ce nombre de racines est portée syntaxiquement par les constructeurs de données et les appels de fonction Fminor. Chaque construction de données et chaque appel de fonction dans notre transformation commence par l'écriture de ce nombre z, plus exactement la quantité de mémoire nécessaire au stockage de ce

nombre de racines en mémoire dans le deuxième emplacement du bloc de pile Cminor courant : $\texttt{addrstack}[4] \leftarrow (z \times 4)$.

Stockage des racines : La liaison locale dans l'ensemble des racines à un emplacement n de Fminor devient une écriture à l'emplacement $\texttt{addrstack}[(2+n) \times 4]$ (les deux premiers mot ayant l'usage décrit juste au-dessus). Enfin, la propriété de bon bornement (voir la section 6.5.2) des termes Fminor construits par compilation, nous garantit que les racines sont écrites dans l'ordre de leur déclaration et de manière contiguë.

7.3 Génération de code Cminor

Nous détaillons ici les mécanismes d'encodage de Fminor en Cminor en commençant par définir une première traduction.

Nous avons vu comment encoder la représentation des structures de données Fminor en Cminor. Nous détaillons maintenant l'ensemble de la génération de code Cminor.

7.3.1 Traduction des atomes

Les atomes de Fminor deviennent des expressions pures de Cminor. Il s'agit de consultations d'environnement. Ainsi, une variable x est traduite par la même variable. Un accès à un emplacement sur la pile $\texttt{root}(n)$ devient une lecture à l'emplacement $(2+n)*4$ sur le bloc de pile courant : $\texttt{addrstack}[(2+n)*4]$. Enfin, l'accès à un élément de fermeture $\texttt{field}\ n\ a$ devient la lecture à l'adresse $[\![a]\!][n*4]$, comme expliqué plus haut. La traduction des atomes Fminor se résume comme suit :

$$\begin{aligned} [\![\texttt{var}(x)]\!] &= x \\ [\![\texttt{root}(n)]\!] &= \texttt{addrstack}[(2+n)*4] \\ [\![\texttt{field}\ n\ a]\!] &= [\![a]\!][n*4] \end{aligned}$$

7.3.2 Encodage du filtrage

Le filtrage est une structure syntaxique généralement associée aux langages fonctionnels permettant d'exploiter les types concrets. Selon un sujet a, qui doit s'évaluer en un constructeur d'un type concret τ, un filtrage propose de sélectionner la suite du calcul selon la valeur de a. Pour cela, pour chaque constructeur du type τ, le filtrage propose une clause. Selon la valeur de a, une clause du filtrage sera choisie comme suite du calcul.

En Cminor, il n'y a ni type concret ni filtrage. Cependant, Cminor est muni de la structure de sélection d'instruction par cas, l'instruction `Sswitch`. Cette instruction discrimine sur une expression a, chaque valeur possible de a est associée à une sortie de bloc Cminor $exit(n)$ (sortie du $n$ième bloc englobant).

Afin d'expliquer notre encodage du filtrage en Cminor, nous commençons par raisonner comme si nous disposions d'une instruction `switch` à la `C`.

Depuis la numérotation des constructeurs (voir la section 2.4.4), les constructeurs sont représentés par des entiers. Un constructeur appliqué devient en Cminor un bloc de tas, tel que décrit sur la figure 7.1.1. La construction d'un constructeur produit un pointeur

sur un tel bloc et l'identifiant (son numéro) du constructeur est écrit dans son premier emplacement.

Considérons le filtrage Fminor suivant :

$$\begin{array}{l}\texttt{match } a \texttt{ with} \\ \qquad\qquad |\vec{p}_0 -> t_0 \\ \qquad\qquad \ldots \\ \qquad\qquad |\vec{p}_n -> t_n\end{array}$$

Un encodage possible avec un switch à la C serait :

$$\begin{array}{l}\texttt{switch } [\![a]\!][0] \\ \qquad \{ \quad case\ 0 : \quad [\![t_0]\!](\vec{p}_0, [\![a]\!]) \\ \qquad\qquad \ldots \\ \qquad\quad case\ n : \quad [\![t_n]\!](\vec{p}_n, [\![a]\!]) \\ \qquad \}\end{array}$$

Nous avons noté $[\![t_i]\!](\vec{p}_i, [\![a]\!])$ la traduction de la $i$ème clause. Il est nécessaire de prendre en paramètre le discriminé $[\![a]\!]$. En effet, les paramètres d'une clause deviennent des accès aux arguments du discriminé.

Le filtrage étant ordonné la $i$ème clause concerne le constructeur de numéro $i-1$. La $i$ème clause devient donc le cas $i-1$. L'instruction associée à ce cas commence par traiter la traduction des paramètres. La traduction des clauses de filtrage sera détaillée dans la section suivante (voir la section 7.3.3). Après le traitement des paramètres, vient la traduction du corps de la clause.

En combinant les blocs et le `Sswitch` de Cminor, on peut encoder en Cminor l'instruction `switch` de `C`.

```
switch e
{
 case a : f;
 case b : g;
 case c : h;
 default : i;
}
```

devient en Cminor :

```
{
 {
  {
   {
    Sswitch e'
     {case a :exit 0; case b :exit 1;case c :exit 2;default :exit 3}
     f';exit 3; }
```

```
   g';exit 2; }
  h';exit 1; }
 i';exit 0; }
```

Nous avons noté `i'` la traduction en Cminor de l'expression (ou de l'instruction) `i` du langage `C`.

Nous notons `mkMatch` la fonction qui prend en paramètre une expression Cminor e, qui s'évalue en un entier et $i_0; \ldots; i_n$ une liste d'instructions, la fonction qui construit un encodage de filtrage à la `C` en Cminor :

$$\begin{aligned}\texttt{mkMatch}\ e\ (i_0; \ldots; i_n) \quad = \quad & \\ & \{\ldots\{\texttt{Sswitch}\ e\ [(0,0); \ldots; (n,n)]\ n+1] \\ & \qquad ;\ i_0\ ;\ \texttt{exit}\ n; \}\ldots\}\ ;\ i_n\ ;\ \texttt{exit}\ 0\}\end{aligned}$$

7.3.3 Traduction des termes

On note $[\![t]\!]_x$ la génération de code Cminor correspondant à la liaison de t à la variable x. L'expression Cminor encodant le terme Fminor t est affectée à la variable x. L'algorithme est présenté sur la figure 7.3.1.

Les atomes sont encodés comme décrit plus haut (voir la section 7.3.1), cet encodage est alors affecté à x.

Considérons une liaison locale de variable `let var` $y = t_1$ `in` t_2 traduite pour être affectée à une variable x. On commence par traduire t_1 comme terme à affecter à la variable y, puis on traduit t_2 à affecter à x.

Pour une liaison locale à un emplacement dans l'ensemble des racines `let root`n = t_1 `in` t_2, on traduit t_1 à affecter à `tmp_st`, il s'agit d'une variable spéciale que l'on a réservée depuis la génération de nom frais au chapitre 5. On effectue alors l'écriture dans le bloc de pile Cminor courant, sans oublier de décaler de deux emplacements. Enfin, on traduit t_2 à affecter à x.

Pour la filtrage, nous utilisons l'encodage présenté à la section précédente. L'affectation est repoussée dans la traduction des corps de clause de filtrage. La traduction d'une clause est paramétrée par l'expression correspondant au sujet e. Avant de traduire le corps de la clause il faut traiter les paramètres de la clause, de telle manière à ce que p_i deviennent $e[i+1]$. Si $p_i = \texttt{var}(y)$, alors il faut affecter $e[i+1]$ à la variable y. Si $p_i = \texttt{root}(k)$, il faut alors écrire $e[i+1]$ dans le $k+2$ème mot de la pile Cminor courante.

Intéressons nous maintenant à la traduction des termes potentiellement déclencheurs de GC.

Commençons par les constructeurs de données. Il faut commencer par sauver le nombre de racines présentes sur le bloc de pile Cminor courant, c'est pourquoi l'on écrit z dans le deuxième emplacement du bloc de pile. On peut alors faire appel à la fonction d'allocation du gestionnaire de mémoire avec un mot mémoire de plus que le nombre de données portées syntaxiquement $(n+1)$, et affecter le pointeur qu'il retourne à la variable x (celle de la traduction). S'il s'agit d'un constructeur, on écrit dans le premier emplacement du bloc pointé par x le numéro du constructeur. S'il s'agit d'une fermeture, on y écrit la valeur d'un pointeur sur le code de cette fonction. Puis dans les n emplacements suivants on écrit la traduction des données.

$$
\begin{array}{rcl}
[\![a]\!]_x & = & x \leftarrow [\![a]\!] \\
[\![\texttt{let var } y \ = \ t_1 \texttt{ in } t_2]\!]_x & = & [\![t_1]\!]_y \ ; \ [\![t_2]\!]_x \\
[\![\texttt{let root } n \ = \ t_1 \texttt{ in } t_2]\!]_x & = & [\![t_1]\!]_{\texttt{tmp\_st}} \ ; \ \texttt{addrstack}[(2+n)*4] \leftarrow \texttt{tmp\_st} \ ; \ [\![t_2]\!]_x \\
[\![\texttt{match } a \texttt{ with } \pi_0; \ldots; \pi_n]\!]_x & = & \texttt{mkMatch } [\![a]\!] \ ([\![\pi_0]\!]_x \ [\![a]\!], \ldots, [\![\pi_n]\!]_x \ [\![a]\!]) \\
[\![f \ (a_1; \ldots; a_n)^z]\!]_x & = & \texttt{addrstack}[4] \leftarrow z; \\
 & & x \leftarrow \text{"}\texttt{alloc\_block}\text{"}(\texttt{addrstack}, (n+1)); \\
 & & x[0] \leftarrow \texttt{addrsymbol}(f); \\
 & & x[4] \leftarrow [\![a_1]\!] \ ; \ \ldots; \ x[4 \times n] \leftarrow [\![a_n]\!]; \\
[\![C \ (a_1; \ldots; a_n)^z]\!]_x & = & \texttt{addrstack}[4] \leftarrow z; \\
 & & x \leftarrow \text{"}\texttt{alloc\_block}\text{"}(\texttt{addrstack}, (n+1)); \\
 & & x[0] \leftarrow C; \\
 & & x[4] \leftarrow [\![a_1]\!] \ ; \ \ldots; \ x[4 \times n] \leftarrow [\![a_n]\!]; \\
[\![a \ (a_0; \ldots; a_n)^z]\!]_x & = & \texttt{addrstack}[4] \leftarrow z \ ; \\
 & & x \leftarrow (\sigma(n))[\![a]\!] \ (\texttt{addrstack}, [\![a]\!], [\![a_0]\!], \ldots [\![a_n]\!]) \\
 & & \\
[\![p_1; \ldots; p_n \rightarrow t]\!]_x \ e & = & \texttt{res} \leftarrow \ e \ ; \ [\![p_1]\!] \leftarrow \texttt{res}[1]; \ldots; [\![p_n]\!] \leftarrow \texttt{res}[n]; \ [\![t]\!]_x
\end{array}
$$

FIG. 7.3.1 – Traduction des termes Fminor

Intéressons nous maintenant à la traduction d'un appel de fonction $a \ (a_0; \ldots; a_n)^z$. Avant toute chose, on réactualise le nombre de racines stockées sur la pile courante actuelle. Il s'agit de z que l'on écrit sur le deuxième emplacement de la pile. Les appels de fonction Cminor sont accompagnés d'une signature. La représentation uniforme des données impose la représentation de chaque donnée par un pointeur c'est-à-dire par le type `int` dans les signatures. On fabrique alors une signature indiquant $n+3$ arguments de type `int` et retournant un `int`, où $n+1$ est le nombre d'arguments de l'appel. On fabrique alors un appel Cminor de signature $\underbrace{\texttt{int} \rightarrow \ldots \texttt{int}}_{n+3} \rightarrow \texttt{int}$, cette signature est notée $\sigma(n)$. L'appelant est la traduction de a. Afin de chaîner les blocs de pile Cminor pour former la pseudo-pile d'appels, l'argument `addrstack` est ajouté. De même, il faut ajouter explicitement la fermeture elle-même $[\![a]\!]$. Enfin les autres paramètres sont traduits : l'application peut alors être affectée à la variable x.

7.3.4 Optimisation des appels en position terminale

Moins de blocs de pile : les appels en position terminale

En Cminor, les appels en *position terminale* sont distincts des autres appels de fonction. Leurs exécutions diffèrent, notamment au niveau du bloc de pile Cminor de la fonction appelante. Le bloc de pile de la fonction appelante d'appel en position terminale est inutile à la suite de l'exécution. En effet, cet appelant ne fait que retourner le résultat de l'appel. L'optimisation de la compilation des appels en position terminale est courante dans la compilation de langages fonctionnels. Elle permet de réduire la taille de pile nécessaire au calcul. De plus, par rapport à notre utilisation des blocs de pile Cminor, cette optimisation permet de réduire la sur-approximation des racines. En effet, si l'appelant ne fait que retourner le résultat de l'appel en position terminale, il est évident que les racines présents sur son bloc de pile Cminor ne sont pas des racines pour la suite de l'évaluation.

Dans un terme Fminor il est facile de repérer un appel en position terminale : il est de la forme $E[a\ (a_1; \ldots; a_n)]$ où E est défini comme suit :

$$E ::= [] \mid \texttt{let var}\ x = t_1\ \texttt{in}\ E \mid \texttt{let root}\ n = t_1\ \texttt{in}\ E \mid \texttt{match}\ a\ \texttt{with}\ \ldots\ \vec{p} \to E \ldots.$$

Exemple : Considérons la fonction `rev_append` en syntaxe concrète (à la Caml) :

```
let rec rev_append l1 l2 =
    match l1 with
    [] -> l2
  | a : :l -> rev_append l (a : :l2)
```

L'appel récursif à `rev_append` est en position terminale : il retourne directement son résultat à la fonction appelante.

Afin d'implémenter l'optimisation de la compilation des appels en position terminale, notre génération de code Cminor traduit de tels appels en utilisant la construction syntaxique Cminor `tailcall`, et la construction d'appel normal pour les autres appels. Bien entendu, un appel constituant le corps d'une fonction est un appel en position terminale et les appels dans les sous-termes gauches de liaisons ne sont pas en position terminale : ils restent traduits par la fonction précédente.

Traduction des termes en position terminale

Nous présentons maintenant la fonction principale de traduction de termes Fminor en code Cminor principale. Il s'agit de la fonction qui traduira le corps des fonctions Fminor. Nous optimisons ici en transformant chaque appel en position terminale par un appel `tailcall` de Cminor. Seule les appels des sous-termes gauches de liaisons locales ne sont pas en position terminale, c'est pourquoi de tels termes sont traduits par la transformation précédemment énoncée.

Le principe est de retourner directement après exécution du code issu de la traduction du terme Fminor. Ce code respecte les encodages définis plus haut. On note $[\![t]\!]$ la traduction du terme t comme terme en position terminale. La figure 7.3.2 présente la traduction des termes en position terminale.

Ainsi les traductions des atomes sont retournées directement.

$$
\begin{array}{rcl}
[\![a]\!] & = & \texttt{return}\ [\![a]\!] \\
[\![a\ (a_0;\ldots;a_n)^z]\!] & = & \texttt{addrstack} \leftarrow z\ ; \\
 & & \texttt{tailcall}(\sigma(n))[\![a]\!]\ (\texttt{addrstack}, [\![a]\!], [\![a_0]\!], \ldots [\![a_n]\!]) \\
[\![\texttt{let var}\ x\ =\ t_1\ \texttt{in}\ t_2]\!] & = & [\![t_1]\!]_x\ ;\ [\![t_2]\!] \\
[\![\texttt{let root}\ n\ =\ t_1\ \texttt{in}\ t_2]\!] & = & [\![t_1]\!]_{\texttt{tmp\_st}}\ ;\ \texttt{addrstack}[n] \leftarrow \texttt{tmp\_st}\ ;\ [\![t_2]\!] \\
[\![\texttt{match}\ a\ \texttt{with}\ \pi_0;\ldots;\pi_n]\!] & = & \texttt{mkMatch}\ [\![a]\!]\ ([\![\pi_0]\!]\ [\![a]\!], \ldots, [\![\pi_n]\!]\ [\![a]\!]) \\
[\![t]\!] & = & [\![t]\!]_{\texttt{res}}; \texttt{return res}\ \text{sinon} \\
 & & \\
[\![p_1;\ldots;p_n \rightarrow t]\!]\ e & = & \texttt{res} \leftarrow\ e\ ;\ [\![p_1]\!] \leftarrow \texttt{res}[1]; \ldots; [\![p_n]\!] \leftarrow \texttt{res}[n];\ [\![t]\!]
\end{array}
$$

FIG. 7.3.2 – Traduction optimisante des termes Fminor

Les constructions de structures de données sont affectées à la variable réservée `res` qui est tout de suite retournée.

Lors des traductions des liaisons locales, les sous-termes gauches sont traduits par la fonction de traduction des termes en position non terminale. Pour les liaisons de variables à affecter à la variable mise en jeu, pour les emplacements dans l'ensemble des racines à affecter à la variable réservée `tmp_st`.

7.3.5 Traduction d'une fonction

Une fonction Fminor d est assez proche d'une fonction Cminor.

Elle a énuméré ses variables locales d.`vars`. Les variables locales de la fonction Cminor générée seront les mêmes.

La taille du bloc de pile nécessaire à l'exécution du corps de la fonction est stockée dans le champ `stackspace`. Il suffit de considérer les deux mots supplémentaires sur la pile Cminor et de parler en taille de mots mémoire pour obtenir le champ `stacksize` de la fonction Cminor générée, soit : $8 + d.\texttt{rootsize} * 4$.

Les paramètres de la fonction d seront aussi des paramètres de la fonction générée. Cependant, pour l'interaction avec la pile, et le chaînage des blocs de pile en particulier, la nouvelle fonction aura en tête de ses paramètres un paramètre supplémentaire nommé `Roots`.

La traduction du corps de la fonction $[\![d.\texttt{body}]\!]$, traduction d'un terme en position terminale, est précèdée par le chaînage des blocs de pile assuré par l'écriture dans le premier emplacement de la pile courante du paramètre `Roots`.

Les fonctions Cminor sont accompagnées d'une signature `sig`. Ici, toutes les structures de données sont représentées par des pointeurs, qui sont de type `int` en Cminor. La signature d'une fonction Cminor générée prend donc $|d.\texttt{params}| + 1$ arguments de type `int` et rend un objet de type `int`.

Il vient alors la traduction d'une fonction de nom f et de définition d suivante :

$$\begin{aligned}[[(f,d)]] \;=\; (f, \{&\texttt{sig} = \underbrace{\texttt{int} \to \ldots \texttt{int}}_{|d.\texttt{params}|+1} \to \texttt{int} \;;\\ &\texttt{params} = \texttt{Roots}; d.\texttt{params} \;;\\ &\texttt{vars} = d.\texttt{vars} \;;\\ &\texttt{stacksize} = 8 + d.\texttt{rootsize} \times 4 + 4 \;;\\ &\texttt{body} = \texttt{addrstack} \leftarrow \texttt{Roots} \;;\; [[d.\texttt{body}]] \;\})\end{aligned}$$

7.3.6 Traduction d'un programme

Enfin la traduction d'un programme Fminor p est formée de la traduction de chaque fonction de la liste de déclarations de fonction de p, $p.\texttt{defs}$. En tête de la liste de fonction Cminor ainsi obtenue, $[[d.\texttt{defs}]]$, on place la traduction de la fonction principale de p, $p.\texttt{main}$ que l'on nomme `main`. Enfin, on indique comme fonction d'entrée du programme Cminor la fonction de nom `main`.

$$\begin{aligned}[[p]] \;=\; &\{\; \texttt{functs} \;:\; [[(\texttt{main}, p.\texttt{main})]]; [[p.\texttt{defs}]] \;;\\ &\texttt{main} \;:\; \texttt{main} \;;\}\end{aligned}$$

7.4 Axiomatisation et spécification du comportement du gestionnaire de mémoire

Avant de montrer la préservation sémantique de la génération de code Cminor, nous spécifions le comportement du gestionnaire de mémoire.

7.4.1 Que sait-on du gestionnaire de mémoire ?

L'interaction entre le code Cminor généré par traduction de terme Fminor et le gestionnaire de mémoire nous est maintenant familière :

- L'allocateur du gestionnaire, la fonction Cminor `"alloc_block"` qui prend en argument la taille du bloc à allouer et retourne un pointeur sur ce bloc.
- L'enregistrement des racines via le chaînage des blocs de pile successifs, le stockage des racines sur les blocs de pile et les actualisations du nombre de racines sur le bloc avant un déclenchement potentiel de GC.

Cependant, le seul lien direct avec le gestionnaire de mémoire sont les appels à l'allocateur du gestionnaire, dont nous ne connaissons pas le code.

Le comportement attendu d'un appel à la fonction `"alloc_block"` peut être décrit informellement comme suit. Supposons l'appel à `"alloc_block"` dans une configuration d'environnements Cminor telle que le bloc de pile courant soit sp et la configuration mémoire soit m. L'argument passé à `"alloc_block"` est n pour la taille du bloc. L'exécution de l'appel se conclut par :

- Un bloc frais b qui a été alloué dans le tas tel que la taille de b soit n.

- La valeur de retour est un pointeur sur le nouveau bloc $(b, 0)$.
- La nouvelle configuration mémoire m' est telle que toutes les racines et blocs accessibles depuis sp dans m sont conservés dans m'. De plus, les parties globales de m et m' sont identiques.

Nous notons cette dernière propriété : $m \simeq_{sp} m'$.

L'axiomatisation du gestionnaire de mémoire réside dans cette propriété. Nous développons ici l'axiomatisation de cette propriété au travers de la spécification du comportement de la mémoire au cours de l'exécution d'un programme Cminor obtenu par traduction d'un programme Fminor et son interaction avec le gestionnaire de mémoire.

7.4.2 Les différents type de blocs mémoire

Tout d'abord, remarquons que l'on peut distinguer les blocs mémoire selon leurs contenances et la manière dont ils ont été alloués.

Les blocs globaux (`GB`, pour Global Block) Les fonctions sont, au début de l'exécution d'un programme, enregistrées en mémoire, dans la partie globale. A chaque fonction est alors associée un bloc mémoire. Un tel bloc sera appelé bloc global. Un bloc global est alloué en début d'exécution d'un programme. En particulier dans notre transformation, seules les fonctions globales seront placées dans ce genre de blocs. Cela signifie que si b est un bloc global :
- Il a été alloué avant l'exécution du point d'entrée du programme `main`, cela signifie qu'il est plus petit que le `nextblock` de la configuration mémoire initiale, dans notre cas, cela signifie qu'il est négatif.
- Il est présent en mémoire du début à la fin de l'exécution du programme.
- Son contenu n'est jamais modifié.

Les blocs du tas (`HB`, pour Heap Block) A chaque appel à `"alloc_block"` un bloc de tas est alloué. Un bloc de tas contient une structure de données. Mis à part le premier emplacement d'un bloc de tas, les autres emplacements d'un bloc de tas contiennent des pointeurs vers d'autres représentations de structures de données, autrement dit vers d'autres blocs du tas : ils sont accessibles depuis ce bloc de tas. Les blocs de tas accessibles depuis un bloc de tas b ont été alloués avant ce dernier, autrement dit ils sont inférieurs à b [1].

Les blocs de pile (`RB`, pour Root Block) A chaque appel de fonction, Cminor alloue dynamiquement un nouveau bloc de pile. Un bloc de pile est libéré soit à la fin de l'exécution de son corps, soit par un appel en position terminale contenu dans son corps. Un bloc de pile, issu de la génération de code Cminor a une structure bien définie. Le premier emplacement le chaîne à la liste des blocs de pile précédents. Dans le second emplacement, le nombre de racines est stocké. Dans les emplacements suivants sont écrites les racines du corps de la fonction courante. Ces racines sont des représentations de structures de données, il s'agit de pointeurs vers des blocs du tas.

La figure 7.4.2 reprend le chaînage des blocs de pile de la figure 7.2.1 et l'explicite un peu plus. Les racines contenues dans les blocs de pile sont des pointeurs vers des blocs de tas.

[1] En Cminor, les blocs mémoire sont identifiés par des entiers. La fonction d'allocation attribue les identifiants dans l'ordre croissant au cours de l'exécution d'un programme. Autrement dit, l'identifiant du dernier bloc alloué est supérieur aux autres blocs alloués jusqu'ici.

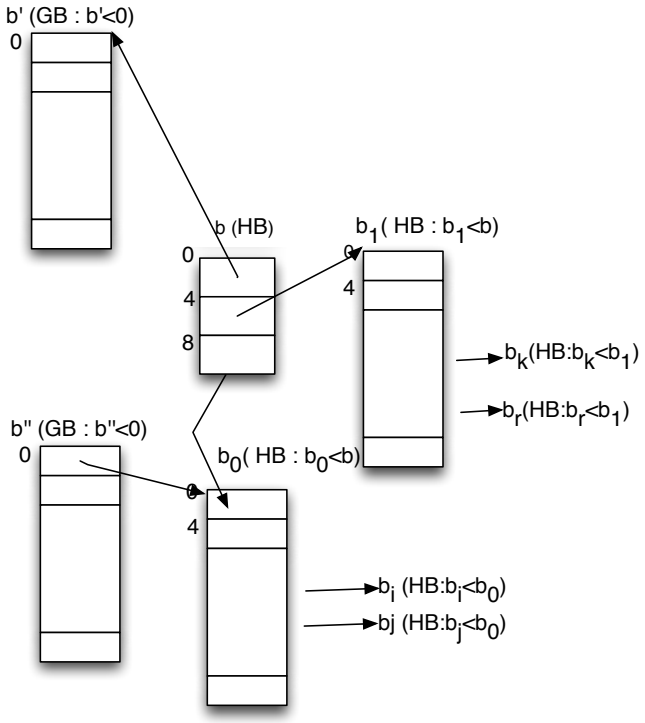


FIG. 7.4.1 – Le bloc de tas b

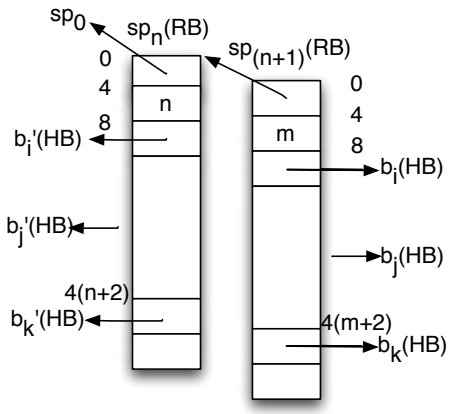


FIG. 7.4.2 – Les blocs de pile

Les types des blocs sont importants dans la spécification du comportement du gestionnaire de mémoire sur la mémoire. Afin de connaître le type de chaque bloc d'une configuration mémoire, on la munit d'une fonction associant à chaque bloc une information de type. Il s'agit d'une fonction partielle qui répond ε si le bloc n'est pas alloué, sinon elle précise de quel type est le bloc :
- `GB` pour un bloc global,
- `RB` pour un bloc de pile et
- `HB` pour un bloc de tas.

Nous appelons une telle fonction un `block_mapping`. Les `block_mapping` vont accompagner les configurations mémoire tout le long de notre formalisation afin de renseigner sur les types des blocs.

Définition 7.4.1 (Cohérence entre un `block_mapping` et une configuration mémoire)
On note $f \vdash m$ le fait que les blocs présents dans m vérifient les informations qui leur sont associées par f. Pour cela, les trois propriétés suivantes doivent être vérifiées :
- *si b n'est pas un bloc valide de m, alors $f(b) = \varepsilon$,*
- *si $f(b) \neq \varepsilon$ alors, b est un bloc valide de m,*
- *si b est un bloc valide de m où $f(b) \neq \varepsilon$.*

Les informations sur les blocs ne changent pas par une écriture. Ainsi, si $f \vdash m_1$, et m_2 est une configuration mémoire issue d'une écriture sur m_1, alors $f \vdash m_2$.

Par contre, les allocations modifient les `block_mapping`. Nous définissons la relation d'extension entre deux `block_mapping`, que l'on note $\subseteq$.

Définition 7.4.2 (Extension de `block_mapping`)
On note $f_1 \subseteq f_2$, le fait que le `block_mapping` *f_2 soit une extension du* `block_mapping` *f_1. On a $f_1 \subseteq f_2$ si :*
- *$\forall b$, si $f_1(b) \neq \varepsilon$ alors $f_1(b) = f_2(b)$,*
- *$\forall b, f_2(b) =$ `GB` alors $f_1(b) =$ `GB`.*

Cette relation est réflexive et transitive.

7.4.3 Accessibilité

Un bloc b' est accessible à partir d'un bloc b s'il existe une chaîne de pointeurs du bloc b au bloc b'. En considérant une configuration mémoire donnée m et un `block_mapping` f en cohérence avec m, $f \vdash m$, on peut raffiner cette notion d'accessibilité. Dans notre formalisation, l'accessibilité qui nous intéresse concerne les blocs accessibles depuis un bloc racine (`RB`). L'accessibilité peut alors se définir par trois relations :
- Accessibilité entre blocs de pile, reflétant le chaînage des blocs de pile.
- Accessibilité entre blocs du tas.
- Accessibilité d'un bloc à partir d'un bloc de pile donné.

Accessibilité entre blocs de pile : Cette relation se note $f, m \vdash R \gg_R R'$ et se lit :"le bloc de pile R' est accessible depuis le bloc de pile R dans la configuration mémoire m selon le `block_mapping` f". Elle se définit par les deux règles suivantes :

$$\frac{f(R) = \texttt{RB}}{f, m \vdash R \gg_R R}$$

$$\frac{f(R) = \texttt{RB} \quad f(R') = \texttt{RB} \quad m(R, 0) = (R', 0) \quad m \vdash R' \quad f, m \vdash R' \gg_R R''}{f, m \vdash R \gg_R R''}$$

Un bloc de pile est accessible depuis lui-même. Le bloc de pile R' auquel le bloc de pile R est directement chaîné est accessible depuis R ainsi que tous les blocs de pile accessibles depuis R'. $f, m \vdash \gg_R$ est donc une relation transitive et réflexive.

Accessibilité entre blocs du tas : Cette relation est notée $f, m \vdash b \gg_H b'$ et se lit : "le bloc de tas b' est accessible depuis le bloc de tas b dans la configuration mémoire m, selon le `block_mapping` f". Elle se définit par les deux règles suivantes :

$$\frac{f(b) = \texttt{HB} \quad m \vdash b}{f, m \vdash b \gg_H b} \qquad \frac{\begin{array}{c} f(b) = \texttt{HB} \quad b.\texttt{low} = 0 \quad 0 \leq n \times 4 < \texttt{max\_block} \\ f(b') = \texttt{HB} \quad m(b, n \times 4) = (b', 0) \quad b' < b \quad f, m \vdash b' \gg_H b'' \end{array}}{f, m \vdash b \gg_H b''}$$

Un bloc de tas valide est accessible depuis lui-même (réflexivité). Tout bloc de tas b' dont le bloc de tas b contient un pointeur est accessible depuis b, de même tout bloc de tas b'' accessible depuis un tel bloc b' est accessible depuis b (transitivité).

Accessibilité : Nous appelons ici accessibilité la relation d'accessibilité d'un bloc de tas à partir d'un bloc de pile. Cela définit exactement l'ensemble de blocs mémoire que doit préserver un GC si on lui indique le bloc de pile à considérer comme racine, tel que nous le définissons dans notre choix de transmission des racines. Nous notons $f, m \vdash R \ggg b$ le fait que le bloc de tas b est accessible depuis le bloc de pile R dans la configuration mémoire m selon le `block_mapping` f.

Définition 7.4.3 (Accessibilité)

$$\frac{f(R) = \texttt{RB} \quad f, m \vdash R \gg_R R' \quad m(R', 8 + n \times 4) = (b, 0) \quad f, m \vdash b \gg_H b'}{f, m \vdash R \ggg b'}$$

7.4.4 Préservation des racines et des blocs accessibles

Nous pouvons maintenant définir dans quelles conditions une configuration mémoire m' a préservé les racines et les blocs accessibles d'une configuration mémoire m à partir d'un bloc de pile *sp*.

On note $m(b)$ le bloc b dans la configuration mémoire m. $m(b) = m'(b)$ signifie que les bornes et les contenus du bloc b sont identiques dans les configurations mémoire m et m'.

Définition 7.4.4 (Préservation des accessibles depuis un bloc de pile)
Les accessibles depuis le bloc de pile sp dans la configuration mémoire m_1, telle que $f_1 \vdash m_1$, sont préservés dans la configuration m_2, telle que $f_2 \vdash m_2$ et $f_1 \subseteq f_2$ si :

- $f_1(sp) = \texttt{RB}$,

– $\forall r$, *si* $f_1, m_1 \vdash sp \gg_R r$ *alors* $m_1(r) = m_2(r)$,
– $\forall b$, *si* $f_1, m_1 \vdash sp \ggg b$ *alors* $m_1(b) = m_2(b)$.
On note cette préservation :$(f_1, m_1) \simeq_{sp} (f_2, m_2)$.

La relation de préservation des racines et accessibles est une relation réflexive et transitive.

Lors de la preuve de préservation sémantique de la génération de code nous nous intéresserons non seulement à la préservation des blocs accessibles depuis le bloc de pile courant mais aussi de la préservation des accessibles depuis le bloc de pile de l'appelant. Une propriété importante de la préservation des racines et accessibles depuis un bloc de pile est la suivante :

Lemme 7.4.5 *Si* $(f_1, m_1) \simeq_r (f_2, m_2)$ *et* $f_1, m_1 \vdash r \gg_R r'$ *alors* $(f_1, m_1) \simeq_{r'} (f_2, m_2)$.

Plus particulièrement :

Définition 7.4.6 (Préservation des racines et accessibles)
On note $(f_1, m_1) \sim_{sp} (f_2, m_2)$ *la préservation des racines et accessibles depuis le bloc de pile appelant du bloc sp c'est-à-dire le bloc pointé par* $m(sp, 0)$.

7.4.5 Spécification de l'allocateur dans le tas

L'interaction avec un gestionnaire de mémoire s'axiomatise dans notre développement au niveau de la spécification du comportement de l'allocateur dans le tas `"alloc_block"`. En effet, c'est lors de l'appel de cette fonction du gestionnaire de mémoire que peut se produire un cycle du GC. Pour que l'allocation ait lieu, il faut que la taille du bloc à allouer soit d'une taille valide dans le modèle mémoire, c'est-à-dire inférieur à la taille maximale d'un bloc, `max_block`. La fonction d'allocation dans le tas prend en paramètre la taille du bloc à allouer. Le bloc alloué doit vérifier la spécification d'un bloc du tas (`HB`) et il doit être plus jeune que les autres blocs de tas présents : son identifiant est supérieur aux autres blocs de tas présents. Enfin, la configuration mémoire issue d'un appel à `"alloc_block"` peut fortement différer de la configuration avant l'appel. Cependant, elle doit avoir préservé les racines et accessibles depuis le bloc de pile courant à son appel.

Définition 7.4.7 (Axiomatisation du comportement de l'allocateur dans le tas)
Considérons l'appel à l'allocateur dans le tas pour allocation d'un bloc de taille n *à affecter à la variable id dans une configuration mémoire* m_1*, telle que* $f_1 \vdash m_1$ *et avec pour bloc de pile courant sp. Si* $4 \times n + 4 <$ `max_block` *alors il existe* m_a*,* f_a *et b tels que :*
– $G, sp \vdash id \leftarrow$ "`alloc_block`"$(\texttt{addrstack}, n), e, m \Rightarrow \texttt{Out\_normal}, \varepsilon[id \leftarrow (b, 0)], m_a$,
– b *dans la configuration mémoire* m_a *a pour borne inférieure* 0 *et pour borne supérieure* $4 \times n + 4$,
– $\forall b'$, *si* $f_1(b') =$ `HB` *alors* $b' < b$,
– $f_a(b) =$ `HB`,
– $(m_1, f_1) \simeq_{sp} (m_a, f_a)$.

7.5 Préservation sémantique

Nous décrivons ici l'ensemble des propriétés et les lemmes de simulation nécessaires à la preuve de préservation sémantique de la transformation d'un programme Fminor en un programme Cminor.

Informellement, ce théorème peut se décrire comme suit :

Théorème 7.5.1 (Préservation sémantique de la génération de code Cminor) *Considérons l'exécution d'un programme Fminor, $\vdash p \Rightarrow v$. Alors il existe une valeur Cminor v' telle que $[\![p]\!]$ s'évalue en v et v' soit observationnellement équivalente à v.*

Ce théorème se montre au travers d'un théorème de simulation. Nous commençons par en décrire les invariants avant d'énoncer les différents schémas de simulation que nous avons prouvés simultanément. Enfin, on énoncera le théorème formel de préservation sémantique.

7.5.1 Les invariants

Nous commençons par définir les propriétés dont nous avons besoin pour mettre en place les invariants de la preuve de préservation sémantique de la traduction de code Fminor en code Cminor. Ces invariants interviennent lors de la preuve de simulation.

Correspondance entre environnements d'évaluation

Correspondance entre environnements globaux Il s'agit de la mise en correspondance de la liste de définitions de fonctions d'un programme Fminor S et l'environnement global d'un programme Cminor G. En plus de vérifier que G contient les traductions des fonctions de S, on vérifie que selon un `block_mapping` initial $f_{\texttt{init}}$, les blocs contenant les codes de fonction sont bien de la sorte `GB`.

Définition 7.5.2 (Correspondance des environnements globaux GM) *Considérons S le liste de définitions de fonctions d'un programme Fminor, G l'environnement global d'un programme Cminor et $f_{\texttt{init}}$ le* `block_mapping` *initial. S correspond à G selon $f_{\texttt{init}}$, noté $S \sim_{f_{\texttt{init}}} G$ si $\forall (x,d) \in S$ il existe b et f tels que :*

- `find_symbol` $(G, x) =$ `Some` b,
- `find_funct` $(G, (b, 0)) =$ `Some` f,
- $[\![(x,d)]\!] = (x, f)$,
- $\|d\|$ *la définition d est bien bornée (voir 6.5.2),*
- $f_{\texttt{init}}(b) =$ `GB`.

On notera par la suite cette propriété dans les énoncés suivants par GM.

Correspondance entre valeurs sémantique La correspondance entre une valeur sémantique Fminor v et un bloc du tas Cminor b dans une configuration mémoire m cohérente avec un `block_mapping` f est notée $f, m \vdash v \sim b$. Cette relation se définit par les deux règles suivantes :

$f, m \vdash (C, v_1; \ldots; v_n) \sim b$ Un constructeur appliqué $(C, v_1; \ldots; v_n)$ correspond au bloc de tas de la configuration m cohérente avec f si b est tel que décrit dans la figure 7.5.1. b est de taille $4 \times n + 4$. Son premier emplacement contient l'entier C et les n autres

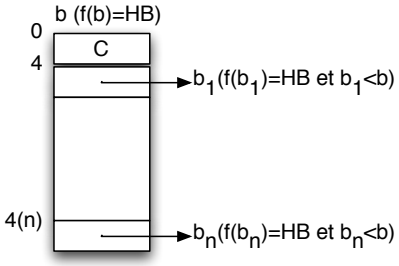


FIG. 7.5.1 – Équivalence entre un constructeur appliqué et un bloc du tas

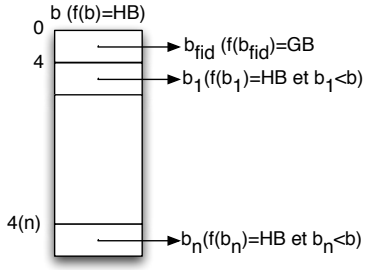


FIG. 7.5.2 – Équivalence entre une valeur fermeture et un bloc du tas

emplacements contiennent des pointeurs vers les blocs $b_1; \ldots; b_n$ tel que : $\forall i, i \in [1; n]$, on a $f, m \vdash v_i \sim b_i$ et $b_i < b$.

$f, m \vdash (fid, v_1; \ldots; v_n) \sim b$ Une valeur fermeture $(fid, v_1; \ldots; v_n)$ correspond au bloc de tas de la configuration m cohérente avec f si b est tel que décrit dans la figure 7.5.2. b est de taille $4 \times n + 4$. Son premier emplacement contient un pointeur vers le bloc global b_{fid} telle que b_{fid} contient le code Cminor de la fonction fid et n autres emplacements contiennent des pointeurs vers les blocs $b_1; \ldots; b_n$ tel que : $\forall i, i \in [1; n]$,
on a $f, m \vdash v_i \sim b_i$ et $b_i < b$.

Correspondance entre environnements de variables La correspondance entre les environnements de variables est l'extension de la relation de correspondance entre valeurs sémantiques Fminor et bloc de tas Cminor à chaque variable contenue dans les environnements :

Définition 7.5.3 (Correspondance entre environnements de variables)
Considérons m et f tels que $f \vdash m$. L'environnement de variables Fminor $e_{\mathtt{var}}$ est en

correspondance avec l'environnement de variables Cminor e' si :
$\forall x$, si $e_{\texttt{var}}(x) = v$, il existe un certain b tel que : $e'(x) = (b, 0)$ et $f, m \vdash v \sim b$.

On note $f, m \vdash_{\texttt{var}} e_{\texttt{var}} \sim e'$ la correspondance entre les environnements de variables Fminor $e_{\texttt{var}}$ et Cminor e' dans une configuration mémoire m cohérente avec f.

Correction de la transmission des racines

Correspondance entre environnement de racines et le bloc de pile courant La correspondance entre l'environnement de racines Fminor et le bloc de pile courant Cminor vérifie les caractéristiques du bloc de pile et son chaînage et étend la relation d'équivalence entre les valeurs sémantiques des racines et le contenu de la pile.

Définition 7.5.4 (Correspondance entre environnement de racines et bloc de pile) *Considérons le bloc de pile courant sp, la configuration mémoire m cohérente selon f et l'environnement de racines Fminor $e_{\texttt{root}}$. L'environnement $e_{\texttt{root}}$ correspond à sp dans m cohérente avec f si :*

- *$f(sp) = \texttt{RB}$,*
- *$m(sp, 0) = (sp', 0)$ tel que : $f(sp') = \texttt{RB}$ et $sp' < sp$,*
- *$\forall n$, si $e_{\texttt{root}}(n) = v$ il existe b tel que : $m(sp, 4 \times (n + 2)) = (b, 0)$ et $f, m \vdash v \sim b$.*

On note $f, m \vdash_{\texttt{root}} e_{\texttt{root}} \sim sp$ la correspondance entre l'environnement de racines Fminor $e_{\texttt{root}}$ et le bloc de pile sp dans une configuration mémoire m cohérente avec f.

Plus que la correction de la transmission des racines, cet invariant assure le bon usage du bloc de pile courant. Ce bon usage est assuré en plus de la correspondance entre l'environnement de racines et le bloc de pile courant par la vérification que le terme Fminor en cours d'évaluation est bien borné (voir la section 6.5.2).

7.5.2 Les théorèmes de simulation

Lors de la preuve de simulation, nous avons à considérer les différents cas de figure introduits par la traduction. En effet, nous avons défini deux traductions des termes Fminor en instructions Cminor. La première sert à la traduction des termes en position non terminale, la seconde concerne les termes en position terminale. Cette dernière fait appel à la première.

Traduction des atomes La traduction d'un atome produit une consultation de l'environnement d'évaluation Cminor.

Théorème 7.5.5 (Simulation traduction des atomes)

Considérons l'évaluation de l'atome a suivante :

$$S, e_{\texttt{var}}, e_{\texttt{root}} \vdash a \Rightarrow v.$$

Sous GM *et les hypothèses :*

H_1 : *$f, m \vdash_{\texttt{var}} e_{\texttt{var}} \sim e$,*

H_2 : *$f, m \vdash_{\texttt{root}} e_{\texttt{root}} \sim sp$,*

H_3 : *$\|a\| \leq q$ avec $4 \times q + 4$ la borne supérieure du bloc sp dans m,*

il existe b tel que :

$$G, (sp, 0), e, m \vdash [\![a]\!] \Rightarrow (b, 0) \text{ et } f, m \vdash v \sim b.$$

Simulation pour la traduction d'un terme en position non terminale La traduction d'un terme Fminor t en position non terminale produit une instruction Cminor qui correspond à une affectation à une variable donnée x de l'expression Cminor qui correspond à t. En plus des invariants communs au théorème 7.5.5, nous avons besoin de savoir que le `block_mapping` d'entrée est une extension du `block_mapping` initial $f_{\texttt{init}}$.

En plus de conclure l'équivalence observationnelle de l'évaluation de l'instruction Cminor produite, la simulation pour la traduction des termes en position non terminale conclut les propriétés sur les environnements d'évaluation qui serviront à reconstruire les invariants pour les environnements intermédiaires. Par exemple, lors de la simulation de la traduction du terme Fminor $\texttt{let}\ x = t_1\ \texttt{in}\ t_2$, les conclusions de la simulation de t_1 servent à la construction des invariants de la simulation de t_2. Enfin, il faut s'assurer de la préservation des racines et accessibles depuis le bloc de pile appelant.

Le théorème de simulation pour la traduction des termes en position non terminale s"énonce comme suit :

Théorème 7.5.6 (Simulation pour les termes en position non terminale)
Considérons l'évaluation du terme Fminor t suivante :

$$S, e_{\texttt{var}}, e_{\texttt{root}} \vdash t \Rightarrow v, e'_{\texttt{var}}, e'_{\texttt{root}}.$$

Sous GM *et les hypothèses :*

H_1 : $f, m \vdash_{\texttt{var}} e_{\texttt{var}} \sim e$,

H_2 : $f, m \vdash_{\texttt{root}} e_{\texttt{root}} \sim sp$,

H_3 : $\|t\| \leq q$ *avec* $4 \times q + 4$ *la borne supérieur du bloc* sp *dans* m,

H_4 : $f_{\texttt{init}} \subseteq f$,

on conclut l'existence de f', e', m' *et* b *tels que :*

C_1 : $G, sp \vdash [\![t]\!]_x, e, m \Rightarrow \texttt{Out\_normal}, e'[x \leftarrow (b, 0)], m'$,

C_2 : $f', m' \vdash v \sim b$

C_3 : $f', m' \vdash_{\texttt{var}} e'_{\texttt{var}} \sim e'$,

C_4 : $f', m' \vdash_{\texttt{root}} e'_{\texttt{root}} \sim sp$,

C_5 : *la borne supérieure du bloc* sp *dans* m' *est* $4 \times q + 4$,

C_6 : $(f, m) \sim_{sp} (f', m')$.

Simulation pour la traduction d'un terme en position terminale La traduction d'un terme Fminor t en position terminale produit une instruction Cminor qui correspond au retour immédiat de la valeur de t. Il y a deux configurations à considérer :

- les termes ne comportant pas d'appel en position terminale,
- les termes comportant au moins un appel en position terminale.

Cette distinction est importante car en Cminor la sémantique des deux sortes d'appel diffère. Lors de l'évaluation d'un appel en position terminale, le bloc de pile de l'appelant est libéré. La preuve de correction de la préservation des accessibles et racines est alors différente du

cas d'un appel en position non terminale. Le chaînage des blocs de pile diffère aussi dans les deux cas.

Les invariants sont identiques à ceux de la simulation pour la traduction d'un terme en position non terminale. L'évaluation de l'instruction issue de la traduction produit un retour du pointeur sur le bloc de tas correspondant à la valeur d'évaluation du terme source. Dans le cas d'un terme ne comportant pas d'appel en position terminale, le retour est un retour normal et les conclusions son identiques à celles d'un terme en position non terminale (voir le théorème 7.5.6).

Dans le cas d'un terme contenant au moins un appel en position terminale, il faut considérer le fait que le bloc de pile de l'appelant a été libéré. Pour cela, on vérifie le chaînage des blocs de pile et on récupère le bloc ayant appelé le bloc libéré et on conclut la préservation des racines et accessibles depuis ce bloc de pile.

Théorème 7.5.7 (Simulation pour les termes en position terminale)
Considérons l'évaluation du terme Fminor t suivante :

$$S, e_{\texttt{var}}, e_{\texttt{root}} \vdash t \Rightarrow v, e'_{\texttt{var}}, e'_{\texttt{root}}.$$

Sous GM *et les hypothèses :*

H_1 : $f, m \vdash_{\texttt{var}} e_{\texttt{var}} \sim e$,

H_2 : $f, m \vdash_{\texttt{root}} e_{\texttt{root}} \sim sp$,

H_3 : $\|q\| \leq a$ *avec* $4 \times q + 4$ *la borne supérieure du bloc* sp *dans* m,

H_4 : $f_{\texttt{init}} \subseteq f$,

on conclut l'existence de e', m' *et out tels que :*

C_1 : $G, sp \vdash [\![t]\!], e, m \Rightarrow out, e', m'$,

ainsi que l'un des deux cas suivants :

Retour d'un appel normal : *il existe* b *et* f' *tels que :*

C_2 : $out = \texttt{Out\_return}(\texttt{Some}(b, 0))$,

C_3 : $f', m' \vdash v \sim b$,

C_4 : $f', m' \vdash_{\texttt{var}} e'_{\texttt{var}} \sim e'$,

C_5 : $f', m' \vdash_{\texttt{root}} e'_{\texttt{root}} \sim sp$,

C_6 : *la borne supérieure du bloc* sp *dans* m' *est* $4 \times q + 4$,

C_7 : $(f, m) \sim_{sp} (f', m')$.

Retour d'un appel en position terminale : *il existe* b, f' *et* sp' *tels que :*

$\mathsf{C}_{2'}$: $out = \texttt{Out\_tailcall\_return}(\texttt{Some}(b, 0))$,

$\mathsf{C}_{3'}$: $f', m' \vdash v \sim b$,

$\mathsf{C}_{4'}$: $f', m' \vdash_{\texttt{var}} e'_{\texttt{var}} \sim e'$,

$\mathsf{C}_{5'}$: $sp[0]\ m = (sp', 0)$,

C_7 : $(f, m) \simeq_{sp'} (f', m')$.

Les théorèmes 7.5.6 et 7.5.7 se démontrent simultanément.

7.5.3 Préservation sémantique de la traduction d'un programme Fminor

Nous sommes maintenant en mesure de définir le théorème formel de préservation sémantique de la traduction d'un programme Fminor en un programme Cminor.

Théorème 7.5.8 (Préservation sémantique de la génération de code Cminor) *Supposons* $\vdash p \Rightarrow v$ *et* $\|p\|$. *Alors il existe* f, m *et* b *tels que :*

$$\vdash [\![p]\!] \Rightarrow (b, 0) \text{ et } f, m \vdash v \sim b.$$

La preuve de ce théorème commence par construire la correspondance entre les environnements globaux et par mettre en place les invariants initiaux. On applique alors le théorème de simulation pour la traduction d'un terme Fminor, il s'agit du théorème combiné des théorèmes 7.5.6 et 7.5.7.

7.6 Conclusion et perspectives

La clé de notre preuve de préservation sémantique réside dans l'utilisation des fonctions de `block_mapping` afin de spécifier la mémoire. Notre preuve opère sur un modèle mémoire abstrait composée de plusieurs blocs mémoire, chaque allocation renvoyant un bloc distinct. Cependant, une implantation concrète d'un gestionnaire de mémoire en Cminor représente naturellement le tas comme un seul bloc, ou un petit nombre de blocs, chaque allocation renvoyant une portion de ce bloc. Il serait intéressant de prouver une correspondance entre l'interaction avec un gestionnaire de mémoire sur notre modèle de mémoire abstrait et celle sur un modèle mémoire plus concret, c'est-à-dire, où le tas serait représenté par un seul bloc.

Une piste pour prouver la correspondance entre un tel modèle mémoire concret et le modèle mémoire abstrait utilisé dans notre formalisation serait une adaptation des fonctions de `block_mapping` en injection de mémoire [12]. Il s'agit d'une relation entre les blocs du modèle mémoire abstrait et des emplacements sur le bloc du tas du modèle concret. Nous avons déjà utilisé de telles injections mémoire dans [11]. L'utilisation combinée des `block_mapping` et des injections permettrait aussi la vérification formelle d'un GC susceptible de déplacer les blocs mémoire (GC à copie, marquage-compactage). Le déplacement de blocs serait reflété au niveau des injections uniquement.

8 Expérimentations

Dans les chapitres précédents nous avons présenté chacune des transformations du compilateur pour miniML vers Cminor, ainsi que les preuves de préservation sémantique. Dans ce chapitre, nous commençons par décrire la fonction qui compose toutes les transformations et montrons la préservation sémantique du compilateur. Puis, nous expliquons comment obtenir un compilateur exécutable et donnons quelques résultats de performance.

8.1 La chaîne de compilation de εML à Cminor

8.1.1 Composition des passes

Nous avons composé notre chaîne de compilation de εML à Cminor, en omettant la transformation CPS. Cette composition est monadique, elle doit donc considérer les cas d'échec possibles. La fonction de compilation d'un programme εML en un programme Cminor est sur la figure 8.1.1.

```
Definition ml2Cminor (p :ml.program) :=
 match (numerotation p) with
 | None => None
 | Some pp =>
      match  (closure_conversion (letrec_elim (decurry pp) nil)) with
      | None => None
      | Some dp =>
          match (root_explicitation (form-inter_monadic dp)) with
          | None =>None
          | Some fp => Some (cminor_gen fp)
          end
      end
 end.
```

FIG. 8.1.1 – Code `Coq` de la fonction de compilation de εML en Cminor.

Les transformations se composent dans l'ordre de présentation dans ce manuscrit. La première transformation est la numérotation des constructeurs (voir la section 2.4.4). Elle

est monadique, en effet, elle rejette la transformation si un constructeur n'a pas été défini, ou bien encore s'il n'y a pas unicité des noms de constructeur dans le programme source. Suit ensuite la décurryfication (voir le chapitre 3), composée avec l'élimination du `letrec` remplacé par `let` et μ (voir la section 3.4.2). On effectue alors l'explicitation des fermetures (voir le chapitre 5). Cette transformation est aussi monadique. Elle fabrique un état, la liste des fonctions globales et s'assure que toute les noms de variables rencontrées sont ceux générés. Nous mettons alors le programme en forme intermédiaire monadique (voir le chapitre 6). On peut alors expliciter l'ensemble des racines (voir le même chapitre 6). Cette transformation est aussi monadique. En plus d'expliciter l'ensemble des racines, la génération de code Fminor rejette les programmes pouvant produire des blocs mémoire dont les tailles seraient incorrectes. Enfin, on génère le programme Cminor capable d'interagir avec un gestionnaire de mémoire automatique au travers de la fonction `alloc_block` (voir le chapitre 7).

8.1.2 Vérification formelle du compilateur pour εML en Cminor

La vérification formelle de notre compilateur réside en sa preuve de préservation sémantique : *Étant donné un programme εML p s'évaluant en une valeur v, si la compilation de p n'échoue pas, alors le programme Cminor compilé s'exécute en produisant une valeur observationnellement équivalente à v.*

Les valeurs sémantiques des deux langages (εML et Cminor) sont fortement distincte. Les valeurs Cminor sont des valeurs observables, on peut observer un pointeur dans une configuration mémoire. Ce n'est pas forcément le cas des valeurs sémantiques de εML (constructeurs appliqués et fermetures). Parmi les valeurs sémantiques de εML, on peut distinguer des valeurs de premier ordre (valeurs observables) :

$$v_1 ::= C(v_1; \ldots; v_n).$$

En effet, les constructeurs constants s'observent par leur identifiant. De même, un constructeur uniquement appliqué à des constructeurs (eux-mêmes soit constants soit appliqués à des constructeurs et récursivement) s'observe par un arbre d'identifiants.

Cependant, dès la première passe du compilateur, les identifiants des constructeurs sont remplacés par leurs numéros (voir la section 2.4.4). De même, un constructeur appliqué est représenté en Cminor par un pointeur sur un bloc de tas dans une certaine configuration mémoire qui, elle-même doit vérifier les informations d'un `block_mapping` donné (voir la section 7.5.1).

Autrement dit, même si nous ne considérions que les valeurs de premier ordre dans notre théorème de préservation sémantique, il nous faudrait définir une relation d'équivalence observationnelle entre les valeurs de premier ordre de εML et celle de Cminor.

En effet, si l'on considère le constructeur constant de εML $C($`nil`$)$, après numération, il correspond au constructeur constant $C(0)$ dans tous les autres langages fonctionnels du compilateur (de εML`#` à Fminor) selon une numérotation Γ. Puis, à la génération de code Cminor, il correspond à un pointeur sur un bloc b d'une configuration mémoire m tel que m vérifie un certain `block_mapping` f qui informe que b est un bloc de tas. Soit :

$$\frac{\Gamma(x) = n \qquad f \vdash m \qquad m(b, 0) = n}{\Gamma, f, m \vdash (x) \sim (b, 0)}$$

Devant la nécessité de construire une équivalence observationnelle pour les valeurs de premier ordre de εML, nous avons défini une équivalence observationnelle pour toutes les valeurs sémantiques de εML. Il faut alors paramétrer notre relation par les états globaux de nos différents langages (les listes de déclarations de fonctions de Fml, Mon et Fminor, et l'environnement global de Cminor). Plus précisément cette relation est la composition des relations de correspondance entre valeurs sémantiques utilisées dans les preuves de préservation sémantique de toutes nos passes. Nous définissons $\Gamma, S_{Fml}, S_{Mon}, S_{Fminor}, G, f, m \vdash \varepsilon v \sim b$, la relation de correspondance entre la valeur sémantique εML εv et le bloc mémoire Cminor b comme suit :

Définition 8.1.1 (Correspondance entre valeurs sémantiques)

$$\Gamma, S_{Fml}, S_{Mon}, S_{Fminor}, G, f, m \vdash \varepsilon v \sim b$$

si :

Existence d'une valeur εML# équivalente : *il existe une valeur $\varepsilon v\#$ telle que : $\Gamma \vdash \varepsilon v \sim \varepsilon v\#$ (voir la section 2.4.4).*

Existence d'une valeur nML équivalente : *il existe une valeur nv telle que : $\varepsilon v\# \sim nv$ (voir la section 3.3.2).*

Existence d'une valeur nML avec μ équivalente : *il existe nv' telle que : $nv \sim nv'$.*

Existence d'une valeur Fml équivalente : *il existe fv telle que : $S_{Fml} \vdash nv' \sim fv$ (voir la section 5.3.1).*

Existence d'une valeur Mon équivalente : *il existe mv telle que : $S_{Fml} \vdash fv \sim mv$ (voir la section 6.3.3).*

Existence d'une valeur Fminor équivalente : *il existe Fv telle que : $S_{Mon}, S_{Fminor} \vdash mv \sim Fv$ (voir la section 6.5.3) et $f, m \vdash Fv \sim b$ (voir la section 7.5.1).*

Nous avons montré le théorème de préservation sémantique de notre compilateur.

```
Theorem Cml2Cminor_correctness :
 ∀  (p : εML programme) ( v : εML valeur) ( cp : Cminor programme)
 (⊢  p  →  v  ) ( ml2Cminor p = Some cp),
  ∃ Γ : numerotation,
   ∃ S_Fml : list (ident*fml_def),
     ∃ S_Mon : list (ident * Mon_funct),
     ∃ S_Fminor : list (ident * fminor_function),
       ∃ f : mapping_block,m : mem, b : block,
       ⊢ cp → (b,0) ∧ Γ, S_Fml, S_Mon, S_Fminor, G, f, m ⊢ v ∼ b.
```

La preuve se fait en construisant les évaluations successives des programmes issus de l'application de chacune des passes. Pour chaque passe on obtient alors l'existence d'une valeur sémantique intermédiaire telle que :

- le programme obtenu par cette passe s'évalue en cette valeur et
- la valeur est une valeur intermédiaire équivalente.

L'application successive dans l'ordre des transformations des théorèmes de préservation sémantique nous permet de construire l'exécution du programme Cminor compilé et l'équivalence des valeurs sémantiques.

8.2 Détails pratiques de la preuve

Le développement et la vérification formelle du compilateur pour εML produisant du code Cminor dans l'assistant de preuve `Coq` représentent deux années de travail. Le tableau présenté sur la figure 8.2.1 donne une idée de l'ordre de grandeur de ce développement via une quantification en lignes de code.

	Coq	*Spec*	*Énoncés*	*Scripts de preuves*	*Autres*	*Total*
Compléments à la bibliothèque Coq		0	182	456	26	664
Langage εML (source)	0	82	519	510	54	1165
Langage εML#	0	57	69	50	4	180
Numérotation des constructeurs	85	0	338	699	73	1195
Langage nML (fonctions n-aires)	0	60	72	66	5	203
Décurryfication	181	0	567	1352	44	2144
Langage nML avec μ	0	248	245	775	66	1334
Élimination du let rec	29	0	126	122	14	291
Transformations CPS	133	279	1552	3973	275	6212
Langage Fml (fermetures explicites)	0	83	0	0	9	92
Explicitation des fermetures	240	0	521	989	59	1809
Langage Mon	0	197	7	9	38	251
Mise en forme intermédiaire monadique	123	0	365	742	50	1280
Langage Fminor (racines explicites)	0	205	120	421	50	796
Explicitation des racines	242	0	737	1475	75	2529
Génération de Cminor	177	0	1546	3371	193	5287
Composition des passes	29	0	14	33	33	109
Total	1239	1211	6980	15043	1068	25541

FIG. 8.2.1 – Détails du développement du compilateur pour εML

La première colonne concerne le code effectif du compilateur. Il s'agit de code `Coq` à extraire : les transformations et leur composition. Le code à extraire représente à peine 5% du développement. La deuxième colonne décrit les spécifications des langages, de leur sémantique ainsi que les fonctions utiles aux preuves (substitution par exemple). La troisième colonne, en plus de comptabiliser les énoncés de lemmes et théorèmes, comptabilise aussi les définition de relations et propriétés. Comme attendu, les scripts de preuves représentent la plus grande partie du développement (un peu moins de 60%).

La partie du compilateur la plus importante (en taille) concerne les transformations CPS. Toutefois, il faut préciser qu'elle contient plus que la définition du langage CPS, les deux transformations et leurs preuves de préservation sémantique. En effet, nous avons défini une sémantique à deux environnements pour le langage CPS et prouver qu'elle était

équivalente à la sémantique par substitution. De plus, nous avons défini une transformation de "retour" vers nML avec μ (comptabilisé dans le code à extraire) dont nous avons prouvé la préservation sémantique.

La génération de code Cminor est aussi une partie conséquente de notre travail, notamment au travers de la spécification de l'interaction avec un gestionnaire de mémoire automatique. La construction des invariants, aussi bien initiaux qu'intermédiaires (invariants à construire pour utiliser les hypothèses d'induction) est un effort important pour la preuve de préservation sémantique de cette dernière transformation.

Nous avons fortement utilisé le module `Coqlib` de CompCert dans nos scripts de preuves. Nous l'avons aussi complété dans notre développement (compléments à la bibliothèque standard de `Coq`). Pour le calcul des racines, nous avons utilisé le module `FSet` de la bibliothèque standard. Enfin, dans nos définitions de langages, nous avons choisi d'utiliser les listes de la bibliothèque standard pour définir des listes de termes au lieu de définir mutuellement des termes_listes. Les avantages sont de bénéficier de l'ensemble des fonctions et lemmes de la bibliothèque standard, éviter la définition pour chaque langage de fonctions telles que la longueur d'une terme_listes et éviter les lemmes de réécriture entre de telles fonctions sur terme_liste d'un langage à l'autre. Le prix à payer est la redéfinition des schémas de récurrence (voir [10]), l'utilisation de points fixes locaux dans nos transformations (en effet, on ne peut pas définir mutuellement une fonction sur terme et liste de terme) et pour les besoin de la preuve définir globalement les fonctions de transformations sur les listes de termes.

Sur un MacBook avec le système d'exploitation Mac OS X version 10.5.5 et pour processeur l'Intel Core 2 duo @1.6 GHz, l'assistant de preuve `Coq` en version 8.1pl2 prend 7min de CPU pour vérifier l'ensemble de la vérification formelle et du développement du compilateur.

8.3 Obtention d'un compilateur exécutable

Le développement du compilateur formellement vérifié est fait en `Coq`, nous utilisons le mécanisme d'extraction de `Coq` vers le langage Caml. Cette extraction provoque aussi l'extraction en `Caml` des bibliothèques standards de `Coq` utilisées.

En tête de notre compilateur, nous avons une transformation initiale qui prend du code Ocaml purement fonctionnel (avec variables nommées) et produit du code εML. Cette transformation se compose du parser d'OCaml, de la transformation des noms de variables en indices de de Bruijn et de l'encodage les fonctions mutuellement récursives.

Nous branchons notre compilateur à l'entrée du back-end CompCert qui produit du code assembleur PowerPC.

8.4 Environnement d'exécution

La compilation efficace d'un langage fonctionnel inclut la présence d'un environnement d'exécution. Aussi avons nous mis en place un tel environnement afin de pouvoir utiliser notre compilateur. Pour cela nous avons ajouté un gestionnaire de mémoire automatique à glaneur de cellules nous fournissant la fonction `alloc_block` que nous utilisons dans notre génération de code Cminor (voir le chapitre 7).

Tests	taille lignes de code	MLComp en ms	ocamlc en ms	ocamlopt en ms	MLComp *vs* ocamlc (%)	MLComp *vs* ocamlopt (%)
exp3_8	29	0.4	1.4	0.25	350	62,5
exp7_20	76	0.06	0.2	0.02	333	33
fib	70	0.1	0.5	0.04	500	33
permut7	242	3	17	7	566	233
heapsort	220	0.3	0.9	0.1	300	33
nqueens	89	10 000	48 000	4 630	480	40
takeushi	31	8	69	2	862	25
KB	698	300	1 000	60	333	20
Moyenne					438	43

FIG. 8.5.1 – Comparaison de performances de codes générés

Il y a deux possibilités de stratégies pour le glaneur de cellules, le marquage-balayage ou bien la copie. Le programme qui est présenté dans l'annexe est le code Cminor d'un glaneur de cellules à copie. Ce code est extrait du jeu de tests du back-end CompCert. C'est l'allocateur de mémoire que nous avons utilisé pour nos tests. Son code est donné en annexe. Cet environnement d'exécution est enrichi de fonctionnalité nous permettant d'observer et d'étudier la compilation. Pour cela, nous avons ajouté un imprimeur de valeurs et une mesure du temps d'exécution.

8.5 Tests et mesures de performances

Nous avons comparé les performances de notre compilateur avec ceux d'ocamlc (produisant du bytecode) et ceux d'ocamlopt (produisant du code natif). Cette mesure de performance consiste à comparer les temps d'exécution des codes compilé par les trois compilateurs. Ces mesures ont été effectuées sur un Apple PowerMac avec deux processeurs 2.0 GHz PowerPC 970 (G5) et 6 Gb de RAM, sur le système d'exploitation MacOS 10.4.11.

La figure 8.5.1 répertorie les mesures de performance à travers les temps d'exécution des exécutables obtenus par les compilateurs ocamlc, ocamlopt et le nôtre (MLCompCert). ocamlc et ocamlopt sont les deux compilateurs du système Objective Caml. ocamlc compile vers du code-octet qui doit être interprêté par une machine abstraite ; tandis que ocamlopt produit du code natif (assembleur). Nous avons utilisé la version 3.10.0 du système OCaml.

Le jeu de test est constitué de 9 programmes allant de 29 à 698 lignes de code. Parmi ces programmes certains ont été extraits par le mécanisme d'extraction de Coq : exp3_8, exp7_20, fib, heapsort, permut7 ; d'autres sont des programmes écrits manuellement en Caml : takeushi, nqueens et KB.

Les tests exp3_8 et exp7_20, calculent respectivement 3^8 et 7^{20} en arithmétique de Peano (le type `nat` de Coq) et en arithmétique binaire (le type `positive` de Coq). Le test

fib calcule *fibonacci* 20 en arithmétique binaire. Le test heapsort, applique la fonction de tri par tas de la librairie standard de Coq à une liste générée d'entiers binaires. permut7 calcule la liste de toutes les permutations d'une liste de 7 entiers de Peano.

Concernant les programmes écrits directement en ML, takeushi teste la fonction de Takeushi appliquée aux entiers naturels 18, 12 et 6. nqueens calcule le nombre de solutions du problème des n reines pour $n = 10$. Enfin, KB désigne l'algorithme de complétion de Knuth-Bendix. L'implémentation utilisée est issue des notes de cours de Gérard Huet [54]

Les performances du compilateur CompCert pour miniML sont raisonnables. Les codes générés par notre compilateur sont environ 4 fois plus rapide que ceux obtenus par ocamlc et 2 à 3 fois plus lent que ceux obtenus par ocamlopt. Les surcoûts par rapport à ocamlopt s'explique d'une part par l'absence d'expansion en ligne (optimisation implantée dans ocamlopt). D'autre part la représentation des constructeurs constants (sans argument) est plus efficace en ocamlopt, ils sont représentés par des entiers tandis que dans le compilateur MLCompCert, ils sont alloués en mémoire. Enfin, l'interaction avec le gestionnaire de mémoire est plus coûteuse, l'enregistrement explicite des racines demande plus de travail que les stratégies de traçage des racines à travers le back-end mises en jeu dans ocamlopt.

9 Conclusions et perspectives

La compilation de langages fonctionnels est un domaine particulièrement bien étudiée. Aussi, nous nous sommes efforcé de présenter ce domaine sous le regard de la vérification formelle.

9.1 Bilan de notre étude

9.1.1 Résultats obtenus

Ce manuscrit nous a permis d'explorer la conception, le développement, la vérification formelle et l'utilisation d'un compilateur optimisant et réaliste pour un langage purement fonctionnel.

Le développement et la vérification formelle du compilateur ont été faits dans l'assistant de preuves `Coq`. Ce compilateur est optimisant et réaliste. L'expressivité de notre langage source vise le fragment purement fonctionnel de mini-ML (sans module et fonction mutuellement récursive). La chaîne de compilation contient une passe de décurryfication et lors de la génération de code Cminor, la compilation des appels en position terminale est optimisée. Notre chaîne de compilation est compatible avec l'intégration d'un gestionnaire automatique de mémoire à glaneur de cellules. Cette chaîne de compilation constitue un front-end compilant du mini-ML en du code Cminor. Le langage Cminor est le langage d'entrée du back-end CompCert produisant de l'assembleur PowerPC.

L'expressivité de notre langage source en fait non seulement un compilateur formellement vérifié pour le fragment purement fonctionnel de mini-ML, mais aussi un maillon de sûreté supplémentaire pour le développement de programmes formellement vérifiés. En effet, les développements menés en `Coq` pourront être compilés par notre compilateur vérifié pour mini-ML purement fonctionnel, via le mécanisme d'extraction de `Coq` (qui lui-même est en cours de vérification [41]). En plus, de la vérification algorithmique apportée par l'utilisation de `Coq`, la compilation vérifiée des développements menés en `Coq`, après extraction, garantit que le code exécutable se comporte comme le code vérifié. Parmi les premiers programmes qui pourraient bénéficier de ce procédé se trouvent les compilateurs CompCert, et le compilateur vérifié pour mini-ML lui-même (voir la section 9.4).

9.1.2 Les langages intermédiaires pour la vérification formelle d'un compilateur

Le choix des langages intermédiaires dans le développement d'un compilateur est important, il est mené par les choix de compilation et les optimisations implantées. Notre expérience nous a appris que dans la vérification formelle d'un compilateur, le choix des

langages intermédiaires est aussi dirigé par les nécessités de la preuve de préservation sémantique.

C'est clairement le cas pour le langage Fminor (voir la section 6.4), non seulement sémantiquement (voir la section 9.1.4) mais aussi syntaxiquement. En effet, la syntaxe explicite l'ensemble des racines et porte sur certaines structures syntaxique le nombre courant d'éléments dans l'ensemble des racines.

Les représentations des variables sont aussi dépendante de la preuve. Une première partie de nos langages, du langage source jusqu'au langage CPS sont dans le formalisme de de Bruijn. Cela nous permet d'éviter les problèmes liés aux noms de variables et à leurs liaisons.

9.1.3 La syntaxe pour encoder des propriétés

Parmi les langages intermédiaires mis en jeu dans notre chaîne de compilation, certains correspondent à l'encodage de style ou forme spécifiques du langage ML. C'est le cas pour les langages CPS (voir le chapitre 4) et Mon (voir le chapitre 6). Au lieu de définir des prédicats sur des termes de mini-ML signifiant qu'ils sont en style CPS ou en forme intermédiaire monadique, nous avons encodé grammaticalement le style CPS et la forme intermédiaire monadique au travers de ces deux langages. Cela nous permet de raisonner sur la préservation sémantique de la mise en style CPS et la mise en forme intermédiaire monadique sans avoir à raisonner sur des prédicats supplémentaires.

9.1.4 De l'usage de la sémantique dans la vérification

Cette étude est bien plus que la réalisation d'un compilateur pour un langage purement fonctionnel réaliste et vérifié formellement. La vérification formelle a dirigé nos choix de conception des différentes transformations. Elle nous a aussi poussé à voir les sémantiques formelles de nos langages sous un regard différent.

La sémantique comme vecteur de propriétés

La sémantique opérationnelle permet de formaliser l'évaluation d'un langage. Dans nos travaux, elle nous a souvent permis de transmettre des propriétés prouvées à une passe aux passes suivantes. Bien souvent, les sémantiques ont servi de vecteurs de propriétés entre des passes non directement voisines. Par exemple, l'unicité des noms est prouvée à leurs créations lors de l'explicitation des fermetures (voir le chapitre 5). Cette unicité est alors transmise jusqu'à la fin de la chaîne de compilation au travers des sémantiques des langages intermédiaires suivants. La preuve de préservation sémantique assume alors que l'unicité des noms est encore effective à la dernière passe.

De même, d'un point de vue plus technique, nous avons réservé trois noms de variables Cminor (voir le chapitre 7) :

- `Root` (le positif `xH`) le paramètre supplémentaire des fonctions Cminor permettant le chaînage des blocs de pile Cminor (voir la section 7.2.2),
- `res` (le positif `xO xH`) variable utilisée dans la génération de code Cminor optimisant (voir 7.3.2),
- `tmp_st` (le positif `xI xH`) variable permettant de lier la traduction liaison locale à un élément de l'ensemble des racines (voir 7.3.2).

Pour cela , il nous a suffit de générer les noms de variables à partir de (`xO xH`) à l'explicitation de fermetures (voir le chapitre 5), ensuite lors de la mise en forme intermédiaire monadique (voir le chapitre 6) l'encodage des noms nous garantit que nos trois variables réservées ne sont pas générées. Cependant, cette information est utile quelques passes plus loin, nous avons utilisé les sémantiques des langages pour transmettre cette propriété jusqu'à la génération de code Cminor.

La sémantique comme spécification

Lors de l'enregistrement effectif des racines (voir la section 6.4), nous avons utilisé la sémantique du langage Fminor (voir la section 6.4.4) afin de spécifier l'interaction avec un gestionnaire de mémoire. Plus précisément, l'évaluation d'un terme Fminor se fait dans deux environnements, l'un concernant les variables, le second concernant l'ensemble des racines et retourne les environnements modifiés après l'évaluation du terme. Ainsi, le bon calcul des racines est spécifié par l'évaluation des termes potentiellement déclencheur de `GC` (structures de données et appels de fonction). Seules les racines sont suffisantes à l'évaluation de tels termes et seules les racines sont nécessaires à l'évaluation de la suite du calcul après l'évaluation de tels termes, par définition d'une racine. Le calcul des racines est alors correct si un terme potentiellement déclencheur de GC s'évalue avec un environnement de variables vide et que son évaluation retourne le même environnement de l'ensemble des racines.

9.2 Perspectives à court terme

Plusieurs améliorations de notre compilateur sont envisageables à court terme, comme nous l'avons évoqué dans certains des chapitres de ce manuscrit.

9.2.1 Une plus grande expressivité

Compilation des fonctions mutuellement récursives

D'un point de vue de l'expressivité, le traitement des fonctions mutuellement récursives serait un bénéfice important. Les termes d'un langage peuvent être décrits par les types concrets mutuellement récursifs. Les fonctions sur ces termes peuvent être mutuellement récursives. Cependant, d'après le lemme de Besik, les fonctions mutuellement récursives s'encodent par des fonctions simplement récursives. Par exemple :

```
let rec f x = g x 1 and g y t = f t + 1 in f (g 4)
```

devient

```
let rec f g x = g x 1 in
 let rec g y t = f g t + 1 in
   f g (g 4).
```

Cette solution n'est pas très efficace d'un point de vue de la compilation. Le traitement des fonctions mutuellement récursives interroge sur la représentation en Cminor des fermetures mutuellement récursives. La section 5.4 décrit différentes représentations classiques de

fonctions mutuellement récursives. Le choix de la représentation des fermetures mutuelles a une influence sur notre spécification de l'interaction avec un gestionnaire de mémoire automatique. La structure des blocs de tas joue un rôle important dans cette spécification.

Compilation d'un système de module

Les développements Coq sont souvent modulaires, notamment ceux concernant des compilateurs vérifiés. L'implantation de notre compilateur est elle-même modulaire. En effet, chaque langage est défini dans un module (un fichier Coq), de même pour chaque transformation. La compilation des modules est donc une amélioration que l'on peut apporter à notre chaîne de compilation bien qu'il soit facile de contourner sa nécessité (expansion en ligne des modules dans un seul fichier, avec quelques modifications de noms simples [107]). Nous pourrions nous inspirer du système de module du système Objective Caml [73] décrit dans [67], l'assistant de preuve Coq dispose d'un système de modules similaire.

Compilation du filtrage

Dans notre compilateur vérifié formellement pour mini-ML, la compilation du filtrage est simple car le filtrage ne s'effectue que sur le constructeur de tête. Bien que les filtrages en profondeur soit encodable par une cascade de filtrages de tête (voir la section 2), une compilation plus efficace des filtrages en profondeur est envisageable. La compilation des filtrages en profondeur est implanté dans les compilateurs pour langages fonctionnels modernes. Essentiellement, la compilation d'un filtrage en profondeur produit un automate de filtrage. Ces automates peuvent être représentés de deux manières différentes : soit par des automates avec *backtracking* [6], soit par des arbres de décision. La compilation en arbre de décision présentée dans [76] présente de bonne propriétés pour être vérifiée formellement. Il n'y a pas de génération de noms de variables et la preuve de préservation sémantique semble être indépendante de l'heuristique utilisée pour choisir l'ordre des tests. L'adaptation de cette compilation de filtrage à notre chaîne de compilation nous permettrait d'obtenir une compilation de filtrages en profondeur optimisée.

9.2.2 Davantage d'optimisations

Bien que les performances de notre compilateur soient tout à fait raisonnables, la mise en place et la vérification d'autres optimisations seraient une étude intéressante.

L'expansion en ligne des fonctions

L'expansion en ligne de code de fonctions (*inlining*) est une optimisation courante dans la compilation de langages fonctionnels. L'idée est simple : considérons l'appel de fonction $f\ M$ où f est lié à une fonction de paramètre x et de corps t ; l'expansion en ligne de cette fonction produit $\texttt{let}\ x\ =\ M\ \texttt{in}\ t$ ce qui élimine le coût d'un appel de fonction. Cependant, toutes les expansions ne sont pas bonnes à effectuer : elles peuvent entraîner une explosion de la taille du code compilé. Cette optimisation met donc en jeu des heuristiques [104, 56]. L'implantation de l'expansion en ligne de fonction dans notre chaîne de compilation apporterait à son efficacité. Cela poserait la question de la spécification des heuristiques.

La décurryfication d'ordre supérieur

A l'heure actuelle, le compilateur effectue une décurryfication de premier ordre qui se trouve être exactement celle implantée dans le compilateur OCaml (voir le chapitre 3). Bien que la question de la décurryfication d'ordre supérieur se pose naturellement, elle n'est implantée dans aucun des compilateurs pour langage fonctionnel à notre connaissance. L'étude de la décurryfication d'ordre supérieur pose de nouveaux défis, notamment au niveau du cadre d'équivalence observationnelle à mettre en place. Des travaux menés avec Xavier Leroy ont été entamés afin de vérifier une décurryfication d'ordre supérieur comme nous l'avons mentionné à la section 3.5.

Raffinement du calcul des racines

Une autre amélioration pourrait être apportée au niveau du calcul des racines. Notre mode de transmission des racines se fait par enregistrement dans une structure dédiée. L'enregistrement d'une variable comme étant une racine se fait lors de sa déclaration, et cette variable reste dans l'ensemble des racines jusqu'à la fin de sa portée lexicale. Un calcul plus fin des racines permettrait qu'une variable reste enregistrée comme étant une racine durant sa durée de vie uniquement, comme nous l'avons évoqué à la section 6.6.

9.3 Perspectives à plus long terme

9.3.1 Développement et vérification d'un environnement d'exécution

Afin de pouvoir exploiter et utiliser notre compilateur, nous l'avons muni d'un environnement d'exécution contenant un gestionnaire de mémoire automatique à glaneur de cellules à copie écrit en Cminor. Pour l'instant ce code n'est pas vérifié.

La vérification de gestionnaire de mémoire automatique est un domaine déjà exploré. Damien Doligez, dans sa thèse [33] proposait la vérification d'un algorithme de glaneur de cellules concurrent en TLA. Plus récemment, Andrew McCreight et *al* [81] ont vérifié dans l'assistant de preuves `Coq` un glaneur de cellules à marquage-balayage et un à copie, écrits dans un langage assembleur. Dans ses travaux de stage de master, Tahina Ramananandro propose des pistes au développement et à la vérification de glaneur de cellules écrits en Cminor [98]. L'adaptation de notre développement afin de le munir d'un de ces gestionnaires de mémoire automatique ajouterait à la sûreté de notre compilateur. Comme nous l'avons évoqué à la la section 7.6, une injection entre notre modèle mémoire et celui utilisé par le gestionnaire de mémoire automatique vérifié sera nécessaire.

Une autre piste envisageable serait de vérifier le code Cminor implantant le GC utilisé dans nos tests. Cela induit la conception d'un outil de raisonnement sur les programmes Cminor, un peu comme l'outil `Caduceus` [38]. Des pistes ont été envisagées par Andrew Appel et Sandrine Blazy [3], comme l'utilisation de la logique de séparation.

9.3.2 Compilation du langage `ML`

Le langage `ML` ne se réduit pas à son fragment purement fonctionnel non typé.

La particularité de ML est son caractère fortement typé : les types n'y sont pas explicites dans la syntaxe mais inférés [25]. Dans un premier temps, notre étude se place dans un cadre où le typage n'est pas nécessaire. En effet, nous plaçant comme compilateur de code extrait par le mécanisme d'extraction de Coq, nous bénéficions des garantis du typeur de Coq. Cependant, le langage traité est aussi le sous-ensemble purement fonctionnel non typé de ML. Dès lors se pose la question de munir notre langage source d'un système de types et notre chaîne de compilation d'une inférence de types. Le premier point délicat serait de déterminer quel système de types serait adapté à notre langage source. Nous ne voulons pas perdre de l'expressivité, et continuer à traiter les programmes issus de l'extraction de Coq. Dès lors, le système de types de Caml semble être opportun. De plus, l'algorithme d'inférence W [25] a déjà fait l'objet de vérifications formelles [35, 87].

De même, compléter le compilateur afin qu'il traite ML dans son intégralité est une perspective naturelle. En effet, ML comporte des traits impératifs : les références, les exceptions, *etc.*...

Les références semblent facilement intégrables : les sémantiques des langages fonctionnels (de εML à Fminor) devraient comporter un état mutable (*store*) et leurs transformations en Cminor, qui est un langage impératif, seraient aisées.

Concernant les exceptions, elles peuvent dans un premier temps être encodées comme des types concrets, ou bien encore par une monade d'exception (un peu comme nous le faisons pour la monade d'erreur). Mais ces encodages mènent à du code peu efficace. Il nous semble intéressant de vérifier des implantations plus efficaces et de plus bas niveau des exceptions [99].

9.4 La touche finale : le bootstrap

Que serait un compilateur qui ne se compile pas lui-même ? Le bootstrap nous semble être une étape essentielle à notre développement. Non seulement pour le sérieux de notre compilateur mais aussi afin de renforcer encore une fois la sûreté. En effet, le bootstrap nous augmenterait le degré de vérification des outils de développement de logiciels vérifiés. Les premiers bénéficiaires seraient naturellement les compilateurs CompCert. Il s'agit de logiciels développés et vérifiés dans l'assistant de preuves Coq. Leurs extractions appartiennent au langage que nous compilons. A l'heure actuelle, ces compilateurs sont extraits vers du code Caml et sont compilés par le compilateur du système Objective caml [73], qui même si développé par des personnes pointilleuses ne sont pas à l'abri d'introduire des bugs. Il en est de même pour notre chaîne de compilation. "Bootstrapper" notre compilateur pour mini-ML, rendrait la compilation de code extrait par le mécanisme d'extraction de Coq indépendante d'outils non vérifiés. Le bootstrap se ferait comme suit :

Extraction en code Caml nous commençons par extraire notre compilateur en code Caml,

Compilation de la première extraction par le compilateur OCaml, nous obtenons un premier exécutable,

Extraction en code mini-ML nous pouvons alors extraire notre compilateur vers le langage d'entrée de notre chaîne de compilation,

Compilation par le compilateur nous compilons alors par le premier exécutable et obtenons un compilateur vérifié compilé par un compilateur vérifié (lui-même).

La listes de perspectives présentées est conséquente (mais sans doute non exhaustive). Cependant, avant d'arriver à bootstrapper le compilateur, il serait raisonnable de commencer par ajouter les fonctions mutuellement récursives et l'optimisation d'expansion en ligne de fonction. En ajoutant les fonctions mutuellement récursives, nous pourrions compiler l'intégrité des programmes issus du mécanisme d'extraction. Enfin, l'expansion en ligne de code de fonction apporterait un gain non négligeable de performance.

A Le programme de GC à copie en Cminor

```
/* A simple stop-and-copy garbage collector */

var "alloc_ptr"[4]
var "fromspace_start_ptr"[4]
var "fromspace_end_ptr"[4]
var "tospace_start_ptr"[4]
var "tospace_end_ptr"[4]

/* Format of blocks :
      - header word : 30 bits size + 2 bits kind
          kind = 0   block contains raw data (no pointers)
          kind = 1   block contains pointer data
          kind = 2   block is closure (all pointers except first word)
          kind = 3   block was forwarded
      - [size] words of data

   Blocks are stored in one big global array and addressed by pointers
   within this block.  The pointer goes to the first word of data.
*/

#define KIND_RAWDATA 0
#define KIND_PTRDATA 1
#define KIND_CLOSURE 2
#define KIND_FORWARDED 3
#define Kind_header(h) ((h) & 3)
#define Size_header(h) ((h) & 0xFFFFFFFC)

/* Copy one block.  The reference to that block is passed by reference
   at address [location], and will be updated. */

"copy_block"(copy_ptr, location) : int -> int -> int
{
  var optr, header, kind, size, src, dst;

  optr = int32[location];
  if (optr == 0) return copy_ptr;
```

```
  header = int32[optr - 4];
  kind = Kind_header(header);
  if (kind == KIND_FORWARDED) {
    /* Already copied.  Reference of copy is stored in the
       first field of original. */
    int32[location] = int32[optr];
  } else {
    /* Copy contents of original block (including header) */
    size = Size_header(header) + 4;
    src = optr - 4;
    dst = copy_ptr;
    {{ loop {
      int32[dst] = int32[src];
      src = src + 4;
      dst = dst + 4;
      size = size - 4;
      if (size == 0) exit;
    } }}
    copy_ptr = copy_ptr + 4;
    /* Mark original as forwarded */
    int32[optr - 4] = header | KIND_FORWARDED;
    int32[optr] = copy_ptr;
    /* Update location to point to copy */
    int32[location] = copy_ptr;
    /* Finish allocating space for copy */
    copy_ptr = copy_ptr + Size_header(header);
  }
  return copy_ptr;
}

/* Finish the copying */

"copy_all"(scan_ptr, copy_ptr) : int -> int -> int
{
  var header, kind, size;

  {{ loop {
    if (scan_ptr >= copy_ptr) exit;
    header = int32[scan_ptr];
    scan_ptr = scan_ptr + 4;
    kind = Kind_header(header);
    size = Size_header(header);
    if (kind == KIND_RAWDATA) {
      /* Nothing to do for a RAWDATA block */
      scan_ptr = scan_ptr + size;
    } else {
      /* Apply [copy_block] to all fields if PTRDATA, all fields except
```

```
         first if CLOSURE. */
      if (kind == KIND_CLOSURE) { scan_ptr = scan_ptr + 4; size = size - 4; }
      {{ loop {
        if (size == 0) exit;
        copy_ptr = "copy_block"(copy_ptr, scan_ptr) : int -> int -> int;
        scan_ptr = scan_ptr + 4;
        size = size - 4;
      } }}
    }
  } }}
  return copy_ptr;
}

/* Copy the roots.  The roots are given as a linked list of blocks :
       offset 0 : pointer to next root block (or NULL)
       offset 4 : number of roots N
       offset 8 and following words : the roots
*/

"copy_roots"(copy_ptr, root) : int -> int -> int
{
  var n, p;

  {{ loop {
    if (root == 0) exit;
    n = int32[root + 4];
    p = root + 8;
    {{ loop {
      if (n == 0) exit;
      copy_ptr = "copy_block"(copy_ptr, p) : int -> int -> int;
      p = p + 4;
      n = n - 1;
    } }}
    root = int32[root];
  } }}
  return copy_ptr;
}

/* Garbage collection */

extern "gc_alarm" : int -> void

"garbage_collection"(root) : int -> void
{
  var heap_base, copy_ptr, tmp;

  copy_ptr = int32["tospace_start_ptr"];
```

```
  copy_ptr = "copy_roots"(copy_ptr, root) : int -> int -> int;
  copy_ptr = "copy_all"(int32["tospace_start_ptr"], copy_ptr) : int -> int -> int;
  /* Swap fromspace and tospace */
  tmp = int32["tospace_start_ptr"];
  int32["tospace_start_ptr"] = int32["fromspace_start_ptr"];
  int32["fromspace_start_ptr"] = tmp;
  tmp = int32["tospace_end_ptr"];
  int32["tospace_end_ptr"] = int32["fromspace_end_ptr"];
  int32["fromspace_end_ptr"] = tmp;
  /* Reinitialise allocation pointer */
  int32["alloc_ptr"] = copy_ptr;
  "gc_alarm"(copy_ptr - int32["fromspace_start_ptr"]) : int -> void;
}

/* Allocation */

extern "out_of_memory" : void

"alloc"(root, kind, size) : int -> int -> int -> int
{
  var p, np;

  size = size * 4;
  loop {
    p = int32["alloc_ptr"];
    np = p + size + 4;
    if (np <= int32["fromspace_end_ptr"]) {
      int32["alloc_ptr"] = np;
      int32[p] = size | kind;
      return p + 4;
    }
    "garbage_collection"(root) : int -> void;
    if (int32["alloc_ptr"] + size + 4 > int32["fromspace_end_ptr"]) {
      "out_of_memory"() : void;
    }
  }
}

/* Initialize a heap of size [hsize] bytes */

extern "malloc" : int -> int

"init_heap"(hsize) : int -> int
{
  var from, to;

  from = "malloc"(hsize) : int -> int;
```

```
  to = "malloc"(hsize) : int -> int;
  if (from == 0 || to == 0) return -1;
  int32["fromspace_start_ptr"] = from;
  int32["fromspace_end_ptr"] = from + hsize;
  int32["tospace_start_ptr"] = to;
  int32["tospace_end_ptr"] = to + hsize;
  int32["alloc_ptr"] = from;
  return 0;
}
```

Bibliographie

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4) :375–416, 1991.

[2] A. W. Appel. *Compiling with continuations*. Cambridge University Press, 1992.

[3] A. W. Appel and S. Blazy. Separation logic for small-step Cminor. In *Theorem Proving in Higher Order Logics, 20th Int. Conf. TPHOLs 2007*, volume 4732 of *Lecture Notes in Computer Science*, pages 5–21. Springer, 2007.

[4] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *16th symposium Principles of Programming Languages*, pages 293–302, 1989.

[5] A. W. Appel and D. A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5) :657–683, 2001.

[6] L. Augustsson. Compiling Pattern Matching. In *Functional Programming Languages and Computer Architecture*, pages 368–381, 1985.

[7] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses : The POPLmark challenge. In *Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005.

[8] H. P. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[9] J. M. Bell, F. Bellegarde, and J. Hook. Type-Driven Defunctionalization. In *International Conference on Functional Programming*, pages 25–37, 1997.

[10] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development – Coq'Art : The Calculus of Inductive Constructions*. EATCS Texts in Theoretical Computer Science. Springer, 2004.

[11] S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. In *FM 2006 : Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006.

[12] S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 2009. Accepté pour publication.

[13] H.-J. Boehm. Simple garbage-collector-safety. In *Programming Language Design and Implementation*, pages 89–98, 1996.

[14] H.-J. Boehm and M. Weiser. Garbage collection in an Uncooperative Environment. *Softw. Pract. Exper.*, 18(9) :807–820, 1988.

[15] L. Cardelli. The functional abstract machine. *Polymorphism Newsletter*, 1(1), 1983.

[16] F. Cardone and J. R. Hindley. History of Lambda-calculus and Combinatory Logic. In D. M. Gabbay and J. Woods, editors, *Handbook of the History of Logic, volume 5*. Elsevier, 2006.

[17] A. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Programming Language Design and Implementation 2007*, pages 54–65. ACM Press, 2007.

[18] A. Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2) :56–68, 1940.

[19] G. E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 2(12) :655–657, December 1960.

[20] Coq development team. The Coq proof assistant. Logiciel et documentation disponibles sur `http ://coq.inria.fr/`, 1989–2008.

[21] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3) :95–120, 1988.

[22] T. Coquand and C. Paulin-Mohrind. Inductively defined types. In P.Martin-Löf and G. Mints, editors, *Proceedings of International Conference on Computer Logic*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[23] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8(2) :173–202, 1987.

[24] H. B. Curry and R. Feys. *Combinatory Logic, volume I.* North-Holland, 1958. Third edition 1974.

[25] L. Damas and R. Milner. Principal type-schemes for functional programs. In *9th symposium Principles of Programming Languages*, pages 207–212. ACM Press, 1982.

[26] O. Danvy and L. R. Nielsen. Defunctionalization at work. In *International Conference on Principles and Practice of Declarative Programming*, pages 162–174, 2001.

[27] O. Danvy and L. R. Nielsen. A first-order one-pass CPS transformation. *Theoretical Computer Science*, 308(1-3) :239–257, 2003.

[28] Z. Dargaye. Décurryfication certifiée. In *Journées Francophones des Langages Applicatifs (JFLA'07)*, pages 119–133. INRIA, 2007.

[29] Z. Dargaye and X. Leroy. Mechanized verification of CPS transformations. In *Logic for Programming, Artificial Intelligence and Reasoning, 14th Int. Conf. LPAR 2007*, volume 4790 of *Lecture Notes in Artificial Intelligence*, pages 211–225. Springer, 2007.

[30] M. A. Dave. Compiler verification : a bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6) :2–2, 2003.

[31] N. G. de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indag. Math.*, 34(5) :381–392, 1972.

[32] A. Dold and V. Vialard. A mechanically verified compiling specification for a Lisp compiler. In *Proc. FST TCS 2001*, volume 2245 of *Lecture Notes in Computer Science*, pages 144–155. Springer, 2001.

[33] D. Doligez. *Conception, réalisation et certification d'un glaneur de cellules concurrent.* Thèse de doctorat, Université Paris 7, 1995.

[34] C. Dubois. Proving ML Type Soundness Within Coq. In *Theorem Proving in Higher Order Logics, 20th Int. Conf. TPHOLs*, pages 126–144, 2000.

[35] C. Dubois and V. Ménissier-Morain. Certification of a Type Inference Tool for ML : Damas-Milner within Coq. *Journal of Automated Reasoning*, 23(3-4) :319–346, 1999.

[36] M. Felleisen and D. P. Friedman. Control operators, the SECD machine and the λ-calculus. In *Formal Description of Programming Concepts III*, pages 131–141. North-Holland, 1986.

[37] R. R. Fenichel and J. C. Yochelson. A LISP garbage-collector for a virtual-memory computer system. *Communications of the ACM*, 12(11) :611–612, November 1969.

[38] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program verification. In *International Conference on Computer aided Verification CAV*, pages 173–177, 2007.

[39] A. Fischbach and J. Hannan. Specification and correctness of lambda lifting. *Journal of Functional Programming*, 13(3) :509–543, 2003.

[40] C. Flanagan, A. Sabry, B. Duba, and M. Felleisen. The essence of compiling with continuations. In *Programming Language Design and Implementation 1993*, pages 237–247. ACM Press, 1993.

[41] S. Glondu. Extraction certifiée dans Coq-en-Coq. In *Journées Francophones des Langages Applicatifs (JFLA'09)*. INRIA, 2009.

[42] A. Goldberg and D. Robson. *Smalltalk-80 : the language and its implementation.* Addison-Wesley, 1983.

[43] A. D. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In *Higher-order logic theorem proving and its applications 1993*, volume 780 of *Lecture Notes in Computer Science*, pages 414–426. Springer, 1994.

[44] M. Gordon. From LCF to HOL : a short history. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction*, pages 169–186. MIT Press, 2000.

[45] B. Grégoire. *Compilation des termes de preuves : un (nouveau) mariage entre Coq et OCaml.* Thèse de doctorat, Université Paris 7, 2003.

[46] J. Guttman, L. Monk, J. Ramsdell, W. Farmer, and V. Swarup. The VLISP verified Scheme system. *Lisp and Symbolic Computation*, 8(1-2) :33–110, 1995.

[47] J. Hannan and P. Hicks. Higher-order arity raising. In *International Conference on Functional Programming*, pages 27–38, 1998.

[48] J. Hannan and P. Hicks. Higher-order uncurrying. In *25th symposium Principles of Programming Languages*, pages 1–11, 1998.

[49] J. Hannan and P. Hicks. Higher-order uncurrying. *Higher-Order and Symbolic Computation*, 13(3) :179–216, 2000.

[50] J. Hannan and F. Pfenning. Compiler verification in LF. In *Logic in Computer Science*, pages 407–418, 1992.

[51] T. Hardin, L. Maranget, and B. Pagano. Functional runtimes within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2) :131–176, 1998.

[52] F. Henderson. Accurate garbage collection in an uncooperative environment. In *The Workshop on Memory Systems Performance (MSP 2002) and The International Symposium on Memory Management (ISMM 2002)*, pages 256–263, 2002.

[53] W. A. Howard. *The formulae-as-types notion of construction*, pages 480–490. Academic Press, 1980.

[54] G. Huet. Formal Structures for Computation and Deduction. *Course Notes*, page 184 pages, May 1986.

[55] T. Johnsson. Lambda Lifting : Transforming Programs to Recursive Equations. In *Functional Programming Languages and Computer Architecture*, pages 190–203, 1985.

[56] S. L. P. Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4&5) :393–433, 2002.

[57] S. P. Jones. *Haskell 98 Language and Libraries : The Revised Report*. Cambridge University Press, 2003.

[58] G. Kahn. Natural semantics. In *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

[59] R. Kelsey, W. Clinger, and J. Rees. The revised^5 report on the algorithmic language Scheme. *SIGPLAN Notices*, 33(9) :26–76, 1998.

[60] A. Kennedy. Compiling with continuations, continued. In *International Conference on Functional Programming 2007*. ACM Press, 2007.

[61] B. W. Kernighan and D. M. Ritchie. *The C programming language*. Addison-Wesley, Second edition, 1988.

[62] D. Kranz, N. Adams, R. Kelsey, J. Rees, P. Hudak, and J. Philbin. ORBIT : an optimizing compiler for Scheme. In *SIGPLAN '86 symposium on Compiler Construction*, pages 219–233. ACM Press, 1986.

[63] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6 :308–320, 1964.

[64] X. Leroy. The ZINC experiment : an economical implementation of the ML language. Technical report 117, INRIA, 1990.

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

[66] X. Leroy. Le système Caml Special Light : modules et compilation efficace en Caml. In *Actes des Journées Francophones des Langages Applicatifs*, pages 111–131. INRIA, 1996.

[67] X. Leroy. A modular module system. *Journal of Functional Programming*, 10(3) :269–303, 2000.

[68] X. Leroy. Formal certification of a compiler back-end, or : programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[69] X. Leroy. The Compcert verified compiler, commented Coq development. Disponible à `http ://gallium.inria.fr/~xleroy/compcert/`, Sept. 2007.

[70] X. Leroy. A Formally Verified Compiler Back-end. July 2008. Submitted.

[71] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1) :1–31, 2008.

[72] X. Leroy, D. Doligez, et al. The Caml Light system, release 0.74. Logiciel et documentation disponibles sur le Web, `http ://caml.inria.fr/`, 1997.

[73] X. Leroy, D. Doligez, J. Garrigue, and J. Vouillon. The Objective Caml system. Logiciel et documentation disponibles à `http ://caml.inria.fr/`, 1996–2008.

[74] P. Letouzey. A new extraction for Coq. In *Types for Proofs and Programs, Workshop TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science*, pages 200–219. Springer, 2003.

[75] P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. Thèse de doctorat, Université Paris-Sud, July 2004.

[76] L. Maranget. Compiling pattern matching to good decision trees. In *Worshop on ML*, pages 35–46, 2008.

[77] S. Marlow and S. Peyton Jones. Making a fast curry : push/enter vs. eval/apply for higher-order languages. *Journal of Functional Programming*, 16(4–5) :375–414, 2006.

[78] J. Mc Carthy and J. A. Painter. Correctness of a compiler for arithmetical expressions. In *Symp. in Applied Mathematics*, pages 33–41, 1967.

[79] C. McBride and J. McKinna. I am not a number ; I am a free variable. In *Proc. 2004 Haskell Workshop*, pages 1–9. ACM Press, 2004.

[80] J. McCarthy. Recursive Functions of Symbolic Expression and Their Computation by Machine. *Communications of the ACM*, 3(4) :184–195, April 1960.

[81] A. McCreight, Z. Shao, C. Lin, and L. Li. A general framework for certifying garbage collectors and their mutators. In *Programming Language Design and Implementation 2007*, pages 468–479. ACM Press, 2007.

[82] R. Milner. A calculus for the mathematical theory of computation. In *International Sympoisum on Theoretical Programming*, pages 332–343, 1972.

[83] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3) :348–375, 1978.

[84] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The definition of Standard ML (revised)*. The MIT Press, 1997.

[85] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *23rd symposium Principles of Programming Languages*, pages 271–283. ACM Press, 1996.

[86] Y. Minamide and K. Okuma. Verifying CPS transformations in Isabelle/HOL. In *MERLIN '03 : Proc. workshop on Mechanized reasoning about languages with variable binding*, pages 1–8. ACM Press, 2003.

[87] W. Naraschewski and T. Nipkow. Type Inference Verified : Algorithm W in Isabelle/HOL. In *Types for Proofs and Programs, Workshop TYPES*, pages 317–332, 1996.

[88] G. C. Necula. Proof-carrying code. In *24th symposium Principles of Programming Languages*, pages 106–119. ACM Press, 1997.

[89] G. C. Necula. Translation validation for an optimizing compiler. In *Programming Language Design and Implementation 2000*, pages 83–95. ACM Press, 2000.

[90] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis.* Springer, 1999.

[91] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions.* Thèse, université Paris 7, Jan. 1989.

[92] S. L. Peyton-Jones. *The implementation of functional programming languages.* Prentice-Hall, 1987.

[93] S. L. Peyton Jones, N. Ramsey, and F. Reig. C- : a portable assembly language that supports garbage collection. In *PPDP'99 : International Conference on Principles and Practice of Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 1–28. Springer, 1999.

[94] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Programming Language Design and Implementation*, pages 199–208, 1988.

[95] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1(2) :125–159, 1975.

[96] G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61 :17–139, 2004.

[97] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS '98*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1998.

[98] T. Ramananandro. Vérification formelle d'une implémentation d'un gestionnaire de mémoire pour un compilateur certifié. mémoire de stage de Master 2, ENS Paris, Sept. 2007.

[99] N. Ramsey and S. L. P. Jones. A single intermediate language that supports multiple implementations of exceptions. In *Programming Language Design and Implementation*, pages 285–298, 2000.

[100] D. Rémy and J. Vouillon. Objective ML : A simple object-oriented extension of ML. In *24th symposium Principles of Programming Languages*, pages 40–53. ACM Press, 1997.

[101] D. Rémy and J. Vouillon. Objective ML : An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1) :27–50, 1998.

[102] J. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM National Conference*, pages 717–740. ACM Press, 1972.

[103] D. S. Scott. A Type-Theoretical Alternative to ISWIM, CUCH, OWHY. *Theor. Comput. Sci.*, 121(1&2) :411–440, 1993.

[104] M. Serrano. Inline Expansion : When and How ? In *International Symposium on Programming Languages, Implementations, Logics, and Programs*, pages 143–157, 1997.

[105] O. Shivers. Control-flow analysis in Scheme. In *Programming Language Design and Implementation 1988*, pages 164–174. ACM Press, 1988.

[106] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991.

[107] J. Signoles. Calcul statique des applications de modules paramétrés. In *Journées françaises des langages applicatifs, JFLA*, pages 21–36, 2003.

[108] G. L. Steele. Rabbit : a compiler for Scheme. Technical Report AI-TR-474, Artificial Intelligence Laboratory,Massachusetts Institute of Technology, May 1978. M.Sc.thesis.

[109] G. J. Sussman and G. L. S. Jr. Scheme : a Interpreter for Extended Lambda Calculus. *Higher-Order and Symbolic Computation*, 11(4) :405–439, 1998.

[110] Y. H. Tian. Mechanically Verifying Correctness of CPS Compilation. In *CATS '06 : Proceedings of the 12th Computing : The Australasian Theory Symposium*, pages 41–51. Australian Computer Society, 2006.

[111] M. Tofte, L. Birkedal, M. Elsman, and N. Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3) :245–265, 2004.

[112] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2) :109–176, 1997.

[113] C. Urban. Nominal Techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4) :327–356, 2008.

[114] C. P. Wadsworth. *Semantics and pragmatics of the lambda-calculus*. Thèse de doctorat, Oxford, 1971.

[115] M. Wand and P. Steckler. Selective and Lightweight Closure Conversion. In *symposium Principles of Programming Languages*, pages 435–445, 1994.

[116] S. Weeks. Whole-program compilation in MLton. In *Worshop on ML*, page 1, 2006.

[117] P. R. Wilson. Uniprocessor Garbage Collection Techniques. In *International Workshop on Memory Management*, pages 1–42, 1992.

[118] A. K. Wright and R. Cartwright. A practical soft type system for Scheme. *ACM Transactions on Programming Languages and Systems*, 19(1) :87–152, 1997.

[119] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1) :38–94, 1994.