N^o d'ordre :

Thèse de doctorat

présentée à

L'Université de Paris-Sud

U.F.R. Scientifique d'Orsay

par

JEAN-CHRISTOPHE FILLIÂTRE

pour obtenir

le grade de docteur en sciences
de l'Université de Paris XI Orsay
spécialité : Informatique

Sujet :

Preuve de programmes impératifs en théorie des types

Soutenue le 12 juillet 1999 devant la commission d'examen composée de

M.	BEAUQUIER Joffroy	président
Mme	VIGUIÉ DONZEAU-GOUGE Véronique	rapporteurs
M.	REYNOLDS John C.	
MM.	BIDOIT Michel	examinateurs
	MURTHY Chet	
Mme	PAULIN Christine	directeur

Preuve de programmes impératifs en théorie des types

Jean-Christophe Filliâtre

Mai 1999

À mes parents

Table des matières

Préface

Me voilà parvenu au terme de ces quatre années de recherche. De terme, en fait, il n'y en a point, et la rédaction de ce mémoire de thèse n'est tout juste qu'une occasion, un peu forcée, de faire le point sur un cheminement scientifique déjà entamé. L'occasion pour moi de constater que j'ai pu combiner, dans un même travail, mes passions respectives pour la rigueur mathématique et le plaisir de la programmation, à travers une activité qui les magnifient toutes deux : la preuve de programmes.

Massy, le 4 mai 1999.

Remerciements

Nombreuses sont les personnes que je tiens à remercier, et j'espère n'oublier personne.

En premier lieu, ma gratitude va à Christine Paulin, pour avoir dirigé mes recherches pendant ces longues années. « Diriger » n'est d'ailleurs pas le mot qui convient. Je devrais plutôt parler d'attention, de disponibilité et de soutien. Comme tous ceux qui la connaissent, je lui témoigne une grande admiration, pour ses qualités scientifiques autant que pour ses qualités humaines.

Je remercie très sincèrement Véronique Viguié Donzeau-Gouge et John Reynolds d'avoir accepté d'être les rapporteurs de cette thèse, et j'espère que mon travail a été à la hauteur de leurs exigences scientifiques.

Joffroy Beauquier et Michel Bidoit me font l'honneur de faire partie de mon jury, et je tiens à leur exprimer ici toute ma gratitude.

J'ai rencontré Chet Murthy alors que j'étais en stage de DEA, et ce fut un réel privilège que de pouvoir travailler avec lui et de profiter ainsi de son immense talent de programmeur. Je le remercie chaleureusement d'avoir accepté de faire partie de mon jury.

Je dois à Olivier Marguin la découverte de l'Informatique « sérieuse », alors qu'il était mon professeur en classes préparatoires. Je lui dois l'apprentissage d'une programmation aussi rigoureuse qu'intéressante, et l'initiation à de nombreux algorithmes, dont *heapsort* que je n'ai pu prouver formellement sans avoir une pensée reconnaissante à son égard.

Mes années de thèse m'ont amené à partager la vie de plusieurs équipes de recherche. Je garde de mon passage à Lyon au sein du Lip l'image d'une équipe chaleureuse, dont je tiens à remercier en particulier pour leur accueil Jean Duprat, Philippe Audebaud et Frédéric Prost. J'ai ensuite « débarqué » dans l'équipe Démons du Lri, en plein milieu de ma thèse. Dire qu'il s'agit d'une équipe accueillante serait un faible mot et j'aimerais tous les remercier ici, à savoir — dans le désordre — Laurence, Evelyne, Mina, Jean-Pierre, Claude, Ralf, Benjamin, Judicaël, Pierre, Alexandre, Frédéric, Eduardo et Mihai. Je remercie tout particulièrement Xavier et Patrick pour leur lecture attentive de mon mémoire. Enfin, je tiens à remercier tous les membres de l'équipe Coq de l'INRIA Rocquencourt, et en particulier Hugo Herbelin et Eduardo Giménez pour leur aide au cours de ces quatre années.

Nicolas Magaud a essuyé les plâtres de mon implantation balbutiante au cours d'un stage de première année de magistère. Au delà de ses qualités scientifiques, j'ai apprécié sa gentillesse et ses critiques toujours constructives.

On oublie souvent, à tort, les obscurs inventeurs de nos outils de tous les jours. En ce qui me concerne, il sont d'ailleurs loin d'être des inconnus, et c'est avec une grande sincérité que je remercie Linus Torvalds pour Linux, Richard Stallman pour Emacs, Donald Knuth pour ses ouvrages et pour TeX sans lequel ce mémoire n'aurait pas eu la même allure, et Xavier Leroy pour Objective Caml.

Enfin, et surtout, je tiens à remercier les membres de ma famille pour tout ce qu'ils m'ont apporté, et en particulier mes parents pour m'avoir permis de m'épanouir à ce point dans mes études. Mes remerciements envers Anne, quant à eux, sont éternels et inexprimables.

Introduction

« Software is hard. »
D. E. Knuth

À la veille du XXIe siècle, cette citation de D. E. Knuth sonne comme un jeu de mots grinçant car, mieux que tous les exemples de pertes financières dues à des bugs informatiques, les frissons que le « bug de l'an 2000 » fait courir sur notre planète fournissent la meilleure preuve de la difficulté de cet exercice qu'est la programmation. S'il en fallait une autre, nous pourrions citer cette anecdote racontée par J. L. Bentley dans *Programming Pearls*[1] [7] : alors qu'il enseignait à des programmeurs professionnels des Bell Labs et d'IBM ayant plusieurs années d'expérience, il leur demandait de coder en deux heures, dans le langage de leur choix, l'algorithme de recherche dichotomique dans un tableau trié — en anglais *binary search*. À l'issue du temps imparti, les programmes étaient testés pendant une demi-heure. Les résultats sont effarants : J. L. Bentley rapporte que sur plusieurs années et plusieurs centaines de programmeurs il observa toujours plus de 90% de programmes incorrects. Il cite alors, pour corroborer ces résultats, cette remarque que D. E. Knuth fait dans le troisième volume de *The Art of Computer Programming* [34], page 419, à propos de *binary search* : seize années se sont écoulées entre la première publication de cet algorithme et sa première version complètement exempte de bugs.

Cette qualité plus que médiocre du logiciel tient au paradoxe suivant : alors qu'elle est *a priori* une science exacte[2], l'informatique est traitée comme une science expérimentale. D'un point de vue historique, l'informatique a été inventée par des mathématiciens, qui ont constitué les premiers programmeurs pendant un certain temps. La programmation est d'ailleurs une manipulation formelle de symboles qui se rapproche par beaucoup d'aspects de l'activité mathématique. Mais l'informatique est rapidement devenue une science expérimentale, reposant plus sur des *tests* que sur des raisonnements. Sans doute est-ce parce que, dans la plupart des cas, il est quasi immédiat de vérifier le résultat d'un programme et que, par conséquent, le test *apparaît* comme la méthode la plus rapide de développement. C'est bien sûr une illusion, le domaine d'un programme étant presque toujours infini.

Et c'est d'autant plus vrai lorsque l'on cherche à calculer plusieurs milliards de décimales de π ou à lancer une fusée pilotée par un programme. Dans le premier cas, on n'a pas de moyen de vérifier le résultat, et dans le second cas ce moyen est bien trop coûteux. Il

[1] Cette anecdote est également rapportée dans *Writing Efficient Programs* [6], page 123.
[2] dixit C. A. R. Hoare dans l'introduction de [24] : « Computer programming is an exact science ».

faut alors se persuader mathématiquement de la correction du programme. Depuis toujours, les mathématiciens ont su se convaincre de la validité de leur travaux par la rigueur et la vérification mutuelle de leurs preuves. Il peut en être de même pour la programmation. D. E. Knuth a élevé la programmation au rang d'un art [32, 33, 34] et ses programmes, comme ses ouvrages, sont presque totalement dépourvus de fautes. D'autres, tels R. W. Floyd, C. A. R. Hoare ou E. W. Dijkstra, ont même formalisé un cadre mathématique de la programmation, donnant aux preuves de programmes la même rigueur que celle des preuves mathématiques. Ces travaux datent des années 70 et on ne peut que constater aujourd'hui qu'ils n'ont pas, ou peu, été suivis sur le plan pratique.

À l'approche mathématique de la programmation, le génie logiciel a préféré des approches moins contraignantes, basées sur l'amélioration des langages de programmation. Ainsi sont apparues des syntaxes plus rigoureuses, les notions de typage, d'encapsulation, d'exceptions, etc. Puis des notations formelles ont été définies pour spécifier les programmes. Enfin, ces dernières années ont vu l'apparition et l'utilisation à une échelle industrielle de méthodes formelles, telles que VDM [27, 28], la méthode B [4] ou encore le projet KIV [51], qui permettent non seulement de spécifier des programmes dans le langage mathématique mais également d'en faire formellement la preuve de correction. Nous sommes donc finalement revenus aux idées des années 70, vingt ans plus tard.

S'il a fallu autant de temps aux méthodes formelles pour se développer, c'est en particulier parce qu'elles nécessitent, d'une part, un langage formel de spécification et donc en particulier la formalisation d'une partie des mathématiques et, d'autre part, des outils de preuve. Paradoxalement, les travaux concernant ces deux aspects — formalisation des mathématiques et outils de preuve associés — se sont surtout développés orthogonalement à la preuve de programme. Ainsi ont vu le jour plusieurs assistants de preuve, tels que Nqthm, PVS, HOL, LEGO, Coq, etc. La plupart implantent des formalismes d'ordre supérieur d'une grande expressivité dont la contrepartie est une relativement faible automatisation des preuves. Ils se présentent donc plus comme des outils de preuve interactive que comme des démonstrateurs automatiques. Certains d'entre eux, comme LEGO et Coq, reposent sur le *Calcul des Constructions Inductives*, une extension du système F de J.-Y. Girard, développée par G. Huet, T. Coquand et C. Paulin [9, 48].

C'est dans ce cadre-là que nous avons développé une méthode de preuve de programmes. Le Calcul des Constructions Inductives est notre langage de spécification et notre méthode a été mise en pratique dans le système Coq [1] qui est donc en particulier notre outil de preuve. En ce qui concerne la méthodologie, nous avons essayé de garder l'*esprit* de la logique axiomatique de C. A. R. Hoare [24]. L'idée d'annoter un programme par des assertions logiques est en effet très naturelle et les obligations de preuve que cette méthode génère (préservation d'invariant, vérification des conditions initiales, etc.) sont intuitives et peu nombreuses. Nous avons donc essayé dans ce travail de réunir d'une part des idées relativement anciennes concernant la preuve de programme et d'autre part des travaux récents concernant les logiques formelles et leurs implantations.

Lorsque l'on envisage la preuve de programme se pose immédiatement la question du langage de programmation. C'est presque un faux problème. Il n'est en effet pas réaliste de chercher à traiter un langage de programmation existant dans son entier, avec toutes ses constructions et toutes ses subtilités sémantiques. Les méthodes formelles ne considèrent qu'un petit nombre de constructions, non ambiguës et bien comprises sur le plan sémantique, et on peut d'ailleurs noter que les codes critiques, tels les codes embarqués, sont d'emblée restreints au niveau du langage de programmation utilisé. Dès lors, il est possible de produire du code pour n'importe quelle syntaxe concrète. Ainsi, la méthode B permet de produire du code C ou Ada. En dehors des constructions usuelles des langages impératifs que sont l'affectation, la conditionnelle, la séquence et la boucle, il nous a semblé important de considérer dès le départ les variables locales, à travers une construction let in, et la panoplie complète des procédures et fonctions, à savoir appel par valeur, par variable, et fonctions récursives. Il s'avère en fait que nous traitons même l'ordre supérieur de façon directe, ce qui constitue un bonus. En ce qui concerne les types de données, nos programmes les partagent avec la théorie des types. Nous aurions pu nous donner dès le départ un nombre fixé de types de base et de constructions de type pour notre langage et les axiomatiser ou les coder dans la théorie des types. Mais en procédant comme nous l'avons fait, nous permettons à l'utilisateur de manipuler des valeurs de n'importe quel type défini dans la théorie des types, comme par exemple un type de listes, sans qu'il soit nécessaire pour autant de modifier le langage et donc la méthode.

Pour pouvoir parler de programmes impératifs dans la théorie des types, qui est un λ-calcul typé, nous avons commencé par traduire les programmes en des termes de ce calcul, donc en des programmes purement fonctionnels. Cette traduction exprime la *sémantique* d'un programme impératif sous la forme d'une fonction et une telle interprétation porte le nom de *sémantique dénotationnelle*. Mais contrairement aux approches théoriques qui se font classiquement dans le domaine, nous avons cherché à définir une traduction aussi fine que possible, proche de ce que l'on ferait « à la main », et ceci dans le but de rendre les obligations de preuve aussi naturelles que possible. Cette traduction repose sur une analyse statique des effets, directement inspirée des travaux de J.-P. Talpin et P. Jouvelot [55], et sur l'utilisation de la notion de *monade* [30, 38, 39, 56, 57] que nous avons dû affiner pour l'occasion.

Une fois définie cette traduction, nous avons bâti dessus la méthodologie de preuve de correction proprement dite. Celle-ci peut être vue comme une extension des travaux de C. Parent [45, 46] qui a défini une méthode de preuve de programmes purement fonctionnels dans le Calcul des Constructions Inductives basée sur l'idée suivante : étant donnés un programme et sa spécification, on construit une preuve de la spécification dont la partie informative est le programme donné, la partie logique restant à établir par l'utilisateur et constituant les obligations de preuve. Dans notre cas, nous procédons de façon similaire : à partir de la traduction fonctionnelle du programme impératif, nous construisons un terme de preuve de la traduction fonctionnelle de sa spécification, obtenue à partir de ses pré- et post-conditions et de ses effets. Les annotations contenues à l'intérieur du programme (invariants de boucles, variants, etc.) servent à compléter la traduction fonctionnelle en un terme de preuve suffisamment riche pour mener à des obligations de preuve qu'il soit possible de prouver. Nous établissons à ce propos un résultat de complétude relative montrant qu'à l'exception des invariants de boucle, les annotations internes d'un programme peuvent être construites automatiquement de manière à conduire à des obligations de preuve prouvables dès lors que le programme satisfait la spécification donnée. Une fois les obligations de preuves établies, on a donc construit une preuve de la spécification et le théorème de réalisabilité [48] nous assure la correction du programme fonctionnel, donc du programme impératif en vertu de la correction de la traduction elle-même.

On peut se demander légitimement s'il ne suffisait pas d'écrire un générateur d'obligations de preuve, c'est-à-dire un programme prenant en entrée un programme et sa spécification et générant en sortie un ensemble de propositions à prouver dont la validité établit la correction du programme. Cette question est d'autant plus légitime que c'est dans cet esprit que sont bâties les méthodes formelles précédemment citées (VDM, B et KIV). Il y a plusieurs réponses, mais la meilleure est sans nul doute celle relative à la validité de la méthode elle-même. Dans le cas d'un générateur d'obligations de preuve, il faut établir sa validité, ce qui est loin d'être évident vu le grand nombre de règles dont il est généralement constitué. Dans notre cas, la seule justification faite au niveau méta est celle de la correction de la traduction fonctionnelle et il n'y a rien à dire concernant les obligations de preuve : si elles peuvent être prouvées, alors la correction est établie par l'argument donné ci-dessus.

Notre contribution apparaît donc sur trois plans. Tout d'abord, nous avons défini une traduction fonctionnelle d'un langage impératif relativement réaliste en combinant diverses techniques existantes. Ensuite, nous avons défini une méthode pour établir la correction totale de ces programmes dans le cadre de la théorie des types reposant sur cette traduction. Enfin, nous avons implanté cette méthode dans le système Coq et l'avons mise en œuvre sur des algorithmes non triviaux. Les trois parties de cette thèse exposent ces trois contributions.

Plan

Le premier chapitre traite de la traduction fonctionnelle de programmes impératifs. Nous commençons par introduire un langage de programmation d'ordre supérieur, muni d'un typage statique et d'une sémantique formelle. Puis nous introduisons une analyse statique de ses effets, dans un cadre aussi général que possible. À cette notion d'effet nous associons alors une notion généralisée de monade, grâce à laquelle nous définissons la traduction proprement dite. Ce premier chapitre se termine sur la preuve de correction sémantique de la traduction fonctionnelle.

Le deuxième chapitre traite de la partie théorique de notre méthode de preuve de programmes dans le Calcul des Constructions Inductives. Nous commençons par un survol rapide des différents systèmes existants, avec une attention particulière concernant la logique de Floyd-Hoare. Nous définissons ensuite notre notion de spécification, sous la forme d'un système de types enrichi, puis celle de programme annoté, et nous y associons un jugement décidant de la bonne formation des annotations. Nous pouvons alors introduire la méthode elle-même, qui consiste en la construction d'un terme de preuve incomplet à partir d'un programme annoté, où les morceaux manquants sont autant d'obligations de preuve. Nous montrons alors la validité de cette méthode, en nous appuyant sur la correction de la traduction fonctionnelle d'une part, et sur le théorème de réalisabilité d'autre part.

Le troisième chapitre traite des aspects pratiques de notre travail. Nous présentons tout d'abord l'implantation que nous avons réalisée dans le système Coq. Nous montrons comment elle se rattache à la théorie exposée dans les chapitres précédents, sans pour autant rentrer dans les détails du code. Nous exposons ensuite plusieurs études de cas réalisées avec cette implantation, qui montrent la viabilité de notre méthode.

Enfin nous tentons de faire une synthèse de ce mémoire dans un dernier chapitre de conclusion. Nous y dégagerons en particulier quelques généralisations et extensions de notre travail nécessaires à l'établissement d'un réel environnement de certification de logiciel.

Chapitre 1

Traduction fonctionnelle de programmes impératifs

Dans ce premier chapitre, nous allons définir une traduction fonctionnelle pour une certaine classe de programmes impératifs. Cette traduction constituera le fondement principal de la méthodologie de preuve de programmes que nous présenterons dans le chapitre suivant.

L'idée d'interpréter un programme par une fonction n'est pas nouvelle : c'est même l'idée centrale de la *sémantique dénotationnelle*. Cependant, nous ne souhaitons pas interpréter les programmes impératifs comme des fonctions d'un état global de la mémoire vers un nouvel état global, comme c'est le cas traditionnellement lorsque l'on fait de la sémantique des langages impératifs. En effet, nous avons un objectif très pragmatique en point de mire, qui est celui d'obtenir des obligations de preuve aussi naturelles que possible où, en particulier, les variables du programme sont directement représentées par des variables de la logique et non à travers une représentation plus ou moins complexe d'adresses et d'états de la mémoire.

C'est d'ailleurs ce que l'on fait lorsque l'on traduit « à la main » un programme impératif en un programme fonctionnel équivalent. Quiconque a déjà fait une telle traduction explicitement, ou même seulement migré d'un langage impératif vers un langage fonctionnel, sait bien que toute boucle peut s'écrire comme une fonction récursive terminale où les références modifiées par la boucle deviennent autant d'arguments de la fonction récursive. Ainsi le programme suivant qui inverse une liste[1] :

$$\begin{array}{lll} \mathit{reverse} & \stackrel{\text{def}}{=} & \textsf{while } !l \neq [\,] \textsf{ do} \\ & & \quad r := (\mathit{hd}\ \ !l) :: !r; \\ & & \quad l := (\mathit{tl}\ \ !l) \\ & & \textsf{done} \end{array}$$

peut se coder de façon équivalente par la fonction récursive suivante :

$$\begin{array}{lll} \mathit{reverse} & \stackrel{\text{def}}{=} & \textsf{let rec } \mathit{rev}\ r\ l = \\ & & \quad \textsf{if } l \neq [\,] \textsf{ then } (\mathit{rev}\ ((\mathit{hd}\ l) :: r)\ (\mathit{tl}\ l)) \textsf{ else } r \end{array}$$

Nous cherchons donc à définir une traduction fonctionnelle aussi *fine* que possible, proche de ce que l'on ferait manuellement au cas par cas. Cela nécessite en particulier une analyse

[1] Les exemples de programmes seront donnés dans la syntaxe concrète de Caml.

statique des *effets* du programme, dans un sens que nous définirons formellement. Ce n'est là que l'application d'idées classiques, essentiellement contenues dans *The Type and Effect Discipline*, le célèbre article de J.-P. Talpin et P. Jouvelot [55].

Une fois cette analyse des effets réalisée, la traduction fonctionnelle peut être définie sur la base de cette information statique. Si l'on peut la définir de manière directe pour un langage de programmation fixé, nous avons souhaité donner autant de généralité que possible à cette traduction. Pour cela, nous avons recouru à la notion de *monade* [30, 38, 39, 56, 57], un concept introduisant des opérateurs génériques permettant de manipuler des traits impératifs dans un cadre purement fonctionnel.

Notre contribution dans ce chapitre peut donc être vue comme une combinaison de deux techniques existantes, l'analyse statique des effets d'une part, et l'utilisation des monades d'autre part, pour parvenir à la définition d'une traduction fonctionnelle de programmes impératifs simple et naturelle.

Dans la première section, nous commencerons par définir le langage de programmation qui sera l'objet de notre étude dans la quasi totalité de ce mémoire. Nous introduirons ensuite dans la section 1.2 l'analyse statique de ses effets, et le système de types qui lui est associé. La traduction fonctionnelle sera alors introduite dans la section 1.3, après une présentation de la notion de monade et des raffinements que nous apportons à celle-ci. Les limitations de notre approche, ainsi que les comparaisons avec d'autres approches, seront exposées dans la dernière section de ce chapitre.

1.1 Un langage de programmation typé

On cherche à définir un langage de programmation *réaliste* avec à l'esprit l'idée d'établir la correction et la terminaison des programmes écrits dans ce langage. Par *réaliste* on entend que ce langage soit suffisamment expressif pour que l'on puisse y écrire aisément des algorithmes non triviaux manipulant des structures de données complexes. Mais par *réaliste* nous entendons également un langage qui ne soit pas aussi vaste que les langages de programmation usuels. Ceux-ci sont en effet si complexes que leur sémantique n'est nulle part correctement formalisée — quand elle l'est. Chaque langage a ses particularités, ses petites astuces, souvent justifiées en termes d'efficacité, mais elles compliquent incroyablement sa sémantique. Une illustration en est la déclaration COMMON du FORTRAN qui permet de faire se chevaucher deux tableaux. Il est clair qu'il ne serait pas réaliste de chercher à traiter ce genre d'aspects des langages de programmation. De même, nous n'envisageons pas de traiter des constructions telles que goto ou break, du moins dans un premier temps.

Vient ensuite la question du style. Parmi les innombrables langages de programmation, on peut dégager plusieurs grandes familles, même si celles-ci se recouvrent souvent. La plus ancienne est la *programmation impérative*, directement héritée du langage machine. Ses représentants les plus usuels sont le FORTRAN, le C, le Cobol et le Pascal. On peut distinguer ensuite la *programmation orientée objet*, qui est apparue plus tard avec la volonté de fournir de l'encapsulation d'une part et de permettre la réutilisation de code d'autre part. Le C++ et Java en sont les deux principaux représentants. Enfin, une troisième grande famille est la *programmation fonctionnelle*, dont le Lisp fut le précurseur et dont les représentants actuels sont SML, Caml et Haskell.

De nombreuses particularités distinguent ces langages les uns des autres. Nous avons tenté de donner une classification détaillée figure 1.1, en retenant un certain nombre de critères en rapport avec les choix que nous devons faire. Les trois premières lignes du tableau indiquent la présence ou l'absence de traits impératifs (données modifiables en place), d'objets et de traits fonctionnels (fonctions comme objets de première classe). Nous indiquons ensuite le type de gestion de la mémoire : celle-ci peut être allouée statiquement uniquement (c'est le cas du FORTRAN), ou également dynamiquement, soit par le programmeur (dyn.) soit à travers un *garbage collector*[2] (GC). La ligne suivante indique si le langage est typé statiquement (*i.e.* les erreurs de typage sont détectées au moment de la compilation) ou dynamiquement (une exécution peut échouer sur une erreur de typage). Enfin, nous indiquons la présence d'exceptions et la possibilité d'encapsulation, que ce soit à travers des classes ou un langage de modules.

	prog. impérative			*prog. objet*		*prog. fonctionnelle*		
	FORTRAN	C	Pascal	C++	Java	Lisp	Haskell	OCaml
Traits impératifs	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$		$\surd$
Objets				$\surd$	$\surd$		$\surd$	$\surd$
Traits fonctionnels						$\surd$	$\surd$	$\surd$
Gestion mémoire		dyn.	dyn.	dyn.	GC	GC	GC	GC
Typage		stat.	stat.	dyn.	dyn.		stat.	stat.
Exceptions					$\surd$	$\surd$	$\surd$	$\surd$
Encapsulation				classes	classes		classes	modules classes

FIG. 1.1: Classification des langages de programmation usuels

Il est clair que cette classification est loin d'être exhaustive. En dehors du fait que certains critères ne sont pas pris en compte (efficacité, portabilité, bibliothèques, etc.), il manque également bon nombre de langages de programmation, y compris parmi les plus répandus, tels que Ada, Cobol, différentes variantes du Basic, etc. Nous avons seulement cherché à illustrer les points qui nous semblent pertinents lorsque l'on se pose le problème de prouver des programmes. Et parmi ceux-ci la gestion de la mémoire et le typage nous semblent des critères essentiels.

Gestion de la mémoire. Des programmes conséquents manipulent le plus souvent des structures de données dont les dimensions ne peuvent être connues à la compilation. Imposer des tailles maximales à ces données peut s'avérer frustrant pour l'utilisateur[3], voire impossible lorsque la taille des données ne peut être bornée *a priori*. La plupart des langages permettent donc une allocation dynamique de mémoire. En C, C++ ou Pascal, cette allocation doit être faite manuellement, ainsi que par la suite la libération de l'espace mémoire qui a été alloué. C'est une contrainte pour le programmeur, et surtout la source de nombreux *bugs*. C'est

[2] Une traduction française courante est *ramasse-miettes* ; une autre, plus élégante, est *glaneur de cellules*.
[3] TeX en est un triste exemple.

pour pallier à ces inconvénients qu'a été introduite la notion de *garbage collector*. Il s'agit d'une bibliothèque dont la tâche est d'allouer et de libérer la mémoire. L'allocation se fait sur demande du programme et la libération de manière automatique, celle-ci pouvant s'effectuer lorsqu'un bloc n'est plus référencé dans aucun autre bloc. Ainsi, si l'on suppose par exemple que l'on a défini une structure de listes, l'ajout d'un élément à une liste existante se fera en Caml par l'application d'un constructeur ou en Java par la création d'un nouvel objet (à travers un appel à new) ; mais en C la même opération sera généralement de la forme

$l = (\mathsf{struct}\ list*)\ \mathsf{malloc}(\mathsf{sizeof}(\mathsf{struct}\ list))\,;$
$l \rightarrow val = e\,;$
$l \rightarrow next = l_1\,;$

avec à la charge du programmeur la libération ultérieurement du bloc mémoire alloué à l'adresse l, ce qui est loin d'être évident car il faut alors déterminer si ce bloc est ou non toujours référencé quelque part dans une autre structure de donnée.

Dans l'optique de prouver des programmes, une gestion explicite de la mémoire semble assez complexe : outre le fait qu'un certain nombre de conditions sont liées à l'allocation et à la libération elles-mêmes (le bloc est-il de la bonne taille ? est-il convenablement initialisé ? peut-il être libéré ? etc.), c'est surtout la manipulation explicite de pointeurs et les *alias* qu'elle engendre qui rendront les preuves très difficiles. De plus, l'utilisation d'un GC permet de ne pas se soucier de la gestion de la mémoire, tant dans la définition formelle d'un langage de programmation que dans son utilisation pratique. Dès lors, il en sera de même pour les preuves concernant les programmes écrits dans ce langage. Pour toutes ces raisons, nous opterons sans hésiter pour un langage avec GC.

Typage. L'autre point qui nous semble fondamental dans le choix d'un langage de programmation est son caractère *typé*. Par cela, nous entendons qu'une vérification est faite, au moment de la compilation (typage statique) ou de l'exécution (typage dynamique), que le type de tout argument passé à une fonction est bien celui qui est attendu, cela afin d'éviter les lapsus dans les programmes et les plantages à l'exécution qui en résultent[4]. Si certains vieux langages ne sont pas typés (FORTRAN, Basic, Lisp), les langages plus récents incorporent tous du typage. Si celui-ci est statique et s'il parvient à être complet, on dit alors que le langage a la propriété de *typage fort* : elle garantit que l'évaluation d'une expression de type τ donne, si elle termine, un résultat de type τ. Les langages fonctionnels ont généralement cette propriété, et c'est le cas en particulier pour Haskell et Caml. La plupart des langages orientés objets ont, quant à eux, une part de typage statique et une part de typage dynamique, car la classe exacte d'un objet ne peut pas nécessairement être déterminée au moment de la compilation.

Si l'on rajoute le fait que des exceptions et un bon mécanisme d'encapsulation nous semblent incontournables dans un langage de programmation réaliste, il ressort de notre analyse que seuls Java et Caml restent en lice. Je suis personnellement bien plus à l'aise avec la programmation fonctionnelle qu'avec la programmation orientée objet, et par conséquent le choix n'a pas été difficile. Mais nous verrons plus loin que des critères plus objectifs entrent

[4] Un *Segmentation fault* sous UNIX et le droit de rebooter sa machine sous Windows !

également en jeu, tels que l'existence d'une sémantique formelle ou l'adéquation des types récursifs avec les types inductifs de Coq.

Notre langage de programmation sera donc formé d'une grande partie du langage Caml, essentiellement restreint au niveau du typage pour permettre l'analyse des effets. Nous commencerons par introduire son système de types (1.1.1), que nous chercherons à laisser le plus ouvert possible, puis sa syntaxe abstraite (1.1.2), ses règles de typage (1.1.3) et enfin sa sémantique formelle (1.1.4).

1.1.1 Système de types

On s'attend naturellement à ce que notre langage contienne les types de données usuels, tels que les entiers, les booléens ou les nombres à virgule flottante, mais aussi un certain nombre de constructions de types complexes, nécessaires à l'écriture de programmes réalistes. De tels constructions sont par exemple le produit cartésien, les enregistrements (*records*) ou les sommes, tous trois des cas particuliers de types récursifs, ou encore les tableaux ou les pointeurs.

Chaque langage dispose de ses propres constructions de types, plus ou moins puissantes et plus ou moins permissives. En fait, il n'est pas nécessaire de fixer ici de constructions particulières, car notre discours en sera relativement indépendant. Il nous importe seulement de distinguer les *types mutables*, dont les objets sont modifiables en place, et les *types fonctionnels*, dont les objets ne le sont pas.

On ne fera donc aucune supposition particulière quant au système de types fonctionnels, si ce n'est qu'il contient un type int des entiers relatifs, un type bool des booléens, à deux valeurs true et false, et un type unit ayant une unique valeur void. Pour le reste, il suffit de considérer qu'il existe un certain nombre de constructeurs de types d'arités quelconques. Un tel constructeur peut être constant *i.e.* sans argument, comme c'est le cas pour les types unit, bool ou int, ou n-aire, comme par exemple l'opérateur de produit cartésien.

Le système de types comprend d'autre part une construction pour le type des fonctions, notée $\rightarrow$, et une construction pour les types polymorphes, notée $\forall$, dont les variables seront dénotées par des lettres grecques. Enfin, le système de types mutables est supposé ne contenir pour l'instant que le type des références sur des objets de type purement fonctionnel, noté τ ref. On a donc finalement la définition du système de types suivant :

Définition 1 (types) *Les expressions de type sont définies par la grammaire suivante :*

$$\tau \quad ::= \quad (\tau, \ldots, \tau)c \mid \alpha \mid \tau \rightarrow \tau \mid \forall \alpha.\tau \mid \tau \text{ ref} \tag{1.1}$$

où α est une variable de type et c un constructeur de type fonctionnel. Si un type τ ne contient pas la construction ref *on dit qu'il est* pur*, et on le notera « τ* pur *».*

Nous aurons besoin par la suite de l'opération consistant à substituer une expression de type τ_0 à une variable de type α dans une expression de type τ, ce que l'on notera $\tau[\alpha \leftarrow \tau_0]$. Cette substitution est définie de la manière suivante :

Définition 2 (substitution) *La substitution $\tau[\alpha \leftarrow \tau_0]$ est définie par récurrence sur la structure de τ par les équations suivantes :*

$$\begin{array}{lcl}
((\tau_1,\ldots,\tau_n)c)[\alpha \leftarrow \tau_0] & = & (\tau_1[\alpha \leftarrow \tau_0],\ldots,\tau_n[\alpha \leftarrow \tau_0])c \\
\alpha[\alpha \leftarrow \tau_0] & = & \tau_0 \\
\beta[\alpha \leftarrow \tau_0] & = & \beta \qquad \beta \neq \alpha \\
(\tau \rightarrow \tau')[\alpha \leftarrow \tau_0] & = & \tau[\alpha \leftarrow \tau_0] \rightarrow \tau'[\alpha \leftarrow \tau_0] \\
(\forall\alpha.\tau)[\alpha \leftarrow \tau_0] & = & \forall\alpha.\tau \\
(\forall\beta.\tau)[\alpha \leftarrow \tau_0] & = & \forall\beta.(\tau[\alpha \leftarrow \tau_0]) \qquad \beta \neq \alpha \\
(\tau\ \mathsf{ref})[\alpha \leftarrow \tau_0] & = & \tau[\alpha \leftarrow \tau_0]\ \mathsf{ref}
\end{array}$$

Une telle substitution peut nécessiter un renommage préalable des variables liées de τ, pour éviter la capture de variable, que nous supposerons toujours implicitement par la suite.

1.1.2 Syntaxe abstraite

Nous allons maintenant définir la syntaxe abstraite de notre langage, en introduisant successivement ses différentes constructions. Pour une plus grande clarté et une adéquation avec les exemples donnés, nous avons choisi une syntaxe proche de la syntaxe concrète de Caml. Mais nous nous garderons bien de définir une syntaxe concrète, les exemples étant toujours suffisamment parenthésés pour être non ambigus.

Notre langage comporte tout d'abord un noyau purement fonctionnel, qui est celui du système F [16, 17], c'est-à-dire un λ-calcul typé du second ordre, auquel nous ajoutons un ensemble de constantes qu'il n'est pas nécessaire de préciser. Les constructions sont donc les suivantes :

$$e \quad ::= \quad c \mid x \mid \mathsf{fun}\ (x:\tau) \rightarrow e \mid \mathsf{Fun}\ \alpha \rightarrow e \mid (e\ e) \mid (e\ \tau)$$

Pour introduire les fonctions récursives, on pourrait se contenter d'ajouter un opérateur de point fixe général sous la forme d'une constante de type $\forall\alpha\beta.((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)) \rightarrow (\alpha \rightarrow \beta)$, comme cela se fait souvent dans les présentations formelles, mais les fonctions récursives nécessiteront souvent par la suite un traitement particulier. Les expressions de programmes sont donc augmentées d'une construction particulière :

$$e \quad ::= \quad \ldots \mid \mathsf{rec}\ f\ : \tau = e$$

On remarquera qu'en l'absence d'inférence de types, le type du résultat de f doit être donné. Cela sera justifié dans la section 1.1.3 consacrée au typage.

On ajoute ensuite une conditionnelle, parce qu'elle ne peut être considérée comme une fonction lorsque celles-ci sont strictes (ce qui sera le cas comme nous l'indiquerons dans la section 1.1.4), et une construction let in, car elle permet de séquencer les calculs avec effets, à la manière de la traditionnelle séquence, tout en liant le résultat de la première expression dans la seconde :

$$e \quad ::= \quad \ldots \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e$$

On notera qu'on obtient la construction let rec in en combinant les constructions rec et let in.

Vient ensuite la partie impérative proprement dite. Indépendamment de tout trait impératif particulier, on peut déjà introduire la séquence et la boucle, qui forment avec la conditionnelle déjà introduite le noyau des langages impératifs :

$$e \quad ::= \quad \ldots \mid e \; ; \; e \mid \mathsf{while}\ e\ \mathsf{do}\ e\ \mathsf{done}$$

Notons cependant que ces deux nouvelles constructions peuvent être définies en fonction des précédentes, et ne sont donc pas réellement primitives ; en effet, la séquence est une forme particulière du let où le résultat de la première évaluation n'est pas lié dans la seconde, et la boucle while peut être construite à partir de la conditionnelle et d'une fonction récursive. Nous conservons cependant ces constructions comme primitives, car elles interviennent de manière évidente dans les programmes impératifs. En revanche, nous n'introduirons pas de construction for primitive, celle-ci pouvant être aisément définie à l'aide d'une boucle while ou mieux encore d'une fonction récursive (voir page 119).

Enfin, les références sont manipulées à travers trois primitives de *création*, d'*accès* et de *modification*, auxquelles on donne la syntaxe traditionnelle de ML :

$$e \quad ::= \quad \ldots \mid \mathsf{ref}\ e \mid\ !x \mid x \ := \ e$$

On remarquera que l'affectation se fait sur une variable et non sur une expression quelconque (on aurait eu alors la syntaxe $e \ := \ e$). Ce choix, qui n'est guère contraignant en pratique, sera justifié par la suite, lorsque nous introduirons l'analyse des effets (section 1.2).

Définition 3 (syntaxe abstraite) *La syntaxe abstraite de notre langage est donnée ci-dessous.*

$$
\begin{array}{rcl}
e & ::= & c \\
 & \mid & x \\
 & \mid & (e\ e) \\
 & \mid & (e\ \tau) \\
 & \mid & \mathsf{fun}\ (x : \tau) \to e \\
 & \mid & \mathsf{Fun}\ \alpha \to e \\
 & \mid & \mathsf{rec}\ f\ : \tau = e \\
 & \mid & \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
 & \mid & \mathsf{let}\ x = e\ \mathsf{in}\ e \\
 & \mid & e\ ;\ e \\
 & \mid & \mathsf{while}\ e\ \mathsf{do}\ e\ \mathsf{done} \\
 & \mid & \mathsf{ref}\ e \\
 & \mid & !x \\
 & \mid & x\ :=\ e
\end{array}
$$

FIG. 1.2: Syntaxe abstraite

Nous introduirons les tableaux par la suite, dans le chapitre 3, car les considérer dès à présent n'apporte rien en ce qui concerne le discours théorique mais augmente au contraire le nombre de constructions à considérer. Les tableaux subiront la même contrainte syntaxique que les références, à savoir qu'ils seront nécessairement désignés par une variable, sous la forme $x[e]$ et $x[e] := e$. Il est important de noter qu'avec les tableaux ce langage comprendra alors toutes les constructions bien typées du FORTRAN (mise à part la directive COMMON), toutes celles du Pascal qui ne font pas intervenir de pointeurs et bien sûr les constructions usuelles des noyaux des langages fonctionnels typés.

1.1.3 Typage

Il y a essentiellement deux formes de typage : le *typage déclaratif* et *l'inférence de type*. Les langages de programmation comme Pascal ou Java ont un typage déclaratif : le programmeur déclare le type des constantes, des variables et des fonctions et le compilateur vérifie que les expressions sont bien typées. Dans certains cas, comme celui du Pascal, le typage s'avère être décidable au moment de la compilation, et dans d'autres cas, comme celui de Java, il ne l'est pas entièrement et certaines vérifications doivent être faites pendant l'exécution.

La programmation fonctionnelle a introduit une nouvelle forme de typage : *l'inférence de type*. Le programmeur ne donne plus que les définitions de types, et le compilateur infère seul le type de chaque expression. Ceci se justifie pleinement en présence de polymorphisme et d'ordre supérieur, où les types des fonctions peuvent être complexes, et où un style déclaratif ne serait pas toujours évident[5]. L'un des intérêts majeurs de l'inférence de type est la généralisation du type de chaque définition, locale ou globale. En présence de références il s'avère que cette généralisation doit être soumise à certaines restrictions pour que la propriété de typage fort puisse être préservée (voir [37] à ce sujet). Les règles d'inférence de type doivent alors être raffinées et l'algorithme d'inférence lui-même devient plus complexe.

En ce qui nous concerne, nous ne souhaitons pas entrer dans ces détails techniques d'inférence de types polymorphes en présence d'objets mutables, qui sont hors propos ici. Nous avons donc choisi un typage *déclaratif* avec *polymorphisme explicite*, ce qui signifie que les objets doivent être explicitement généralisés par rapport à des variables de types, et explicitement appliqués à des expressions de type par la suite. Les règles de typage ne seront donc plus que des règles de *vérification du typage*, et non d'inférence de type. Cette restriction ne nuit en rien à la présentation théorique, et en pratique on peut tout à fait envisager en amont de notre langage une première phase d'inférence de type sur un langage non déclaratif, selon le schéma suivant :

$$\begin{array}{c}\text{programme}\\\text{non typé}\end{array} \longrightarrow \boxed{\text{inférence de type}} \longrightarrow \begin{array}{c}\text{programme}\\\text{explicitement typé}\end{array} \longrightarrow \boxed{\text{typage}} \longrightarrow \cdots$$

Nous allons maintenant définir les environnements de typage et donner les règles de typage pour notre langage.

Environnements.

Définition 4 (environnements) *Les environnements de typage sont définis par la grammaire suivante :*

$$\Gamma \quad ::= \quad \emptyset \mid \Gamma, \alpha \mid \Gamma, x : \tau$$

[5] À noter tout de même que Caml n'interdit pas le style déclaratif.

Ce sont donc des listes finies composées de variables de types et d'association de types à des variables. Bien qu'un environnement ne puisse être considéré comme un ensemble, vu la répétition possible d'éléments, on se permettra la notation $x : \tau \in \Gamma$, avec la définition suivante : il existe des environnements Γ_1 et Γ_2, tel que $\Gamma = \Gamma_1, x : \tau, \Gamma_2$, avec x n'apparaissant pas dans Γ_2.

Il sera souvent question par la suite de l'ensemble des références d'un environnement donné, et c'est pourquoi nous introduisons la définition suivante :

Définition 5 *L'ensemble des références d'un environnement Γ est défini par :*

$$Refs(\Gamma) \quad \stackrel{def}{=} \quad \{\ x \mid \exists\tau.\ x : \tau \ \mathsf{ref} \in \Gamma\ \}$$

Règles de typage.

On commence par définir les jugements « *le type τ est bien formé dans l'environnement Γ* » et « *l'environnement Γ est bien formé* », que l'on notera respectivement $\Gamma \vdash \tau\ \mathsf{bf}$ et $\Gamma\ \mathsf{bf}$. Cela consiste uniquement à vérifier que les variables de types apparaissant dans τ existent bien dans Γ *i.e.* qu'il n'y a pas de variable de type libre par rapport à un environnement donné, et que les types apparaissant dans un environnement sont tous bien formés.

Définition 6 (types bien formés) *Le jugement $\Gamma \vdash \tau\ \mathsf{bf}$ est défini inductivement sur la structure de τ par l'ensemble des règles d'inférence suivantes :*

$$\frac{\forall i \in 1 \ldots n.\ \Gamma \vdash \tau_i\ \mathsf{bf}}{\Gamma \vdash (\tau_1, \ldots, \tau_n)c\ \mathsf{bf}} \qquad \frac{\Gamma \vdash \tau_1\ \mathsf{bf} \qquad \Gamma \vdash \tau_2\ \mathsf{bf}}{\Gamma \vdash \tau_1 \to \tau_2\ \mathsf{bf}}$$

$$\frac{\alpha \in \Gamma}{\Gamma \vdash \alpha\ \mathsf{bf}} \qquad \frac{\Gamma, \alpha \vdash \tau\ \mathsf{bf}}{\Gamma \vdash \forall\alpha.\tau\ \mathsf{bf}} \qquad \frac{\Gamma \vdash \tau\ \mathsf{bf}}{\Gamma \vdash \tau\ \mathsf{ref}\ \mathsf{bf}}$$

Le jugement $\Gamma\ \mathsf{bf}$ est défini inductivement sur la structure de Γ par les trois règles d'inférence suivantes :

$$\emptyset\ \mathsf{bf} \qquad \frac{\Gamma\ \mathsf{bf} \qquad \alpha \notin \Gamma}{\Gamma, \alpha\ \mathsf{bf}} \qquad \frac{\Gamma\ \mathsf{bf} \qquad \Gamma \vdash \tau\ \mathsf{bf}}{\Gamma, x : \tau\ \mathsf{bf}}$$

On introduit ensuite le jugement « *le programme e a le type τ dans l'environnement Γ* », que l'on notera $\Gamma \vdash e : \tau$.

Définition 7 (typage) *Soit Γ un environnement bien formé. Le jugement $\Gamma \vdash e : \tau$ est défini inductivement sur la structure de e par l'ensemble des règles d'inférence données figure 1.3.*

1.1.4 Sémantique

Nous allons maintenant donner une sémantique à notre langage de programmation. Historiquement, la définition d'une sémantique relève de deux objectifs assez différents : d'une part, l'établissement d'une *référence*, c'est-à-dire une définition non ambiguë du mode d'évaluation du langage, dans le but de permettre par exemple l'écriture de compilateurs par

$$\frac{Type(c) = \tau}{\Gamma \vdash c : \tau}(\mathsf{CONST}) \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}(\mathsf{VAR})$$

$$\frac{\Gamma \vdash \tau \ \mathsf{bf} \qquad \Gamma, x : \tau \vdash e : \tau_1}{\Gamma \vdash \mathsf{fun}\ (x : \tau) \to e : \tau \to \tau_1}(\mathsf{FUN}) \qquad \frac{\Gamma, \alpha \vdash e : \tau \qquad \alpha \notin \Gamma}{\Gamma \vdash \mathsf{Fun}\ \alpha \to e : \forall \alpha.\tau}(\mathsf{FUNT})$$

$$\frac{\Gamma \vdash e : \tau_1 \to \tau \qquad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash (e\ e_1) : \tau}(\mathsf{APP}) \qquad \frac{\Gamma \vdash e : \forall \alpha.\tau \qquad \Gamma \vdash \tau_1 \ \mathsf{bf}}{\Gamma \vdash (e\ \tau_1) : \tau[\alpha \leftarrow \tau_1]}(\mathsf{APPT})$$

$$\frac{\tau = \tau_1 \to \tau_2 \qquad \Gamma \vdash \tau \ \mathsf{bf} \qquad \Gamma, f : \tau \vdash e : \tau}{\Gamma \vdash \mathsf{rec}\ f\ : \tau = e : \tau}(\mathsf{REC})$$

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau}(\mathsf{COND})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau}(\mathsf{LET})$$

$$\frac{\Gamma \vdash e_1 : \mathsf{unit} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ ;\ e_2 : \tau}(\mathsf{SEQ}) \qquad \frac{\Gamma \vdash e_1 : \mathsf{bool} \qquad \Gamma \vdash e_2 : \mathsf{unit}}{\Gamma \vdash \mathsf{while}\ e_1\ \mathsf{do}\ e_2\ \mathsf{done} : \mathsf{unit}}(\mathsf{LOOP})$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{ref}\ e : \tau\ \mathsf{ref}}(\mathsf{REF}) \qquad \frac{x : \tau\ \mathsf{ref} \in \Gamma}{\Gamma \vdash !x : \tau}(\mathsf{DEREF}) \qquad \frac{x : \tau\ \mathsf{ref} \in \Gamma \qquad \Gamma \vdash e : \tau}{\Gamma \vdash x := e : \mathsf{unit}}(\mathsf{AFF})$$

FIG. 1.3: Règles de typage

diverses équipes ; et d'autre part, d'un point de vue plus théorique, l'établissement de propriétés du langage, telle que le *typage fort* ou la correction de réécritures de programmes telles que celles qu'effectue par exemple un compilateur. C'est ce dernier point qui nous intéresse, puisque nous chercherons à montrer la correction sémantique d'une traduction de programmes que nous définirons plus loin. Avant de donner une définition formelle de la sémantique, nous allons commencer par la décrire de façon informelle.

Avant tout, nous choisissons une sémantique *stricte* c'est-à-dire *d'appel par valeur*. Ce choix se justifie par la présence d'effets de bords. En effet, si une sémantique d'appel par nom convient parfaitement à des langages purement fonctionnels[6], il n'en est pas de même en présence d'effets de bords, car la séquentialité des évaluations devient difficile à saisir pour le programmeur.

Il convient ensuite de définir l'ordre d'évaluation des différentes composantes de chaque construction du langage. Nous adoptons les choix sémantiques du langage Caml (voir [36]), qui sont assez naturels :

– Dans une conditionnelle if e_1 then e_2 else e_3, le test e_1 est tout d'abord évalué puis, s'il s'évalue en true alors l'expression e_2 est évaluée, sinon e_3. On constate donc que la conditionnelle ne peut pas se voir comme une fonction, du fait de son évaluation « paresseuse » de e_2 et e_3.

– Dans une liaison let $x = e_1$ in e_2, l'expression e_1 est évaluée, puis l'expression e_2 est évaluée dans un contexte où x est liée au résultat de e_1.

– Dans les expressions ref e et $x := e$, l'expression e est évaluée avant d'être associée à une nouvelle référence ou affectée à une référence existante.

– Enfin, dans une séquence e_1 ; e_2 les expressions e_1 et e_2 sont évaluées successivement, comme on s'en doute.

Plus arbitraire est le choix de l'ordre d'évaluation des arguments d'une fonction. Il est cependant nécessaire de le fixer, puisque ces évaluations peuvent produire des effets de bords. Nous adoptons là encore le choix sémantique de Caml, qui consiste à évaluer l'argument avant la fonction. Par conséquent, lors d'une application multiple, les arguments sont évalués de droite à gauche.

La sémantique est encore à ce point imprécise, mais nous allons maintenant en donner une définition rigoureuse. Il y a pour cela plusieurs techniques, qui conduisent à des preuves de natures très différentes.

Un premier type de sémantique est la *sémantique dénotationnelle* : elle consiste à construire un *modèle*, encore appelé domaine, dans lequel on *interprète* les programmes. Montrer une propriété d'un programme peut alors se ramener à montrer une propriété de son interprétation. C'est ainsi que R. Milner a montré la propriété de typage fort du noyau fonctionnel de ML. Il a été suivi dans cette voie par L. Damas, M. Abadi et d'autres encore, pour des fragments plus importants de ML, incorporant parfois des références. Mais à chaque fois le domaine a du être reconstruit, et les preuves, bien plus complexes, entièrement refaites.

[6]Une sémantique d'appel par nom s'avère en fait même indispensable en pratique dans un langage purement fonctionnel, par exemple pour pouvoir écrire une fonction *gensym* à l'aide d'une liste infinie $[1; 2; 3; \ldots]$.

Un deuxième type de sémantique est la *sémantique opérationnelle structurelle*. Plus intuitive, elle consiste à définir inductivement un jugement $E \vdash e \Rightarrow v$ qui s'interprète par « dans l'environnement E, le programme e s'évalue en la valeur v ». Cette sémantique porte l'adjectif de *structurelle* car les règles d'inférence pour établir ce jugement sont basées sur la syntaxe du programme. Elle fut utilisée par M. Tofte pour montrer la propriété de typage fort de ML.

Mais la sémantique la plus intuitive que je connaisse est sans nul doute celle définie par A. K. Wright et M. Felleisen dans leur article *A Syntactic Approach to Type Soundness* [59]. Basée sur des techniques de réécriture, elle est très proche de ce que l'on ferait « à la main » si l'on devait calculer l'évaluation d'un programme donné. Elle se rapproche de la sémantique opérationnelle structurelle, mais avec une notion plus élémentaire de réduction, qui la classe dans la catégorie des sémantiques *à petits pas*. C'est cette sémantique que nous allons adopter.

Avant d'introduire cette sémantique formelle, nous pouvons simplifier nos expressions de programmes pour rendre sa définition et sa manipulation plus aisée. Nous pouvons tout d'abord nous débarrasser des types : en effet, le typage sert à restreindre les expressions de programmes mais n'est plus utile pour l'évaluation une fois qu'il a été vérifié[7]. D'autre part, certaines expressions de programmes, comme la séquence ou la boucle while peuvent s'exprimer à l'aide de constructions plus simples. Nous commençons donc par définir une première interprétation sémantique, notée *Sem*, de la manière suivante :

$$
\begin{array}{rcl}
\mathit{Sem}(\mathsf{Fun}\ \alpha \rightarrow e) & = & \mathit{Sem}(e) \\
\mathit{Sem}((e\ \tau)) & = & \mathit{Sem}(e) \\
\mathit{Sem}(\mathsf{fun}\ x : \tau \rightarrow e) & = & \lambda x.\mathit{Sem}(e) \\
\mathit{Sem}(\mathsf{rec}\ f\ \ : \tau = e) & = & (\mathsf{Y}\ \lambda f.\mathit{Sem}(e)) \\
\mathit{Sem}(e_1\ ;\ e_2) & = & \mathsf{let}\ \_ = \mathit{Sem}(e_1)\ \mathsf{in}\ \mathit{Sem}(e_2) \\
\mathit{Sem}(\mathsf{while}\ e_1\ \mathsf{do}\ e_2\ \mathsf{done}) & = & ((\mathsf{Y}\ \lambda w.\lambda u.\mathsf{if}\ \mathit{Sem}(e_1)\ \mathsf{then\ let}\ \_ = \mathit{Sem}(e_2)\ \mathsf{in}\ (w\ \mathsf{void}) \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else\ void})\ \mathsf{void}) \\
\mathit{Sem}(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3) & = & \mathsf{if}\ \mathit{Sem}(e_1)\ \mathsf{then}\ \mathit{Sem}(e_2)\ \mathsf{else}\ \mathit{Sem}(e_3) \\
\mathit{Sem}((e_1\ e_2)) & = & (\mathit{Sem}(e_1)\ \mathit{Sem}(e_2)) \\
\mathit{Sem}(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) & = & \mathsf{let}\ x = \mathit{Sem}(e_1)\ \mathsf{in}\ \mathit{Sem}(e_2) \\
\mathit{Sem}(\mathsf{ref}\ e) & = & \mathsf{ref}\ \mathit{Sem}(e) \\
\mathit{Sem}(x := e) & = & x := \mathit{Sem}(e) \\
\mathit{Sem}(e) & = & e, \quad \text{sinon}
\end{array}
$$

où $\lambda x.e$ désigne la λ-abstraction usuelle, et Y un opérateur de point fixe. Lorsque nous évoquerons par la suite la sémantique d'un programme e, nous évoquerons tacitement celle de $\mathit{Sem}(e)$ s'il n'y a pas d'ambiguïté possible.

Comme le titre de l'article de A. K. Wright et M. Felleisen le suggère, la sémantique qu'ils définissent reste très proche de la syntaxe. Elle consiste en effet à définir des règles de réécriture *syntaxique* des programmes. Pour cela, les auteurs distinguent syntaxiquement les

[7] L'information de typage peut être utile pour l'optimisation de l'évaluation mais n'est plus utile ici dans le cadre de la définition d'une sémantique formelle.

expressions des valeurs, de la manière suivante :

$$\begin{array}{llll} \textit{expressions} & e & ::= & v \mid (e\ e) \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \theta.e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\ \textit{valeurs} & v & ::= & c \mid x \mid \mathsf{Y} \mid \lambda x.e \mid \mathsf{ref} \mid\ !\ \mid\ :=\ \mid (:=\ v) \\ \textit{états} & \theta & ::= & \{(x, v), \ldots, (x, v)\} \end{array}$$

On constate que les constructions ref, ! et := ne sont pas considérées comme des constructions syntaxiques particulières, mais comme de simples valeurs ; cela permet de réduire le nombre de cas à étudier dans bon nombre de situations.

On remarque également qu'une construction $\theta.e$ a été ajoutée : elle représente l'expression de programme e dans l'*état* θ, un état étant une assignation finie de valeurs à des variables représentant des références. Cette assignation est représentée par un ensemble θ de couples (x, v) où x est une variable et v une valeur. On note $dom(\theta)$ l'ensemble $\{x \mid \exists v.(x, v) \in \theta\}$. On suppose que chaque variable de $dom(\theta)$ apparaît exactement une fois comme premier élément d'un couple de θ. Dès lors, on s'autorisera la notation $\theta(x)$ lorsque $x \in dom(\theta)$ pour désigner l'unique v tel que $(x, v) \in \theta$. Enfin, on définit sur les états l'opération $\uplus$ de la manière suivante :

$$\theta_1 \uplus \theta_2 \quad \stackrel{\text{def}}{=} \quad \{(x, v) \mid (x, v) \in \theta_2 \lor ((x, v) \in \theta_1 \land x \notin dom(\theta_2))\}$$

On peut voir la construction $\theta.e$ comme l'abréviation syntaxique suivante :

$$\begin{array}{rcl} \{(x_1, v_1), \ldots, (x_n, v_n)\}.e & \equiv & \mathsf{let}\ x_1 = \mathsf{ref}\ v_1\ \mathsf{in} \\ & & \quad \ddots \\ & & \qquad \mathsf{let}\ x_n = \mathsf{ref}\ v_n\ \mathsf{in}\ e \end{array}$$

Les auteurs définissent alors des *notions de réductions*. Il y en a six pour le fragment fonctionnel, et cinq pour le fragment impératif.

$$\begin{array}{rcllr} (c\ v) & \longrightarrow & \delta(c, v) & \text{si } \delta(c, v) \text{ est défini} & (\delta) \\ (\lambda x.e\ v) & \longrightarrow & e[x \leftarrow v] & & (\beta) \\ \mathsf{let}\ x = v\ \mathsf{in}\ e & \longrightarrow & e[x \leftarrow v] & & (\mathit{let}) \\ (\mathsf{Y}\ v) & \longrightarrow & (v\ \lambda x.((\mathsf{Y}\ v)\ x)) & & (Y) \\ \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 & \longrightarrow & e_1 & & (\mathit{if\text{-}true}) \\ \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 & \longrightarrow & e_2 & & (\mathit{if\text{-}false}) \\ \\ (\mathsf{ref}\ v) & \longrightarrow & \{(x, v)\}.x & x \text{ fraîche} & (\mathit{ref}) \\ \theta.R[!x] & \longrightarrow & \theta.R[v] & (x, v) \in \theta & (\mathit{deref}) \\ \theta.R[x\ :=\ v] & \longrightarrow & \theta \uplus \{(x, v)\}.R[\mathsf{void}] & x \in dom(\theta) & (\mathit{assign}) \\ \theta_1.\theta_2.e & \longrightarrow & \theta_1 \uplus \theta_2.e & & (\theta_{\mathit{merge}}) \\ R[\theta.e] & \longrightarrow & \theta.R[e] \quad \text{si } R \neq [\,] & & (\theta_{\mathit{lift}}) \end{array}$$

où l'on suppose s'être donné une interprétation δ des constantes du langage, comme une fonction partielle prenant en argument une constante et une valeur et retournant une valeur. Dans les réductions ci-dessus, R représente les contextes dans lesquels on peut manipuler l'état i.e. accéder à la valeur d'une référence ou la modifier. Ces contextes sont définis par la grammaire suivante :

$$R \quad ::= \quad [] \mid (R\ v) \mid (e\ R) \mid \mathsf{let}\ x = R\ \mathsf{in}\ e \mid \mathsf{if}\ R\ \mathsf{then}\ e\ \mathsf{else}\ e$$

Pour définir la fonction d'évaluation, les auteurs commencent par définir la relation $\longmapsto$ par

$$E[e] \longmapsto E[e'] \quad \text{si et seulement si} \quad e \longrightarrow e'$$

où le *contexte d'évaluation* E est défini par

$$E \quad ::= \quad [] \mid (E\ v) \mid (e\ E) \mid \mathsf{let}\ x = E\ \mathsf{in}\ e \mid \mathsf{if}\ E\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \theta.E$$

Cette définition de E fixe l'ordre d'évaluation pour les différentes composantes du langage. On notera qu'en particulier $\mathsf{ref}\ E$ et $x := E$ sont des contextes d'évaluation. Les auteurs montrent alors que la relation $\longmapsto$ est en fait une fonction (voir [59] page 64). D'autre part, la réécriture $\longmapsto$ ne peut terminer que sur une construction de la forme $\theta.v$, tout autre construction étant nécessairement un rédexe. On peut donc énoncer le lemme suivant :

Lemme 1 (évaluation, [59]) *Soit e une expression. Alors soit la réduction de e par $\longmapsto$ ne termine pas, soit il existe un état θ et une valeur v uniques tels que $e \longmapsto^\star \theta.v$.*

Le résultat principal de l'article de A. K. Wright et M. Felleisen est la propriété de typage fort (*type soundness*) pour un noyau de ML comprenant des références et des exceptions, c'est-à-dire le fait qu'une expression bien typée de type τ s'évalue en une valeur de type τ dès lors que son évaluation termine. Notre langage entre dans ce cadre, avec des règles de typage identiques, et possède donc également la propriété de typage fort.

Par la suite, nous aurons également besoin du lemme suivant, qui se montre par une récurrence immédiate sur la dérivation $\longmapsto^\star$:

Lemme 2 *Soit θ un état et e une expression tels que $\theta.e \longmapsto^\star \theta'.e'$. Alors on a*

$$dom(\theta) \ \subseteq \ dom(\theta')$$

1.2 Effets

Dans l'optique que nous nous sommes fixée de traduire des programmes impératifs vers des programmes fonctionnels, une première étape consiste à en analyser les *effets*. Par effets, nous entendons les comportements à l'exécution qui ne font pas partie du noyau purement fonctionnel. Pour le langage que nous venons de présenter, il s'agit en l'occurrence d'*effets de bord* sur les références (création, lecture et modification), mais dans un langage plus riche, la levée d'exceptions ou les entrées-sorties seraient également considérées comme des effets.

De nombreuses recherches ont été faites sur l'analyse statique des effets dans les programmes, qu'ils soient impératifs (effets de bord, *aliasing* dans les pointeurs ou les tableaux, etc.) ou fonctionnels (levée d'exceptions). Le plus célèbre des articles du domaine est sans nul doute *The Type and Effect Discipline* de J.-P. Talpin et P. Jouvelot [55], dans lequel les auteurs définissent un système de types avec effets pour un langage fonctionnel polymorphe comprenant des traits impératifs, et un algorithme donnant le type principal et l'effet minimal d'un programme. De même, J.-C. Guzmán et A. Suárez ont défini, dans *An Extended Type System for Exceptions* [20], un système de types et un algorithme de typage pour un langage fonctionnel avec exceptions qui rend compte des levées potentielles d'exceptions dans les programmes.

Nous nous sommes inspirés de ces travaux pour définir un système de types avec effets pour notre langage, ainsi qu'un algorithme de typage avec inférence des effets. Dans notre cas, ce système est tout de même relativement restreint, vu notre objectif de traduction fonctionnelle et plus tard de preuve de correction. En particulier, nous ne traiterons pas de polymorphisme d'effets — possibilité d'abstraire le type d'une fonction par rapport à un effet — ce qui simplifie le système de types et les règles de typage. Notre contribution ne réside donc pas ici dans l'analyse des effets, qui s'inspire de travaux plus généraux, mais plutôt dans l'utilisation que nous en faisons par la suite pour traduire les programmes impératifs en programmes fonctionnels.

Le premier paragraphe concerne directement le langage que nous avons introduit, et s'attache donc à l'étude des effets en présence de références uniquement. Mais nous étudions également dans le second paragraphe l'adjonction d'exceptions à notre langage. Ceci nous permet de dégager, dans un troisième paragraphe, une certaine généralité dans notre propos et de définir une notion abstraite d'effet et le système de typage qui lui est associé.

1.2.1 Références

Dans cette section, nous nous concentrons sur l'analyse des effets pour le langage de programmation que nous venons de définir. Il s'agit plus précisément de déterminer les manipulations effectuées sur les références — à savoir création, lecture et modification — dans chaque partie du programme, pour pouvoir plus tard traduire celui-ci en un programme fonctionnel.

En effet, l'idée de la traduction fonctionnelle que nous proposons est la suivante : une expression de programme accédant au contenu d'une référence r devient un terme fonctionnel prenant en argument la *valeur* de cette référence, et une expression la modifiant devient un terme fonctionnel retournant, éventuellement parmi d'autres valeurs, la *nouvelle valeur* de cette référence. Cette idée est très naturelle : elle correspond à ce que l'on fait lorsque l'on « fonctionnalise » un programme impératif à la main. Si l'on considère par exemple le programme suivant qui ajoute à la référence s la somme des i premiers entiers :

$$\begin{array}{l} \textsf{while } !i > 0 \textsf{ do} \\ \quad s := !s + !i; \\ \quad i := !i - 1 \\ \textsf{done} \end{array}$$

sa traduction fonctionnelle est un programme prenant en argument les valeurs des variables

i et s et retournant leurs nouvelles valeurs. Très exactement, il s'agit du programme suivant :

$$\begin{array}{l}\mathsf{let\ rec}\ w\ i\ s\ =\\ \quad \mathsf{if}\ i > 0\ \mathsf{then}\\ \quad\quad w\ (i-1)\ (s+i)\\ \quad \mathsf{else}\\ \quad\quad (i,s)\end{array}$$

qui est celui que l'on aurait écrit spontanément dans un langage purement fonctionnel — mis à part le fait que l'on ne prendrait pas la peine de retourner la valeur de i, qui est nulle à la sortie de la boucle. Dans cet exemple, les effets du programme sont un accès en lecture et écriture aux variables i et s. C'est cette information qui permet en particulier de déduire le type de la traduction fonctionnelle. Si par exemple le programme ci-dessus ne modifiait pas la valeur de s mais se contentait de la lire alors la traduction fonctionnelle aurait été une fonction prenant en arguments les valeurs de i et s mais ne retournant que la valeur de i.

Un effet va donc être composé de deux ensemble de références, le premier représentant les références potentiellement *lues* par l'expression de programme et le second les références potentiellement *modifiées* par celle-ci. Mais il faut imposer sur ces deux ensembles de variables une condition supplémentaire. En effet, si l'on considère par exemple l'expression conditionnelle

$$\mathsf{if}\ !i > 0\ \mathsf{then}\ s\ :=\ 0\ \mathsf{else\ void}$$

alors une première analyse indique que la variable i est accédée en lecture et la variable s en écriture seulement. Mais si l'on doit écrire la traduction fonctionnelle d'une telle expression, on devra écrire quelque chose comme

$$\lambda i.\lambda s.\mathsf{if}\ i > 0\ \mathsf{then}\ 0\ \mathsf{else}\ s$$

c'est-à-dire une fonction prenant en argument la valeur courante de s pour être en mesure de la retourner lorsque la deuxième branche du test est sélectionnée. De manière générale, il faudrait incorporer dans les variables d'entrée d'une conditionnelle les variables de sortie de chaque branche n'apparaissant pas dans les variables de sortie de l'autre branche. Une solution plus simple consiste à imposer à l'ensemble de variables de sortie d'être toujours contenu dans l'ensemble de variables d'entrée. D'où finalement la définition suivante :

Définition 8 (effets) *Un effet est un couple $\epsilon = (\rho, \omega)$ où ρ et ω sont deux ensembles finis de variables tels que $\omega \subseteq \rho$. On notera $\bot$ l'effet vide, c'est-à-dire le couple $(\emptyset, \emptyset)$.*

L'opération la plus courante sur les effets est leur *union*. Elle intervient naturellement dès que plusieurs expressions de programme sont séquencées, comme les expressions $s\ := !s + !i$ et $i\ := !i - 1$ dans l'exemple ci-dessus. On introduit cette opération sur les effets de la manière suivante :

Définition 9 (union d'effets) *L'opération binaire $\sqcup$ sur les effets est définie de la manière suivante :*

$$(\rho_1, \omega_1) \sqcup (\rho_2, \omega_2) \quad \stackrel{def}{=} \quad (\rho_1 \cup \rho_2, \omega_1 \cup \omega_2)$$

De la même manière que le typage classique est une annotation de chaque expression dans un programme par un type, nous allons annoter chaque expression d'un programme par un effet. Il est pour cela naturel d'étendre le système de types pour y incorporer les effets, et d'étendre en conséquence les règles de typage. La raison en apparaîtra clairement dans la section 1.3 concernant la traduction fonctionnelle elle-même. Pour étendre le système de types, nous devons distinguer la notion de *valeur* de la notion de *calcul*. En effet, si ces deux notions peuvent être confondues dans un langage purement fonctionnel — toute considération d'efficacité étant écartée —, il n'en est plus de même dans un langage comportant des traits impératifs. L'expression $e \equiv x\ :=\ 0\ ;\ 2$ s'évalue en la valeur 2, certes, mais on ne peut pas substituer 2 à e dans n'importe quel contexte, en raison de l'effet de bord résultant de l'évaluation de e. Pour cette raison, le nouveau système de type doit permettre de faire apparaître un effet dans le type de e, mais pas dans le type de sa valeur, à savoir int. De plus, une abstraction associe à une valeur un calcul, correspondant à l'évaluation du corps de l'abstraction. Enfin, l'abstraction elle-même est considérée comme une valeur.

Ces considérations nous amènent à définir le nouveau système de types de la manière suivante :

Définition 10 (types avec effets) *On définit mutuellement récursivement les systèmes de* types de valeurs *et de* types de calculs *de la manière suivante :*

$$\left\{\begin{array}{lll} \textit{types de valeurs} & \tau & ::= \quad (\tau,\ldots,\tau)c \mid \alpha \mid (x:\tau)\rightarrow\kappa \mid \forall\alpha.\tau \mid \tau\ \mathsf{ref} \\ \textit{types de calculs} & \kappa & ::= \quad (\tau,\epsilon) \end{array}\right.$$

On acceptera toujours la notation $\tau \rightarrow \kappa$ lorsque l'argument de la fonction n'est pas une référence et la notation $\tau_1 \rightarrow^{\epsilon} \tau_2$ sera un raccourci pour $\tau_1 \rightarrow (\tau_2, \epsilon)$. La définition de la substitution $\tau[\alpha \leftarrow \tau_0]$ s'étend de manière évidente.

Reste maintenant à définir un système de règles d'inférences pour établir le type de calcul d'un programme donné, dans un environnement donné.

La syntaxe de notre langage, donnée figure 1.2, n'autorise les lectures et écritures que sur les références explicitement nommées, c'est-à-dire qu'elle restreint les constructions $!x$ et $x\ :=\ e$ au cas où x est une *variable*, à la différence des implantations de ML qui autorisent ces constructions à porter sur n'importe quelle expression de type τ ref. Ce choix de notre part vise précisément à simplifier l'analyse des effets — mais il ne limite en rien l'expressivité du langage puisque l'on peut toujours réécrire $!e$ en $\mathsf{let}\ r = e\ \mathsf{in}\ !r$ et $e\ :=\ e'$ en $\mathsf{let}\ r = e\ \mathsf{in}\ r\ :=\ e'$. L'analyse des effets semble alors un problème simple, où chaque expression $!x$ donne un effet $(\{x\}, \emptyset)$ et chaque expression $x\ :=\ e$ un effet $(\{x\} \cup \rho, \{x\} \cup \omega)$, où (ρ, ω) est l'effet de l'expression e, les effets se composant ensuite de façon triviale à travers l'opération $\sqcup$. C'est oublier un peu vite le problème des alias.

Le problème des alias. On dit que l'on a un *alias*[8] lorsque deux variables x et y représentent en fait une même référence, c'est-à-dire une même donnée modifiable en place. On dit alors que x et y sont *physiquement* identiques. De façon observationnelle, toute modification

[8] On parle en anglais de phénomène d'*aliasing*.

de x affecte alors y, et réciproquement. Il est clair que ce phénomène aurait une incidence sur la traduction fonctionnelle de programmes, car un simple programme comme

$$x := !x + 1 \ ; \ y := !y + 1$$

ne se traduira certainement pas de la même façon si x et y représentent la même référence ou non. Dans le premier cas, la traduction serait la fonction $\lambda x.x + 2$ et dans le deuxième la fonction $\lambda(x, y).(x + 1, y + 1)$.

Une telle situation, où deux variables x et y représentent la même référence, n'a qu'une origine possible : une liaison de variable dans le programme a associé à x la valeur de y ou réciproquement. Cette liaison peut résulter soit de la construction let in, soit de l'application de fonction. Ainsi, dans un environnement où x est une référence sur un entier, le programme suivant :

$$\mathsf{let}\ y = x\ \mathsf{in}\ x := !x + 1 \ ; \ y := !y + 1$$

a pour effet d'incrémenter deux fois la référence x, mais il en est également de même du programme suivant :

$$((\mathsf{fun}\ (y : \mathsf{int\ ref}) \rightarrow x := !x + 1 \ ; \ y := !y + 1) \ \ x)$$

Le problème des alias a été largement étudié, essentiellement dans le cadre des *tableaux*. En effet, les indices des tableaux étant des grandeurs numériques, ils résultent rapidement de calculs dont l'indécidabilité ne permet pas de déterminer si dans un programme les expressions $t[i]$ et $t[j]$ représentent ou non le même élément du tableau t, ce qui entrave la possibilité d'établir statiquement des propriétés du programme.

En ce qui concerne les références, la détermination statique des alias est également un problème indécidable. Il existe cependant des méthodes statiques pour déterminer des ensembles de références pouvant être des alias. C'est par exemple la notion de *région* dans l'article de J.-P. Talpin et P. Jouvelot [55]. Considérer de telles méthodes et les appliquer à la traduction fonctionnelle s'avérerait très complexe, pour un faible gain. D'autre part, les alias sont source d'erreurs subtiles dans les programmes, quasi indétectables. Nous avons donc décidé d'*exclure* les alias de nos programmes. Une première solution, très simple, consiste à limiter la manipulation des références dans les programmes, à la manière de certains langages de programmation.

Typage des programmes « Pascal »

Nous pouvons nous inspirer du langage Pascal pour parvenir à un fragment de notre langage pour lequel l'inférence des effets sera décidable. Contrairement à ML, les références dans le langage Pascal, que l'on appelle des *variables*, ne peuvent être introduites que *statiquement*, par la construction var, qu'elles soient locales ou globales. En d'autres termes, il n'y a pas de construction ref qui permette de créer *dynamiquement* de nouvelle référence — si l'on s'interdit l'usage explicite des pointeurs, bien entendu. Cela s'explique en termes de compilation : n'avoir que des déclarations statiques de variables permet d'allouer statiquement les variables globales et dynamiquement sur la pile les variables locales, alors que la création dynamique de nouvelles variables oblige à faire une allocation globale dynamiquement, ce qui est complexe en l'absence de GC (car il faut être en mesure de libérer plus tard cet espace mémoire).

Plutôt que de supprimer la construction ref, qui nous sera utile par la suite, nous allons nous contenter de restreindre par le typage l'application de certaines constructions. Les deux principales modifications sont les suivantes :

- dans une construction $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$, avec e_1 de type $\tau\ \mathsf{ref}$, e_1 doit être de la forme $\mathsf{ref}\ e$ i.e. une nouvelle référence. Une telle construction $\mathsf{let}\ x = \mathsf{ref}\ e\ \mathsf{in}\ e_2$ peut être comparée à l'introduction d'une variable locale en Pascal, à la différence près qu'elle est ici nécessairement initialisée (à la valeur de e).
- dans une application $(f\ e)$, avec e type $\tau\ \mathsf{ref}$, e doit être une variable.

La première restriction permet d'exclure les alias du type $\mathsf{let}\ y = x\ \mathsf{in}\ \ldots$ où x est une référence. La seconde permet l'analyse des effets lors de l'application d'une fonction. Elle n'exclut pas *a priori* la création d'un alias lors de cette application ; cette exclusion-là sera une conséquence de la règle de typage. On peut désormais définir les règles d'inférence de typage avec effets.

Les environnements de typage sont toujours définis de la même manière, comme une liste finie de variables de types et de liaisons $x : \tau$, mais où τ désigne maintenant un type de valeur. Comme précédemment, on commence par définir la notion de types bien formés, qui est maintenant un double jugement sur les types de valeurs et les types de calculs.

Définition 11 (types avec effets bien formés) *Les jugements* $\Gamma \vdash_r \tau\ \mathsf{bf}$ *et* $\Gamma \vdash_r \kappa\ \mathsf{bf}$ *sont définis inductivement sur la structure de* τ *et* κ *par l'ensemble des règles d'inférence suivantes :*

$$\frac{\forall i \in 1\ldots n \quad \Gamma \vdash_r \tau_i\ \mathsf{bf}}{\Gamma \vdash_r (\tau_1, \ldots, \tau_n)c\ \mathsf{bf}} \qquad \frac{\Gamma \vdash_r \tau_1\ \mathsf{bf} \qquad \Gamma, x : \tau_1 \vdash_r \kappa_2\ \mathsf{bf}}{\Gamma \vdash_r (x : \tau_1) \to \kappa_2\ \mathsf{bf}}$$

$$\frac{\alpha \in \Gamma}{\Gamma \vdash_r \alpha\ \mathsf{bf}} \qquad \frac{\Gamma, \alpha \vdash_r \tau\ \mathsf{bf}}{\Gamma \vdash_r \forall\alpha.\tau\ \mathsf{bf}} \qquad \frac{\Gamma \vdash_r \tau\ \mathsf{bf}}{\Gamma \vdash_r \tau\ \mathsf{ref}\ \mathsf{bf}}$$

$$\frac{\Gamma \vdash_r \tau\ \mathsf{bf} \qquad \omega \subseteq \rho \subseteq \mathit{Refs}(\Gamma)}{\Gamma \vdash_r (\tau, (\rho, \omega))\ \mathsf{bf}}$$

La définition de $\Gamma\ \mathsf{bf}$ *reste la même.*

On a également besoin de définir la substitution d'une référence par une autre dans un type avec effets, que ce soit un type de valeur ou un type de calcul, que l'on notera $\tau[x \leftarrow y]$ ou $\kappa[x \leftarrow y]$. Cette substitution n'a lieu qu'à l'intérieur des effets.

Définition 12 (substitutions) *Les substitutions* $\tau[x \leftarrow y]$ *et* $\kappa[x \leftarrow y]$ *sont définies mu-*

tuellement récursivement de la manière suivante :

$$
\begin{array}{llcll}
\textit{valeurs} & ((\tau_1,\dots,\tau_n)c)[x \leftarrow y] & = & (\tau_1[x \leftarrow y],\dots,\tau_n[x \leftarrow y])c & \\
& \alpha[x \leftarrow y] & = & \alpha & \\
& ((x:\tau) \rightarrow \kappa)[x \leftarrow y] & = & (x:\tau[x \leftarrow y]) \rightarrow \kappa & \\
& ((z:\tau) \rightarrow \kappa)[x \leftarrow y] & = & (z:\tau[x \leftarrow y]) \rightarrow (\kappa[x \leftarrow y]) & z \neq x \\
& (\forall\alpha.\tau)[x \leftarrow y] & = & \forall\alpha.(\tau[x \leftarrow y]) & \\
& (\tau\ \mathsf{ref})[x \leftarrow y] & = & \tau[x \leftarrow y]\ \mathsf{ref} & \\
\\
\textit{calculs} & (\tau,\epsilon)[x \leftarrow y] & = & (\tau[x \leftarrow y],\epsilon[x \leftarrow y]) & \\
\\
\textit{effets} & (\rho,\omega)[x \leftarrow y] & = & (\rho[x \leftarrow y],\omega[x \leftarrow y]) &
\end{array}
$$

Tout comme pour la substitution dans les types, nous supposons ici implicitement un renommage préalable des variables liées de τ et de κ afin d'éviter la capture de variable.

Il nous faut aussi définir la *suppression* d'une référence dans un effet, qui n'est que la différence ensembliste.

Définition 13 (suppression) *La suppression d'une référence x dans un effet $\epsilon = (\rho,\omega)$, notée $\epsilon\backslash x$, est définie par*

$$\epsilon\backslash x \quad = \quad (\rho\backslash\{x\},\omega\backslash\{x\})$$

Enfin nous définissons la notion d'occurrence d'une référence dans un effet, un type de valeur ou un type de calcul.

Définition 14 (occurrence) *Les prédicats $x \in \epsilon$, $x \in \tau$ et $x \in \kappa$ sont définis mutuellement récursivement de la manière suivante :*

$$
\begin{array}{llcl}
\textit{valeurs} & x \in (\tau_1,\dots,\tau_n)c & = & x \in \tau_1 \vee \dots \vee x \in \tau_n \\
& x \in \alpha & = & \mathsf{faux} \\
& x \in (z:\tau) \rightarrow \kappa & = & x \in \tau \vee (x \neq z \wedge x \in \kappa) \\
& x \in \forall\alpha.\tau & = & x \in \tau \\
& x \in \tau\ \mathsf{ref} & = & x \in \tau \\
\\
\textit{calculs} & x \in (\tau,\epsilon) & = & x \in \tau \vee x \in \epsilon \\
\\
\textit{effets} & x \in (\rho,\omega) & = & x \in \rho \cup \omega
\end{array}
$$

En d'autres termes, $x \in \tau$ peut s'interpréter comme « x appartient aux effets libres *dans le type τ ».*

On introduit alors le jugement « le programme e a le type de calcul κ dans l'environnement Γ », que l'on notera $\Gamma \vdash_r e : \kappa$.

Définition 15 (typage avec effets) *Le jugement $\Gamma \vdash_r e : \kappa$ est défini inductivement sur la structure de e par l'ensemble des règles d'inférence données figure 1.4.*

$$\frac{\mathit{Type}(c) = \tau}{\Gamma \vdash_r c : (\tau, \bot)}(\mathsf{CONST}_r) \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash_r x : (\tau, \bot)}(\mathsf{VAR}_r)$$

$$\frac{\Gamma, x : \tau \vdash_r e : \kappa \qquad \Gamma \vdash_r \tau \ \mathsf{bf}}{\Gamma \vdash_r \mathsf{fun}\ (x : \tau) \to e : ((x : \tau) \to \kappa, \bot)}(\mathsf{FUN}_r) \qquad \frac{\Gamma, \alpha \vdash_r e : (\tau, \epsilon) \qquad \alpha \notin \Gamma}{\Gamma \vdash_r \mathsf{Fun}\ \alpha \to e : (\forall \alpha.\tau, \epsilon)}(\mathsf{FUNT}_r)$$

$$\frac{\Gamma \vdash_r e : (\tau_1 \to^{\epsilon} \tau, \epsilon_2) \qquad \Gamma \vdash_r e_1 : (\tau_1, \epsilon_1) \qquad \tau_1 \neq \_\ \mathsf{ref}}{\Gamma \vdash_r (e\ e_1) : (\tau, \epsilon_2 \sqcup \epsilon_1 \sqcup \epsilon)}(\mathsf{APP}_r)$$

$$\frac{\Gamma \vdash_r e : ((x : \tau_1\ \mathsf{ref}) \to^{\epsilon} \tau, \epsilon_2) \qquad r : \tau_1\ \mathsf{ref} \in \Gamma \qquad r \notin (\tau, \epsilon)}{\Gamma \vdash_r (e\ r) : (\tau[x \leftarrow r], \epsilon_2 \sqcup \epsilon[x \leftarrow r])}(\mathsf{APPREF}_r)$$

$$\frac{\Gamma \vdash_r e : (\forall \alpha.\tau, \epsilon) \qquad \Gamma \vdash_r \tau_1\ \mathsf{bf}}{\Gamma \vdash_r (e\ \tau_1) : (\tau[\alpha \leftarrow \tau_1], \epsilon)}(\mathsf{APPT}_r)$$

$$\frac{\tau = \tau_1 \to \kappa_2 \qquad \Gamma \vdash_r \tau\ \mathsf{bf} \qquad \Gamma, f : \tau \vdash_r e : (\tau, \bot)}{\Gamma \vdash_r \mathsf{rec}\ f\ : \tau = e : (\tau, \bot)}(\mathsf{REC}_r)$$

$$\frac{\Gamma \vdash_r e_1 : (\mathsf{bool}, \epsilon_1) \qquad \Gamma \vdash_r e_2 : (\tau, \epsilon_2) \qquad \Gamma \vdash_r e_3 : (\tau, \epsilon_3)}{\Gamma \vdash_r \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : (\tau, \epsilon_1 \sqcup \epsilon_2 \sqcup \epsilon_3)}(\mathsf{COND}_r)$$

$$\frac{\Gamma \vdash_r e_1 : (\tau_1, \epsilon_1) \qquad \tau_1 \neq \_\ \mathsf{ref} \qquad \Gamma, x : \tau_1 \vdash_r e_2 : (\tau, \epsilon_2)}{\Gamma \vdash_r \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : (\tau, \epsilon_1 \sqcup \epsilon_2)}(\mathsf{LET}_r)$$

$$\frac{\Gamma \vdash_r e_1 : (\tau_1, \epsilon_1) \qquad \Gamma, x : \tau_1\ \mathsf{ref} \vdash_r e_2 : (\tau, \epsilon_2) \qquad x \notin \tau}{\Gamma \vdash_r \mathsf{let}\ x = \mathsf{ref}\ e_1\ \mathsf{in}\ e_2 : (\tau, \epsilon_1 \sqcup \epsilon_2 \backslash x)}(\mathsf{LETREF}_r)$$

$$\frac{\Gamma \vdash_r e_1 : (\mathsf{unit}, \epsilon_1) \qquad \Gamma \vdash_r e_2 : (\tau, \epsilon_2)}{\Gamma \vdash_r e_1\ ;\ e_2 : (\tau, \epsilon_1 \sqcup \epsilon_2)}(\mathsf{SEQ}_r)$$

$$\frac{\Gamma \vdash_r e_1 : (\mathsf{bool}, \epsilon_1) \qquad \Gamma \vdash_r e_2 : (\mathsf{unit}, \epsilon_2)}{\Gamma \vdash_r \mathsf{while}\ e_1\ \mathsf{do}\ e_2\ \mathsf{done} : (\mathsf{unit}, \epsilon_1 \sqcup \epsilon_2)}(\mathsf{LOOP}_r)$$

$$\frac{\Gamma \vdash_r e : (\tau, \epsilon) \qquad \tau\ \mathsf{pur}}{\Gamma \vdash_r \mathsf{ref}\ e : (\tau\ \mathsf{ref}, \epsilon)}(\mathsf{REF}_r) \qquad \frac{x : \tau\ \mathsf{ref} \in \Gamma}{\Gamma \vdash_r !x : (\tau, (\{x\}, \emptyset))}(\mathsf{DEREF}_r)$$

$$\frac{x : \tau\ \mathsf{ref} \in \Gamma \qquad \Gamma \vdash_r e : (\tau, (\rho, \omega))}{\Gamma \vdash_r x := e : (\mathsf{unit}, (\{x\} \cup \rho, \{x\} \cup \omega))}(\mathsf{AFF}_r)$$

FIG. 1.4: Typage avec effets des programmes « Pascal »

Ces règles de typage nécessitent plusieurs commentaires. Tout d'abord les règles concernant les références, à savoir les règles (DEREF_r) et (AFF_r), traduisent directement le sens de l'effet, en ajoutant les références lues dans la première composante de l'effet, et les références modifiées dans sa seconde composante. On constate ensuite que les variables et les abstractions n'ont pas d'effet : cela est cohérent avec le choix d'une sémantique d'appel par valeur. Si l'on regarde plus particulièrement la règle (FUN_r) on peut voir que l'effet du corps d'une abstraction est « conservé » dans le type κ, grâce auquel il pourra être effectif lors d'une application ultérieure de l'abstraction.

Les règles les plus importantes concernent l'application et la construction let. Ce sont en effet ces constructions qui *lient* une variable à une valeur, et elles se dédoublent suivant que cette valeur est ou non une référence.

En ce qui concerne l'application, la règle (APP_r) traite du cas où l'argument n'est pas une référence. Dans ce cas, le type du résultat est donné par celui de la fonction, comme dans le typage classique, et l'effet est la composée de trois effets : celui résultant de l'évaluation de la fonction (ϵ_2), celui résultant de l'évaluation de l'argument (ϵ_1) et enfin celui résultant de l'application elle-même (ϵ). La règle (APPREF_r), quant à elle, traite du cas de l'application d'une fonction à une référence. Comme nous l'avons expliqué, cette référence est nécessairement une variable, appelée ici r. La prémisse $r \notin (\tau, \epsilon)$ exclut la création d'un alias par la liaison de l'argument formel x à l'argument réel r (voir page 23). Une fois la fonction évaluée, avec un effet ϵ_2, l'argument réel r est substitué à l'argument formel x, *dans l'effet* ϵ *comme dans le type* τ. En effet, le résultat de l'application peut être à son tour une fonction dont les effets ultérieurs affecteront la référence r, comme c'est le cas par exemple pour la fonction curryfiée incrémentant une référence d'une valeur donnée :

$$\mathsf{fun}\ (x : \mathsf{int\ ref}) \rightarrow \mathsf{fun}\ (y : \mathsf{int}) \rightarrow x\ := !x + y$$

dont le type avec effets est le suivant :

$$(x : \mathsf{int\ ref}) \rightarrow (y : \mathsf{int}) \rightarrow^{(\{x\},\{x\})} \mathsf{unit}$$

L'application de cette fonction à une référence r donne une fonction incrémentant r de la valeur de son argument, c'est-à-dire une fonction de type $(y : \mathsf{int}) \rightarrow^{(\{r\},\{r\})} \mathsf{unit}$.

La règle concernant la construction let a également été dédoublée. On distingue en effet le cas général, c'est-à-dire la liaison du résultat d'une évaluation dans une expression, du cas particulier de l'introduction d'une nouvelle référence. La règle générale (LET_r) ne nécessite pas d'explication particulière, et peut être rapprochée de la règle (SEQ_r), ce qui n'a rien d'étonnant étant donné que la construction let est à la programmation fonctionnelle ce que la séquence est à la programmation impérative. La règle (LETREF_r), quant à elle, traite de la construction $\mathsf{let}\ x = \mathsf{ref}\ e_1\ \mathsf{in}\ e_2$, c'est-à-dire de l'introduction d'une nouvelle référence, initialisée à la valeur de e_1 et locale à une expression e_2. Si τ et ϵ_2 désignent respectivement le type et l'effet de l'expression e_2, le caractère local de x se traduit par la suppression, dans l'effet de l'expression toute entière, de toute occurrence de x. Ainsi l'effet résultant est-il $\epsilon_1 \sqcup \epsilon_2 \backslash x$, où ϵ_1 désigne l'effet de l'expression e_1. Mais x pourrait également apparaître dans τ si e_2 est une fonction ayant un effet sur la référence x, auquel cas x *échapperait* à sa portée qui ne doit pas excéder e_2. Ce serait par exemple le cas pour les fameuses fonctions *gensym*[9]

[9]pour *gén*érateur de *sym*boles.

que l'on peut définir de la façon suivante

$$\mathsf{let}\ x = \mathsf{ref}\ 0\ \mathsf{in}\ \mathsf{fun}\ (u : \mathsf{unit}) \to (x\ := !x + 1\ ;\ !x)$$

Nous expliquerons dans la section suivante consacrée à la traduction pourquoi nous ne sommes pas à même de traiter ce genre de construction, et cela justifie la présence de la troisième prémisse $x \notin \tau$.

On peut noter toute de même qu'une telle définition est impossible en Pascal également, mais qu'une solution consiste à déclarer le compteur globalement. Une telle solution est également envisageable dans le cas de notre langage. Le seul problème devient alors un problème de *visibilité* car le compteur peut désormais être modifié par n'importe quelle partie du programme. Pour remédier à cela, certaines implantations de Pascal proposent, comme d'autres langages, une directive static pouvant préfixer une déclaration de variable locale, et dont l'effet est d'en faire en fait une variable globale, uniquement visible dans la fonction où elle est déclarée. Dans un langage avec modules, une autre solution serait de déclarer le compteur globalement à l'intérieur du module, mais de ne pas l'exporter i.e. de ne pas le rendre visible à l'extérieur du module.

Il convient de faire une remarque concernant la règle (REC_r). La syntaxe que nous avons introduite pour les fonctions récursives, à savoir $\mathsf{rec}\ f\ : \tau = e$, impose de déclarer le type τ de la fonction f. Cela se justifie car nous ne souhaitons pas faire de l'inférence de type, comme nous l'avons déjà expliqué. Mais telle qu'est écrite la règle (REC_r), il apparaît qu'il faut déclarer pour f un type $\tau = \tau_1 \to \kappa_2$, c'est-à-dire où les effets du corps de la fonction sont spécifiés dans le type κ_2. On préférerait en pratique n'avoir qu'à donner le type de f sans ses effets, et que ceux-ci soient inférés, comme c'est le cas pour les autres constructions du langage. La règle serait alors de la forme :

$$\frac{\Gamma \vdash_r \tau\ \mathsf{bf} \qquad \Gamma, f : \tau \vdash_r\ e : (\tau, \bot)}{\Gamma\ \vdash_r\ \mathsf{rec}\ f\ \ : \tau_0 = e : (\tau, \bot)}(\mathsf{REC'}_r)$$

où le type τ_0 est un type sans aucun effet — où tous les effets valent $\bot$, si l'on préfère — uniquement déclaré pour ne pas avoir à faire d'inférence de type, mais seulement de l'inférence des effets. Le tout est alors de savoir si on a toujours avec la règle ($\mathsf{REC'}_r$) un typage des effets décidable et un type unique. Il s'avère que l'on peut répondre par l'affirmative à cette question, et c'est donc une règle similaire à la règle ci-dessus que nous avons adoptée en pratique. Les détails concernant l'inférence des effets dans le cas des fonctions récursives sont données dans le chapitre 3 page 117.

Enfin, remarquons que nous avons conservé la règle (REF_r) mais qu'elle n'est d'aucune utilité puisque les expressions de la forme ref _ ne peuvent être liées à une variable qu'à travers la construction $\mathsf{let}\ x = \mathsf{ref}\ \_\ \mathsf{in}\ \_$ pour laquelle la règle particulière (LETREF_r) s'applique.

Nous avons donc établi un typage avec effets pour un fragment important de notre langage qui inclut complètement le langage FORTRAN ou encore le langage Pascal sans pointeurs. Mais nous allons montrer maintenant que nous pouvons traiter un fragment encore plus important, grâce à un typage plus fin des références.

Typage des effets amélioré

Ce que nous avons obtenu jusqu'à présent n'est pas encore entièrement satisfaisant, et ce pour plusieurs raisons. D'une part, on note une dissymétrie entre les deux programmes

$$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \qquad \text{et} \qquad ((\mathsf{fun}\ x \to e_2)\ e_1)$$

lorsque e_1 est une référence : dans le premier cas e_1 doit être une nouvelle référence introduite par la construction ref et dans le second cas e_1 doit être au contraire une variable. Cependant, on a toujours la possibilité de se restreindre à ces cas-là en réécrivant ses programmes grâce aux équivalences suivantes :

$$(f\ (\mathsf{ref}\ e)) \approx \mathsf{let}\ r = \mathsf{ref}\ e\ \mathsf{in}\ (f\ r) \qquad \text{et} \qquad \mathsf{let}\ x = y\ \mathsf{in}\ e[x] \approx e[y]$$

D'autre part, et c'est là un point bien plus important, on souhaite pouvoir écrire des fonctions *retournant des références.* En effet, si l'on envisage d'implanter un type de donnée impératif abstrait par un référence, on a généralement une fonction de création renvoyant un nouvel objet dans ce type, en l'occurrence une nouvelle référence. Or, s'il est actuellement possible d'écrire une telle fonction, il n'est pas possible de l'utiliser, car une expression telle que

$$\mathsf{let}\ r = (f\ v)\ \mathsf{in}\ \ldots$$

avec $(f\ v)$ de type τ ref n'est pas typable. Et on ne peut s'autoriser à inspecter le code de f pour vérifier s'il s'agit ou non d'une instance de ref, car ce serait briser la modularité.

Pour remédier à ce dernier problème, nous proposons la solution suivante : interpréter toute expression de type τ ref comme une référence nouvellement créée. Dès lors, on peut généraliser la règle (LETREF_r) de manière à accepter la liaison à une variable d'une expression quelconque de type τ_1 ref, et non plus seulement d'une expression de la forme ref e_1. En particulier, on pourra définir des fonctions retournant de nouvelles références et lier le résultat de ces fonctions à une variable locale par une construction let. En contrepartie, il ne faut plus accepter d'expression de type τ ref qui ne corresponde pas à une nouvelle référence. Pour cela, il suffit de ne pas accepter dans le programme d'expression réduite à une variable x où x est une référence, sauf bien entendu dans l'application d'une fonction à une référence, cas déjà traité par la règle (APPREF_r).

Cela nous conduit à introduire deux variantes pour les règles (VAR_r) et (LETREF_r).

Définition 16 (typage amélioré) *Le jugement $\Gamma \vdash_r e : \kappa$ est modifié par les variantes des règles* (VAR_r) *et* (LETREF_r) *données figure 1.5.*

Désormais, la création d'un alias par une liaison let in (voir page 23) n'est plus possible. En effet, la règle de typage du let autorise *a priori* la construction

$$\mathsf{let}\ y = x\ \mathsf{in}\ x := !x + 1\ ;\ y := !y + 1$$

mais la règle ($\mathsf{VAR'}_r$) exclut l'usage de la variable de référence x dans l'expression let $y = x$. Les seules manipulations possibles d'une variable de référence r sont donc les constructions $!r$, $r := e$ et $(f\ r)$, et sa création à travers une construction let $r = e_1$ in e_2. La création d'un alias par appel de fonction était quant à elle déjà exclue par la règle (APPREF_r).

Il existe un algorithme d'inférence des types avec effets, ce que l'on peut formuler de la manière suivante :

$$\frac{x : \tau \in \Gamma \qquad \tau \neq \_\ \mathsf{ref}}{\Gamma \vdash_r x : (\tau, \bot)}(\mathsf{VAR'}_r)$$

$$\frac{\Gamma \vdash_r e_1 : (\tau_1\ \mathsf{ref}, \epsilon_1) \qquad \Gamma, x : \tau_1\ \mathsf{ref} \vdash_r e_2 : (\tau, \epsilon) \qquad x \notin \tau}{\Gamma \vdash_r \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : (\tau, \epsilon_1 \sqcup \epsilon \backslash x)}(\mathsf{LETREF'}_r)$$

FIG. 1.5: Typage avec effets amélioré

Proposition 1 (algorithme d'inférence des effets) *Il existe un algorithme qui, étant donné un environnment Γ bien formé et un programme e, termine et, soit indique que e n'est pas typable dans Γ, soit retourne un type τ tel que $\Gamma \vdash_r e : \tau$, le type τ étant alors unique (au renommage près des variables de types liées).*

PREUVE. L'algorithme procède par récurrence sur la structure de e. Pour chaque construction, il existe une et une seule règle pouvant être appliquée. Si l'une des prémisses de celle-ci n'est pas vérifiée, alors l'algorithme échoue et le programme e n'est pas typable dans Γ. Sinon, le type τ est construit de manière unique en fonction de la règle appliquée et des types retournés par les appels récursifs, uniques par hypothèse de récurrence. La terminaison de l'algorithme de typage est justifiée par le fait que la taille du programme décroît strictement à chaque appel récursif. □

Exemples. Nous allons introduire ici quelques exemples de programmes — très simples — pour lesquels nous allons donner le résultat de l'inférence de type avec effets. Ces mêmes programmes seront repris plus tard lorsque nous introduirons la traduction fonctionnelle (section 1.3.2) puis lorsque nous introduirons les annotations et les preuves de programmes (chapitre 2).

Exemple 1 *Dans un contexte Γ contenant une référence entière x, on considère le premier exemple suivant :*

$$ex_1 \quad \equiv \quad x := !x + 1$$

Grâce aux trois règles (APP_r), (CONST_r) *et* (DEREF_r) *on type l'expression* $!x + 1$ *de la façon suivante :*

$$\Gamma \vdash_r !x + 1 : (\mathsf{int}, (\{x\}, \emptyset))$$

Dès lors, par application de la règle (AFF_r) *on a*

$$\Gamma \vdash_r ex_1 : (\mathsf{unit}, (\{x\}, \{x\}))$$

□

Exemple 2 *Toujours dans le même contexte Γ on considère la fonction suivante :*

$$ex_2 \quad \equiv \quad \mathsf{fun}\ (r : \mathsf{int\ ref}) \rightarrow r := !r + 1\ ;\ !r$$

et on a

$$\Gamma \vdash_r ex_2 : ((r : \mathsf{int\ ref}) \rightarrow^{(\{r\},\{r\})} \mathsf{int}, \bot)$$

□

Exemple 3 *En supposant de plus que* Γ *contient une autre référence entière* y*, on peut considérer le programme suivant :*

$$ex_3 \quad \equiv \quad y := (ex_2\ x)$$

qui a le type

$$\Gamma \vdash_r ex_3 : (\mathsf{unit}, (\{x, y\}, \{x, y\}))$$

□

Nous pouvons dès à présent justifier partiellement certains des choix que nous avons fait dans notre système de typage des effets en montrant quelques propriétés de ce système. En particulier, nous pouvons montrer qu'une expression de type τ ref qui n'est pas une variable s'évalue nécessairement en une référence « neuve ». Cette propriété traduit l'absence d'alias dans nos programmes, que nous nous sommes efforcés d'obtenir, et se formalise par le lemme suivant.

Lemme 3 *Soit* Γ *un environnement bien formé et* e *un programme tel que* $\Gamma \vdash_r e : (\tau\ \mathsf{ref}, \_)$*. Soit* θ *un état associant des valeurs aux références de* Γ*. Alors si* e *s'évalue en une valeur* v *dans un état* θ'*, c'est-à-dire*

$$\theta.e \longmapsto^\star \theta'.v$$

cette valeur v *est nécessairement une variable, soit* x*, et l'on a*

$$x \notin dom(\theta)$$

Preuve. Le fait que v soit nécessairement une variable est immédiat, car toute autre valeur serait fonctionnelle, ce qui contredirait $e : \_$ ref et l'autoréduction, établie dans [59]. (On exclut le fait qu'une constante puisse être une référence, hypothèse plus que raisonnable.) La preuve se fait par récurrence sur la longueur de la dérivation $\longmapsto^\star$ et par cas sur la structure de e. L'expression e ne peut pas être une variable, de par la règle modifiée ($\mathsf{VAR'}_r$), et il ne reste donc que quatre possibilités pour l'expression e, à savoir ref e_1, if e_1 then e_2 else e_3, let $x_0 = e_1$ in e_2 ou $(e_2\ e_1)$. (La séquence est vue comme un cas particulier de construction let.)

- $e \equiv \mathsf{ref}\ e_1$: L'évaluation de e est de la forme :

$$\begin{array}{rcllr} \theta.\mathsf{ref}\ e_1 & \longmapsto^\star & \theta_1.\mathsf{ref}\ v_1 & & \\ & \longmapsto & \theta_1.\{(x, v_1)\}.x & x \text{ fraîche} & (\mathit{ref}) \\ & \longmapsto & \theta_1 \uplus \{(x, v_1)\}.x & & (\theta_{merge}) \end{array}$$

  La condition de bord « x fraîche » implique $x \notin dom(\theta_1)$. Or $dom(\theta) \subseteq dom(\theta_1)$ (Lemme 2 page 20) donc $x \notin dom(\theta)$.

- $e \equiv \textsf{if } e_1 \textsf{ then } e_2 \textsf{ else } e_3$: En supposant que e_1 s'évalue en $\textsf{true}$, l'évaluation de e est de la forme :

$$\begin{array}{rcll} \theta.\textsf{if } e_1 \textsf{ then } e_2 \textsf{ else } e_3 & \longmapsto^\star & \theta_1.\textsf{if true then } e_2 \textsf{ else } e_3 & \\ & \longmapsto & \theta_1.e_2 & (\textit{if-true}) \\ & \longmapsto^\star & \theta'.x & \end{array}$$

  Par hypothèse de récurrence appliquée à $\theta_1.e_2 \longmapsto^\star \theta'.x$ on a $x \notin dom(\theta_1)$, d'où le résultat car $dom(\theta) \subseteq dom(\theta_1)$. Idem si e_1 s'évalue en $\textsf{false}$.

- $e \equiv \textsf{let } x_0 = e_1 \textsf{ in } e_2$: L'évaluation de e est de la forme :

$$\begin{array}{rcll} \theta.\textsf{let } x_0 = e_1 \textsf{ in } e_2 & \longmapsto^\star & \theta_1.\textsf{let } x_0 = v_1 \textsf{ in } e_2 & \\ & \longmapsto & \theta_1.e_2[x_0 \leftarrow v_1] & (\textit{let}) \\ & \longmapsto^\star & \theta'.x & \end{array}$$

  Par hypothèse de récurrence appliquée à $\theta_1.e_2[x_0 \leftarrow v_1] \longmapsto^\star \theta'.x$ on a $x \notin dom(\theta_1)$, d'où le résultat car $dom(\theta) \subseteq dom(\theta_1)$. (On remarquera que e_2 ne peut être une variable, du fait de la règle modifiée ($\textsf{VAR'}_r$), et donc ne peut être en particulier égale à x_0.)

- $e \equiv (e_2\ e_1)$: Comme e est de type $\tau\ \textsf{ref}$, e_2 ne peut s'évaluer ni en $\textsf{Y}$ ni en une constante. Dès lors, l'évaluation de e est de la forme :

$$\begin{array}{rcll} \theta.(e_2\ e_1) & \longmapsto^\star & \theta_1.(e_2\ v_1) & \\ & \longmapsto^\star & \theta_2.(\lambda x_0.e_3\ v_1) & \\ & \longmapsto & \theta_2.e_3[x_0 \leftarrow v_1] & (\beta) \\ & \longmapsto^\star & \theta'.x & \end{array}$$

  Par hypothèse de récurrence appliquée à $e_3[x_0 \leftarrow v_1]$ on a $x \notin dom(\theta_2)$, d'où le résultat car $dom(\theta) \subseteq dom(\theta_1) \subseteq dom(\theta_2)$.

□

Nous devons également montrer que les effets recouvrent bien l'intuition que nous en avons donnée, à savoir qu'il y a un rapport entre les ensembles de variables d'entrée et de sortie d'un effet et les variables effectivement lues et modifiées lors de l'évaluation d'un programme. En particulier, on aura besoin plus loin de la propriété suivante.

Lemme 4 *Soit Γ un environnement bien formé et e un programme tel que $\Gamma \vdash_r e : (\tau, (\rho, \omega))$. Soit θ un état associant des valeurs aux références de Γ. Alors*

$$\forall \theta', v.\quad \theta.e \longmapsto^\star \theta'.v \quad\Longrightarrow\quad \forall x \in dom(\theta)\backslash\omega.\ \theta'(x) = \theta(x)$$

Preuve. La preuve se fait par récurrence sur la longueur de la dérivation $\longmapsto^\star$ et par cas sur la structure de e.

- $e \equiv c \mid x \mid \textsf{fun } x \to e_1 \mid \textsf{Y} \mid !x$: Dans ce cas $\theta' = \theta$ et la conclusion est immédiate.

- $e \equiv \mathsf{ref}\ e_1$: L'évaluation de e est de la forme :

$$\begin{array}{llll} \theta.\mathsf{ref}\ e_1 & \longmapsto^\star & \theta_1.\mathsf{ref}\ v_1 & \text{avec } \theta.e_1 \longmapsto^\star \theta_1.v_1 \\ & \longmapsto & \theta_1.\{(x,v_1)\}.x & x \text{ fraîche} \qquad (\mathit{ref}) \\ & \longmapsto & \underbrace{\theta_1 \uplus \{(x,v_1)\}}_{=\theta'}.x & (\theta_{merge}) \end{array}$$

  Par la règle (REF_r) on a $\Gamma \vdash_r e_1 : (\_,(\rho,\omega))$ et par hypothèse de récurrence appliquée à $\theta.e_1 \longmapsto^\star \theta_1.v_1$ on a donc $\forall y \in \mathit{dom}(\theta)\backslash\omega.\ \theta_1(y) = \theta(y)$. Soit alors $y \in \mathit{dom}(\theta)\backslash\omega$. y ne peut être x, car la condition de bord « x fraîche » implique $x \notin \mathit{dom}(\theta_1)$ et d'autre part $\mathit{dom}(\theta_1) \supseteq \mathit{dom}(\theta)$. On en déduit $\theta'(y) = \theta_1(y)$ par la définition de $\uplus$. Et $y \in \mathit{dom}(\theta)\backslash\omega$, donc $\theta_1(y) = \theta(y)$. D'où finalement $\theta'(y) = \theta(y)$, le résultat escompté.
- $e \equiv x\ :=\ e_1$: L'évaluation de e est de la forme :

$$\begin{array}{llll} \theta.x\ :=\ e_1 & \longmapsto^\star & \theta_1.x\ :=\ v_1 & \text{avec } \theta.e_1 \longmapsto^\star \theta_1.v_1 \\ & \longmapsto & \underbrace{\theta_1 \uplus \{(x,v_1)\}}_{=\theta'}.\mathsf{void} & (\mathit{assign}) \end{array}$$

  Par la règle (AFF_r) on a $\Gamma \vdash_r e : (\_,(\rho_1,\omega_1))$ avec $\omega = \{x\} \cup \omega_1$. Or par hypothèse de récurrence appliquée à $\theta.e_1 \longmapsto^\star \theta_1.v_1$ on a $\forall y \in \mathit{dom}(\theta)\backslash\omega_1.\ \theta_1(y) = \theta(y)$. Soit alors $y \in \mathit{dom}(\theta)\backslash\omega$. On a $y \neq x$ car $y \notin \omega$, et donc $\theta'(y) = \theta_1(y)$. De plus $\theta_1(y) = \theta(y)$ car $y \notin \mathit{dom}(\theta)\backslash\omega_1$, étant donné que $\omega_1 \subseteq \omega$. D'où finalement $\theta'(y) = \theta(y)$.
- $e \equiv \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$: En supposant que e_1 s'évalue en true, l'évaluation de e est de la forme :

$$\begin{array}{llll} \theta.\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 & \longmapsto^\star & \theta_1.\mathsf{if}\ v_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 & \text{avec } \theta.e_1 \longmapsto^\star \theta_1.v_1 \\ & \longmapsto & \theta_1.e_2 & (\mathit{if\text{-}true}) \\ & \longmapsto^\star & \theta'.v & \end{array}$$

  Par la règle (COND_r) on a $\Gamma \vdash_r e_1 : (\_,(\rho_1,\omega_1))$, $\Gamma \vdash_r e_2 : (\_,(\rho_2,\omega_2))$ et $\Gamma \vdash_r e_3 : (\_,(\rho_3,\omega_3))$ avec $\omega = \omega_1 \cup \omega_2 \cup \omega_3$. Donc par hypothèse de récurrence appliquée à $\theta.e_1 \longmapsto^\star \theta_1.v_1$ et à $\theta_1.e_2 \longmapsto^\star \theta'.v$ on a

$$\begin{array}{l} \forall x \in \mathit{dom}(\theta)\backslash\omega_1.\ \theta_1(x) = \theta(x) \\ \forall x \in \mathit{dom}(\theta_1)\backslash\omega_2.\ \theta'(x) = \theta_1(x) \end{array}$$

  Soit alors $x \in \mathit{dom}(\theta)\backslash\omega$. Étant donné que $\mathit{dom}(\theta) \subseteq \mathit{dom}(\theta_1) \subseteq \mathit{dom}(\theta')$ on a de façon immédiate $x \in \mathit{dom}(\theta)\backslash\omega_1$ et $x \in \mathit{dom}(\theta_1)\backslash\omega_2$, et donc

$$\theta'(x) = \theta_1(x) = \theta(x)$$

  Idem si e_1 s'évalue en false.
- $e \equiv \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$: On distingue deux cas suivant que e_1 est ou non une référence.
  - $e_1 : \tau_1 \neq \_\ \mathsf{ref}$: la preuve est similaire à celle du if car $\omega = \omega_1 \cup \omega_2$.

– $e_1 : \tau_1$ ref : L'évaluation de e est de la forme :

$$\begin{array}{llll} \theta.\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 & \longmapsto^\star & \theta_1.\mathsf{let}\ x = x_1\ \mathsf{in}\ e_2 & \text{avec } \theta.e_1 \longmapsto^\star \theta_1.x_1 \\ & \longmapsto & \theta_1.e_2[x \leftarrow x_1] & (let) \\ & \longmapsto^\star & \theta'.v & \end{array}$$

Par la règle (LETREF_r) on a $\Gamma \vdash_r e_1 : (\tau_1\ \mathsf{ref}, (\rho_1, \omega_1))$ et $\Gamma, x : \tau_1\ \mathsf{ref} \vdash_r e_2 : (\_, (\rho_2, \omega_2))$ avec $\omega = \omega_1 \cup (\omega_2 \backslash x)$. De manière immédiate on a donc $\Gamma, x_1 : \tau_1\ \mathsf{ref} \vdash_r e_2[x \leftarrow x_1] : (\_, (\rho_2[x \leftarrow x_1], \omega_2[x \leftarrow x_1]))$. Alors par hypothèse de récurrence appliquée à $\theta.e_1 \longmapsto^\star \theta_1.x_1$ et $\theta_1.e_2[x \leftarrow x_1] \longmapsto^\star \theta'.v$ on a

$$\begin{array}{l} \forall y \in dom(\theta)\backslash\omega_1.\ \theta_1(y) = \theta(y) \\ \forall x \in dom(\theta_1)\backslash(\omega_2[x \leftarrow x_1]).\ \theta'(y) = \theta_1(y) \end{array}$$

Soit alors $y \in dom(\theta)\backslash\omega$. y ne peut être x (pas de capture de variable) ni x_1, car $x_1 \notin dom(\theta)$ par le lemme 3. Dès lors, $y \in dom(\theta_1)\backslash(\omega_2[x \leftarrow x_1])$ donc $\theta'(y) = \theta_1(y)$ et $y \in dom(\theta)\backslash\omega_1$ donc $\theta_1(y) = \theta(y)$ d'où le résultat.

– $e \equiv (e_2\ e_1)$: Là encore, on distingue deux cas suivant que e_1 est ou non une référence.

  – $e_1 : \tau_1 \neq \_$ ref : Ce cas est similaire au cas du let in, que e_2 s'évalue en une constante, Y ou une abstraction.

  – $e_1 : \tau_1$ ref : Dans ce cas l'évaluation de e est de la forme :

$$\begin{array}{llll} \theta.(e_2\ r) & \longmapsto^\star & \theta_1.(\lambda x.e_3\ r) & \\ & \longmapsto & \theta_1.e_3[x \leftarrow r] & (\beta) \\ & \longmapsto^\star & \theta'.v & \end{array}$$

Par la règle (APPREF_r) on a $\Gamma \vdash_r e_2 : ((x : \tau_1\ \mathsf{ref}) \to^{\epsilon_3} \tau, \epsilon_2)$ où $\epsilon_2 = (\rho_2, \omega_2)$ et $\epsilon_3 = (\rho_3, \omega_3)$ avec $\omega = \omega_2 \cup \omega_3[x \leftarrow r]$. Par hypothèse de récurrence appliqué à $\theta.e_2 \longmapsto^\star \theta_1.e_3$ et à $\theta_1.e_3[x \leftarrow r] \longmapsto^\star \theta'.v$ on a

$$\begin{array}{l} \forall y \in dom(\theta)\backslash\omega_2.\ \theta_1(y) = \theta(y) \\ \forall x \in dom(\theta_1)\backslash(\omega_3[x \leftarrow r]).\ \theta'(y) = \theta_1(y) \end{array}$$

et la conclusion est immédiate.

□

1.2.2 Exceptions

Nous allons maintenant ajouter des exceptions dans notre langage dans le but de dégager ce qu'il y a de général dans la notion d'effet et son inférence. Cette extension se limitera toutefois à cette section.

Nous allons nous placer dans une situation où l'on ne manipule pas d'expression d'exception, afin de rendre possible l'analyse statique des effets : de la même manière que l'on n'autorise pas des expressions de la forme $e := e_1$ ou $!e$ dans le cas des références, on n'autorisera pas la construction raise e où e est une expression quelconque. Dès lors, le système de types n'a donc pas besoin d'être modifié et on se contentera de la notation

$$\chi : \tau \ \mathsf{exn}$$

pour exprimer que χ est une exception dont l'argument est de type τ, qui sera ici un type pur. Les exceptions sont supposées être déclarées globalement — comme en Caml — et appartenir à une nouvelle classe de noms. Cela implique que les environnements ne contiennent pas d'exceptions, et leur définition reste donc inchangée. Les expressions de programme sont alors augmentées des deux constructions suivantes :

$$e \quad ::= \quad \dots \mid \mathsf{raise}\ (\chi\ e) \mid \mathsf{try}\ e\ \mathsf{with}\ (\chi\ x) \to e$$

où χ représente un nom d'exception. La construction raise $(\chi\ e)$ évalue e puis lève l'exception χ avec pour argument le résultat de cette évaluation. La construction try e_1 with $(\chi\ x) \to e_2$ évalue e_1 et retourne sa valeur, sauf dans le cas où l'évaluation de e_1 lève l'exception χ, auquel cas e_2 est évaluée dans le contexte où x est liée à l'argument de l'exception, et la valeur de e_2 est retournée comme résultat.

Ces deux constructions sont typées de manière usuelle, par les deux règles suivantes :

$$\frac{\chi : \tau\ \mathsf{exn} \qquad \Gamma \vdash e : \tau \qquad \Gamma \vdash \tau_1\ \mathsf{bf}}{\Gamma \vdash \mathsf{raise}\ (\chi\ e) : \tau_1}(\mathsf{RAISE})$$

$$\frac{\chi : \tau\ \mathsf{exn} \qquad \Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau \vdash e_2 : \tau_1}{\Gamma \vdash \mathsf{try}\ e_1\ \mathsf{with}\ (\chi\ x) \to e_2 : \tau_1}(\mathsf{TRY})$$

Dans le cas d'exceptions, déterminer les effets consiste à déterminer, pour chaque expression, les exceptions potentiellement levées par l'évaluation de cette expression. On redéfinit donc la notion d'effet de la manière suivante :

Définition 17 *Un effet est un triplet (ρ, ω, X) où ρ et ω sont deux ensembles de variables tels que $\omega \subseteq \rho$ et X un ensemble d'exceptions. L'effet vide, toujours noté $\bot$, est le triplet $(\emptyset, \emptyset, \emptyset)$.*

Les notions de types de valeurs et de types de calculs ne changent pas : ce sont toujours celles données par la définition 10 page 23. De même, les règles de typage avec effets *ne changent pas*, dès lors que l'on redéfinit l'opération $\sqcup$ sur le nouveau type d'effet :

Définition 18 *L'opération binaire $\sqcup$ sur les effets est redéfinie de la manière suivante :*

$$(\rho_1, \omega_1, X_1) \sqcup (\rho_2, \omega_2, X_2) \quad \stackrel{def}{=} \quad (\rho_1 \cup \rho_2, \omega_1 \cup \omega_2, X_1 \cup X_2)$$

Il nous suffit alors d'ajouter aux règles de la figure 1.4 les règles correspondant aux constructions raise et try et pour une parfaite rigueur à réécrire les règles (DEREF_r) et (AFF_r).

Définition 19 *Le jugement $\Gamma \vdash_r e : \kappa$ est défini inductivement sur la structure de e par l'ensemble des règles d'inférence données figure 1.4, dont on modifie les deux règles suivantes :*

$$\frac{x : \tau\ \mathsf{ref} \in \Gamma}{\Gamma \vdash_r\ !x : (\tau, (\{x\}, \emptyset, \emptyset))}(\mathsf{DEREF}_r) \qquad \frac{x : \tau\ \mathsf{ref} \in \Gamma \qquad \Gamma \vdash_r e : (\tau, (\rho, \omega, X))}{\Gamma \vdash_r x := e : (\mathsf{unit}, (\{x\} \cup \rho, \{x\} \cup \omega, X))}(\mathsf{AFF}_r)$$

et auxquelles on ajoute les deux règles suivantes :

$$\frac{\chi : \tau\ \mathsf{exn} \qquad \Gamma \vdash_r e : (\tau, (\rho, \omega, X)) \qquad \Gamma \vdash_r \tau_1\ \mathsf{bf}}{\Gamma \vdash_r \mathsf{raise}\ (\chi\ e) : (\tau_1, (\rho, \omega, \{\chi\} \cup X))}(\mathsf{RAISE}_r)$$

$$\frac{\chi : \tau\ \mathsf{exn} \qquad \Gamma \vdash_r e_1 : (\tau_1, (\rho_1, \omega_1, X_1)) \qquad \Gamma, x : \tau \vdash_r e_2 : (\tau_1, (\rho_2, \omega_2, X_2))}{\Gamma \vdash_r \mathsf{try}\ e_1\ \mathsf{with}\ (\chi\ x) \rightarrow e_2 : (\tau_1, (\rho_1 \cup \rho_2, \omega_1 \cup \omega_2, X_1 \backslash \chi \cup X_2))}(\mathsf{TRY}_r)$$

Il n'y a là rien de nouveau par rapport à l'article de J.-C. Guzmán et A. Suárez [20] — qui va même un peu plus loin en traitant du polymorphisme d'effet — si ce n'est le fait que nous introduisons cette inférence des levées d'exceptions dans un cadre plus général introduit avec les références. Cela nous permet de dégager une certaine généralité dans les notions de types de valeurs et de types de calculs que nous avons introduites, ainsi que dans les règles d'inférence concernant les constructions « fonctionnelles » du langage. C'est cette généralisation que nous allons tenter de faire maintenant.

1.2.3 Généralisation

De l'étude des références et des exceptions, nous pouvons retenir que l'analyse statique des effets requiert une notion d'*effet nul*, utilisé par exemple pour typer les valeurs, et une notion de *composition d'effets*, utilisée chaque fois que plusieurs évaluations se succèdent. D'autre part, une relation d'ordre partielle sur les effets s'avérera nécessaire par la suite. Dès lors, un effet s'apparente à un semi-treillis.

Définition 20 (effet) *Un* effet *est un semi-treillis* $\mathcal{E} = (E, \bot, \sqcup, \sqsubseteq)$.

L'intuition qu'il faut avoir de ce semi-treillis est que $\bot$ représente l'*effet nul*, c'est-à-dire l'absence d'effet, $\sqcup$ la composition de deux effets et $\sqsubseteq$ la comparaison de deux effets, l'idée étant qu'une expression d'effet ϵ_1 peut également être considérée comme une expression d'effet ϵ_2 dès lors que $\epsilon_1 \sqsubseteq \epsilon_2$. Les notions de types de valeurs et de calculs ne changent pas, au moins en ce qui concerne la partie fonctionnelle.

Définition 21 *Les types de valeurs et de calculs sont définis mutuellement récursivement de la manière suivante :*

$$\begin{cases} \textit{valeurs} & \tau & ::= & (\tau, \ldots, \tau)c \mid \tau \rightarrow \kappa \mid \ \ldots \\ \textit{calculs} & \kappa & ::= & (\tau, \epsilon) \end{cases}$$

où les types de valeurs peuvent être éventuellement complétés par d'autres constructions de types.

La notion d'environnement reste la même, associant des types de valeurs à des variables. La notion de type bien formé dépend, quant à elle, de l'effet choisi. On peut alors donner les règles génériques du typage avec effets, qui permettent d'inférer statiquement le jugement $\Gamma \vdash e : \tau$. Elles sont regroupées figure 1.6. À ces règles génériques viennent s'adjoindre, dans chaque cas particulier, des règles spécifiques aux effets envisagés et aux constructions du langage qui leur sont associées.

Enfin, notons que l'ordre $\sqsubseteq$ peut être étendu sur les types de valeurs et de calculs.

$$\frac{\mathit{Type}(c) = \tau}{\Gamma \vdash c : (\tau, \bot)}(\mathsf{CONST}_g) \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : (\tau, \bot)}(\mathsf{VAR}_g)$$

$$\frac{\Gamma, x : \tau \vdash e : \kappa \qquad \Gamma \vdash \tau \ \mathsf{bf}}{\Gamma \vdash \mathsf{fun}\ (x : \tau) \rightarrow e : ((x : \tau) \rightarrow \kappa, \bot)}(\mathsf{FUN}_g) \qquad \frac{\Gamma, \alpha \vdash e : (\tau, \epsilon) \qquad \alpha \notin \Gamma}{\Gamma \vdash \mathsf{Fun}\ \alpha \rightarrow e : (\forall \alpha.\tau, \epsilon)}(\mathsf{FUNT}_g)$$

$$\frac{\Gamma \vdash e : (\tau_1 \rightarrow^{\epsilon} \tau, \epsilon_2) \qquad \Gamma \vdash e_1 : (\tau_1, \epsilon_1)}{\Gamma \vdash (e\ e_1) : (\tau, \epsilon_2 \sqcup \epsilon_1 \sqcup \epsilon)}(\mathsf{APP}_g)$$

$$\frac{\Gamma \vdash e : (\forall \alpha.\tau, \epsilon) \qquad \Gamma \vdash \tau_1 \ \mathsf{bf}}{\Gamma \vdash (e\ \tau_1) : (\tau[\alpha \leftarrow \tau_1], \epsilon)}(\mathsf{APPT}_g)$$

$$\frac{\tau = \tau_1 \rightarrow \kappa_2 \qquad \Gamma \vdash \tau \ \mathsf{bf} \qquad \Gamma, f : \tau \vdash e : (\tau, \bot)}{\Gamma \vdash \mathsf{rec}\ f\ : \tau = e : (\tau, \bot)}(\mathsf{REC}_g)$$

$$\frac{\Gamma \vdash e_1 : (\mathsf{bool}, \epsilon_1) \qquad \Gamma \vdash e_2 : (\tau, \epsilon_2) \qquad \Gamma \vdash e_3 : (\tau, \epsilon_3)}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : (\tau, \epsilon_1 \sqcup \epsilon_2 \sqcup \epsilon_3)}(\mathsf{COND}_g)$$

$$\frac{\Gamma \vdash e_1 : (\tau_1, \epsilon_1) \qquad \Gamma, x : \tau_1 \vdash e_2 : (\tau, \epsilon)}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : (\tau, \epsilon_1 \sqcup \epsilon)}(\mathsf{LET}_g)$$

$$\frac{\Gamma \vdash e_1 : (\mathsf{unit}, \epsilon_1) \qquad \Gamma \vdash e_2 : (\tau, \epsilon_2)}{\Gamma \vdash e_1\ ;\ e_2 : (\tau, \epsilon_1 \sqcup \epsilon_2)}(\mathsf{SEQ}_g)$$

$$\frac{\Gamma \vdash e_1 : (\mathsf{bool}, \epsilon_1) \qquad \Gamma \vdash e_2 : (\mathsf{unit}, \epsilon_2)}{\Gamma \vdash \mathsf{while}\ e_1\ \mathsf{do}\ e_2\ \mathsf{done} : (\mathsf{unit}, \epsilon_1 \sqcup \epsilon_2)}(\mathsf{LOOP}_g)$$

FIG. 1.6: Typage avec effets (règles génériques)

Définition 22 *L'ordre $\sqsubseteq$ sur les effets est étendu aux types de valeurs et de calculs, sur les types ayant la même structure, de la manière suivante :*

$$\begin{cases} x \sqsubseteq x \\ \tau_1 \to \kappa_1 \sqsubseteq \tau_2 \to \kappa_2 & \stackrel{def}{\iff} \quad \tau_2 \sqsubseteq \tau_1 \wedge \kappa_1 \sqsubseteq \kappa_2 \\ (\tau_1, \epsilon_1) \sqsubseteq (\tau_2, \epsilon_2) & \stackrel{def}{\iff} \quad \tau_1 \sqsubseteq \tau_2 \wedge \epsilon_1 \sqsubseteq \epsilon_2 \end{cases}$$

On remarquera le caractère contra-variant de l'inégalité sur les types flèches. L'intuition de cette inégalité reste la même, à savoir qu'une expression de type τ_1 (resp. κ_1) peut être considérée comme une expression de type τ_2 (resp. κ_2) dès lors que $\tau_1 \sqsubseteq \tau_2$ (resp. $\kappa_1 \sqsubseteq \kappa_2$).

1.3 Monades et traduction fonctionnelle

Nous venons donc de montrer comment l'on peut analyser statiquement les effets d'une certaine classe de programmes. C'est sur la base de cette information que nous allons bâtir notre traduction fonctionnelle, l'idée étant qu'un programme dont les effets nous disent qu'il accède aux références x et y, x en lecture seule et y en lecture/écriture, deviendra une fonction prenant en argument les valeurs de x et y et retournant, outre le résultat de l'évaluation, la nouvelle valeur de y.

Nous pourrions donner directement la traduction fonctionnelle dans le cas du langage introduit à la section 1.1, mais il se dégage de sa définition une certaine généralité que l'on peut relier facilement à la généralité du système de types avec effets défini à la section 1.2.3. On retrouve alors, légèrement raffinée, la notion générale de *monade*, introduite en informatique séparément par E. Moggi [38, 39] et P. Wadler [57, 56, 30].

Le premier paragraphe présente le formalisme des monades dans sa généralité, et montre comment on peut le raffiner lorsque on l'associe à un système de types avec effets. Puis, dans le second paragraphe, nous définissons une traduction fonctionnelle pour notre langage, basée sur une monade particulière associée au système de types que nous avons introduit dans la section 1.2.1. Enfin, dans le dernier paragraphe, nous montrerons la correction sémantique de cette traduction.

1.3.1 Monades

La notion de monades a été introduite en informatique séparément par E. Moggi [38, 39] et P. Wadler [57, 56, 30], dans des contextes relativement différents. Elles sont utilisées par E. Moggi pour définir la sémantique des langages de programmation, alors que P. Wadler les utilise d'une façon bien plus pragmatique pour introduire des aspects « impératifs » (références, exceptions, entrées-sorties, ...) dans des langages de programmation purement fonctionnels. Dans les deux cas, il s'agit cependant d'idées similaires, consistant à exprimer des traits sémantiques impératifs dans des formalismes fonctionnels, que ceux-ci soient du domaine de la méta-théorie ou bien aussi concrets que le langage Haskell. On imagine déjà en quoi ces idées sont proches de nos préoccupations.

L'idée générale des monades est d'introduire, pour chaque type A, un type $\mu(A)$ des *calculs de type* A ainsi que deux opérateurs unit et star, de types :

$$\begin{array}{lcl} \mathsf{unit} & : & A \rightarrow \mu(A) \\ \mathsf{star} & : & \mu(A) \rightarrow (A \rightarrow \mu(B)) \rightarrow \mu(B) \end{array}$$

et dont les significations sont les suivantes : l'opérateur unit est une injection des valeurs vers les calculs ; l'opérateur star permet d'évaluer un calcul puis de passer son résultat à une fonction retournant un nouveau calcul. Il est en fait plus clair de voir l'opérateur star comme une construction $\mathsf{let\ in}$, que l'on peut définir de la manière suivante :

$$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \quad \stackrel{\text{def}}{=} \quad (\mathsf{star}\ e_1\ \lambda x.e_2)$$

En outre, les opérateurs unit et star doivent vérifier les trois axiomes suivants :

$$\begin{array}{ll} (M_1) & (\mathsf{star}\ (\mathsf{unit}\ x)\ f) = (f\ x) \\ (M_2) & (\mathsf{star}\ m\ \lambda x.(\mathsf{unit}\ x)) = m \\ (M_3) & (\mathsf{star}\ m\ \lambda x.(\mathsf{star}\ (g\ x)\ f)) = (\mathsf{star}\ (\mathsf{star}\ m\ g)\ f) \end{array}$$

Une idée importante derrière les monades est que l'on peut manipuler des calculs à travers ces deux opérateurs sans savoir de quelle manière le type $\mu(A)$ est effectivement représenté, et même *sans savoir ce qu'il représente.*

Une monade pour les exceptions. Un bon exemple valant mieux qu'un long discours, donnons un exemple de monade permettant de manipuler des exceptions (sans argument pour l'instant) dans un langage fonctionnel. Il suffit de définir

$$\mu_e(A) \quad \stackrel{\text{def}}{=} \quad \mathsf{Value\ of}\ A \mid \mathsf{Exc\ of}\ id$$

où id est un type représentant les noms des exceptions. On exprime par là qu'un calcul est soit une valeur de type A, soit une exception. L'opérateur unit_e est immédiatement représenté par le constructeur Value. L'opérateur star, quant à lui, évalue son premier argument et, en cas d'exception, se contente de propager cette exception, et sinon lie le résultat de cette évaluation dans son second argument. Il se définit donc de la manière suivante :

$$(\mathsf{star}_e\ m\ f) \quad \stackrel{\text{def}}{=} \quad \mathsf{match}\ m\ \mathsf{with} \quad \begin{array}{ll} \mathsf{Value}\ x & \Rightarrow (f\ x) \\ \mid\ \mathsf{Exc}\ e & \Rightarrow \mathsf{Exc}\ e \end{array}$$

Bien entendu, une monade particulière peut introduire des constructions particulières venant s'ajouter aux opérateurs unit et star. En ce qui concerne les exceptions, il s'agira des constructions raise et $\mathsf{try\ with}$ que l'on peut définir ici de la manière suivante :

$$\begin{array}{lcl} \mathsf{raise} & : & id \rightarrow \mu_e(A) \\ \mathsf{raise}\ e & \stackrel{\text{def}}{=} & \mathsf{Exc}\ e \end{array}$$

et

$$\begin{array}{lcl} \mathsf{try}\ \_\ \mathsf{with}\ \_ & : & \mu_e(A) \to (id \to \mu_e(A)) \to \mu_e(A) \\ \mathsf{try}\ x\ \mathsf{with}\ h & \stackrel{\mathrm{def}}{=} & \mathsf{match}\ x\ \mathsf{with} \\ & & \quad \mathsf{Value}\ x \ \Rightarrow \mathsf{Value}\ x \\ & & \mid\ \mathsf{Exc}\ e \quad \Rightarrow (h\ e) \end{array}$$

Il y a bien sûr d'autres manières de définir l'opérateur try with, suivant que l'on veut en faire un opérateur retournant une valeur dans A ou $\mu(A)$.

Une monade pour les références. Un deuxième exemple, qui nous intéresse directement, est une monade pour les références. On suppose que l'on dispose d'un type S réalisant une table d'association entre des clés du type K et des objets du type T. Ce type représentera l'état des références, elles-mêmes représentées directement par le type des clés K. La monade des références peut alors être représentée par le type des calculs suivant :

$$\mu_r(A) \quad \stackrel{\mathrm{def}}{=} \quad S \to S \times A$$

En effet, un calcul prend en argument un état d'*entrée* et retourne un état de *sortie* accompagné du résultat du calcul. Dès lors, sans rien dire de plus sur le type S des états, on peut définir les opérateurs unit_r et star_r de la monade des références :

$$\begin{array}{lcl} (\mathsf{unit}_r\ x) & \stackrel{\mathrm{def}}{=} & \lambda s.(s, x) \\ (\mathsf{star}_r\ m\ f) & \stackrel{\mathrm{def}}{=} & \lambda s.\mathsf{let}\ (s_1, x) = (m\ s)\ \mathsf{in}\ (f\ x\ s_1) \end{array}$$

De même que pour les exceptions, il faut ensuite définir des constructeurs particuliers aux références. Si l'on suppose que les opérations d'accès et de modification sur le type S sont respectivement $get : S \to K \to T$ et $set : S \to K \to T \to S$, alors on peut définir les constructions $!x$ et $x := e$ sur les références de la manière suivante :

$$\begin{array}{lcl} !\_ & : & K \to \mu_r(T) \\ !x & \stackrel{\mathrm{def}}{=} & \lambda s.(s, get\ s\ x) \end{array}$$

et

$$\begin{array}{lcl} \_ := \_ & : & K \to \mu_r(T) \to \mu_r(\mathsf{unit}) \\ x := e & \stackrel{\mathrm{def}}{=} & \lambda s.\mathsf{let}\ (s_1, v) = (e\ s)\ \mathsf{in}\ (set\ s_1\ x\ v, \mathsf{void}) \end{array}$$

On remarquera que l'on aurait pu définir l'opération $x := e$ de la même manière en utilisant l'opérateur star_r :

$$x := e \quad \stackrel{\mathrm{def}}{=} \quad (\mathsf{star}_r\ e\ \lambda v.\lambda s_1.(set\ s_1\ x\ v, \mathsf{void}))$$

On trouvera de nombreux autres exemples dans l'article de P. Wadler intitulé *Monads for functional programming* [57]. Nous allons maintenant montrer comment la notion de monade peut être raffinée lorsqu'elle est associée à un système de types avec effets tel que celui que nous avons introduit à la section 1.2.3.

Généralisation

La monade que nous venons de présenter pour les références est trop grossière : elle manipule en effet un état global de toutes les références, et c'est là justement ce que nous cherchons à éviter depuis le début. Le système de types avec effets que nous avons introduit pour notre langage nous donne une information beaucoup plus fine sur chaque expression, nous renseignant sur les références accédées ou modifiées de façon individuelle. En conséquence, il doit y avoir autant d'opérateurs unit et star qu'il y a d'effets possibles. C'est cette idée que nous allons formaliser en introduisant la notion de monade généralisée.

Définition 23 (monade généralisée) *Une* monade généralisée *associée à un effet* $\mathcal{E} = (E, \bot, \sqcup, \sqsubseteq)$ *est constituée par la donnée d'une famille d'opérateurs de type* μ_ϵ *indexée sur* E*, et de deux familles d'opérateurs,* $\mathsf{unit}_{\epsilon_1,\epsilon_2}$ *indexée sur* $\{(\epsilon_1,\epsilon_2) \in E^2 \mid \epsilon_1 \sqsubseteq \epsilon_2\}$ *et* $\mathsf{star}_{\epsilon_1,\epsilon_2}$ *indexée sur* E^2 *de types respectifs*

$$\begin{array}{lcl} \mathsf{unit}_{\epsilon_1,\epsilon_2} & : & \mu_{\epsilon_1}(A) \to \mu_{\epsilon_2}(A) \\ \mathsf{star}_{\epsilon_1,\epsilon_2} & : & \mu_{\epsilon_1}(A) \to (A \to \mu_{\epsilon_2}(B)) \to \mu_{\epsilon_1 \sqcup \epsilon_2}(B) \end{array}$$

satisfaisant les six identités suivantes :

$$\begin{array}{ll} (G_1) & \mu_\bot(A) = A \\ (G_2) & (\mathsf{unit}_{\epsilon,\epsilon}\ x) = x \\ (G_3) & (\mathsf{star}_{\bot,\epsilon}\ x\ f) = (f\ x) \\ (G_4) & (\mathsf{star}_{\epsilon_1,\epsilon_2}\ m\ \lambda x.(\mathsf{unit}_{\bot,\epsilon_2}\ x)) = (\mathsf{unit}_{\epsilon_1,\epsilon_1\sqcup\epsilon_2}\ m) \\ (G_5) & (\mathsf{star}_{\epsilon_1,\epsilon_2\sqcup\epsilon_3}\ m\ \lambda x.(\mathsf{star}_{\epsilon_2,\epsilon_3}\ (g\ x)\ f)) = (\mathsf{star}_{\epsilon_1\sqcup\epsilon_2,\epsilon_3}\ (\mathsf{star}_{\epsilon_1,\epsilon_2}\ m\ g)\ f) \\ (G_6) & (\mathsf{star}_{\epsilon_2,\epsilon_3}\ (\mathsf{unit}_{\epsilon_1,\epsilon_2}\ x)\ f) = (\mathsf{unit}_{\epsilon_1\sqcup\epsilon_3,\epsilon_2\sqcup\epsilon_3}\ (\mathsf{star}_{\epsilon_1,\epsilon_3}\ x\ f)) \qquad \textit{pour}\ \epsilon_1 \sqsubseteq \epsilon_2 \end{array}$$

La notion classique de monade s'avère être un cas particulier de monade généralisée. En effet, soit $M_0 = (\mu_0, \mathsf{unit}_0, \mathsf{star}_0)$ une monade classique. Alors nous pouvons construire une monade généralisée associée au treillis des booléens qui lui est équivalente. Les deux éléments du treillis des booléens, $\bot$ et $\top$, expriment respectivement l'absence et la présence d'effet dans une expression, c'est-à-dire les notions de valeurs et de calculs dans les monades classiques. Plus précisément, nous pouvons définir la monade généralisée associée à M_0 de la manière suivante :

- L'opérateur μ est défini par

$$\mu_\bot = id \qquad \text{et} \qquad \mu_\top = \mu_0$$

- L'opérateur unit est défini par

$$\mathsf{unit}_{\bot,\bot} = \mathsf{unit}_{\top,\top} = id \qquad \text{et} \qquad \mathsf{unit}_{\bot,\top} = \mathsf{unit}_0$$

– L'opérateur star est défini par

$$\begin{array}{rcl}
\mathsf{star}_{\perp,\perp} & : & A \to (A \to B) \to B \\
 & = & \lambda x.\lambda f.(f\ x) \\
\mathsf{star}_{\perp,\top} & : & A \to (A \to \mu_0(B)) \to \mu_0(B) \\
 & = & \lambda x.\lambda f.(f\ x) \\
\mathsf{star}_{\top,\perp} & : & \mu_0(A) \to (A \to B) \to \mu_0(B) \\
 & = & \lambda x.\lambda f.(\mathsf{star}_0\ x\ \lambda v.(\mathsf{unit}_0\ (f\ v)) \\
\mathsf{star}_{\top,\top} & : & \mu_0(A) \to (A \to \mu_0(B)) \to \mu_0(B) \\
 & = & \mathsf{star}_0
\end{array}$$

Le lemme suivant établit qu'il s'agit bien là d'une monade généralisée.

Lemme 5 *Le triplet* $(\mu, \mathsf{unit}, \mathsf{star})$ *défini ci-dessus à partir d'une monade* $(\mu_0, \mathsf{unit}_0, \mathsf{star}_0)$ *est une monade généralisée associée au treillis des booléens.*

Preuve. Nous devons montrer que les opérateurs μ, unit et star vérifient bien les identitées (G_1) à (G_6).

– (G_1), (G_2) et (G_3) : immédiat par définition.

– (G_4) : on raisonne par cas sur ϵ_1 et ϵ_2.

  – $\epsilon_1 = \epsilon_2 = \perp$:

$$\begin{array}{rcl}
(\mathsf{star}_{\perp,\perp}\ m\ \lambda x.(\mathsf{unit}_{\perp,\perp}\ x)) & = & (\mathsf{star}_{\perp,\perp}\ m\ \lambda x.x) \\
 & = & m \\
 & = & (\mathsf{unit}_{\perp,\perp}\ m)
\end{array}$$

  – $\epsilon_1 = \epsilon_2 = \top$:

$$\begin{array}{rcl}
(\mathsf{star}_{\top,\top}\ m\ \lambda x.(\mathsf{unit}_{\perp,\top}\ x)) & = & (\mathsf{star}_0\ m\ \lambda x.(\mathsf{unit}_0\ x)) \\
 & = & m \\
 & = & (\mathsf{unit}_{\top,\top}\ m)
\end{array}$$

  – $\epsilon_1 = \perp$ et $\epsilon_2 = \top$:

$$\begin{array}{rcl}
(\mathsf{star}_{\perp,\top}\ m\ \lambda x.(\mathsf{unit}_{\perp,\top}\ x)) & = & (\mathsf{unit}_0\ m) \\
 & = & (\mathsf{unit}_{\perp,\top}\ m)
\end{array}$$

  – $\epsilon_1 = \top$ et $\epsilon_2 = \perp$:

$$\begin{array}{rcl}
(\mathsf{star}_{\top,\perp}\ m\ \lambda x.(\mathsf{unit}_{\perp,\perp}\ x)) & = & (\mathsf{star}_0\ m\ \lambda x.(\mathsf{unit}_0\ (\mathsf{unit}_{\perp,\perp}\ x))) \\
 & = & (\mathsf{star}_0\ m\ \lambda x.(\mathsf{unit}_0\ x)) \\
 & = & m \\
 & = & (\mathsf{unit}_{\top,\top}\ m)
\end{array}$$

– (G_5) : on raisonne par cas sur ϵ_1, ϵ_2 et ϵ_3.

  – si $\epsilon_1 = \epsilon_2 = \epsilon_3 = \perp$ alors l'identité est immédiate, les deux membres valant $(f\ (g\ m))$.

– si $\epsilon_1 = \bot$ et $\epsilon_2 \sqcup \epsilon_3 = \top$ alors

$$\begin{array}{rcl}(\mathsf{star}_{\bot,\top}\ m\ \lambda x.(\mathsf{star}_{\epsilon_2,\epsilon_3}\ x)) & = & (\mathsf{star}_{\epsilon_2,\epsilon_3}\ (g\ m)\ f) \\ & = & (\mathsf{star}_{\epsilon_2,\epsilon_3}\ (\mathsf{star}_{\bot,\epsilon_2}\ m\ g)\ f)\end{array}$$

– si $\epsilon_1 = \top$ et $\epsilon_2 = \epsilon_3 = \bot$ alors

$$(\mathsf{star}_{\top,\bot}\ m\ \lambda x.(\mathsf{star}_{\bot,\bot}\ (g\ x)\ f)) \quad = \quad (\mathsf{star}_0\ m\ \lambda x.(\mathsf{unit}_0\ (f\ (g\ x))))$$

et

$$\begin{array}{rcl}(\mathsf{star}_{\top,\bot}\ (\mathsf{star}_{\top,\bot}\ m\ g)\ f) & = & (\mathsf{star}_0\ (\mathsf{star}_0\ m\ \lambda x.(\mathsf{unit}_0\ (g\ x)))\ \lambda x.(\mathsf{unit}_0\ (f\ x)) \\ & = & (\mathsf{star}_0\ m\ \lambda x.(\mathsf{star}_0\ (\mathsf{unit}_0\ (g\ x))\ \lambda x.(\mathsf{unit}_0\ (f\ x)))) \\ & = & (\mathsf{star}_0\ m\ \lambda x.(\mathsf{unit}_0\ (f\ (g\ x))))\end{array}$$

– si $\epsilon_1 = \epsilon_2 = \epsilon_3 = \top$ alors c'est exactement l'identité (M_3).

– si $\epsilon_1 = \epsilon_2 = \top$ et $\epsilon_3 = \bot$ alors

$$\begin{array}{rcl}(\mathsf{star}_{\top,\top}\ m\ \lambda x.(\mathsf{star}_{\top,\bot}\ (g\ x)\ f)) & = & (\mathsf{star}_0\ m\ \lambda x.(\mathsf{star}_0\ (g\ x)\ \lambda y.(\mathsf{unit}_0\ (f\ y)))) \\ & = & (\mathsf{star}_0\ (\mathsf{star}_0\ m\ g)\ \lambda y.(\mathsf{unit}_0\ (f\ y))) \\ & = & (\mathsf{star}_{\top,\bot}\ (\mathsf{star}_{\top,\top}\ m\ g)\ f)\end{array}$$

– si $\epsilon_1 = \epsilon_3 = \top$ et $\epsilon_2 = \bot$ alors

$$\begin{array}{rcl}(\mathsf{star}_{\top,\top}\ m\ \lambda x.(\mathsf{star}_{\bot,\top}\ (g\ x)\ f) & = & (\mathsf{star}_0\ m\ \lambda x.(f\ (g\ x))) \\ & = & (\mathsf{star}_0\ m\ \lambda x.(\mathsf{star}_0\ (\mathsf{unit}_0\ (g\ x))\ f)) \\ & = & (\mathsf{star}_0\ (\mathsf{star}_0\ m\ \lambda x.(\mathsf{unit}_0\ (g\ x)))\ f) \\ & = & (\mathsf{star}_{\top,\top}\ (\mathsf{star}_{\top,\bot}\ m\ g)\ f)\end{array}$$

– (G_6) : on raisonne par cas ϵ_1 et ϵ_2.
  – si $\epsilon_1 = \epsilon_2 = \bot$ alors

$$\begin{array}{rcl}(\mathsf{star}_{\bot,\epsilon_3}\ (\mathsf{unit}_{\bot,\bot}\ x)\ f) & = & (\mathsf{star}_{\bot,\epsilon_3}\ x\ f) \\ & = & (\mathsf{unit}_{\epsilon_3,\epsilon_3}\ (\mathsf{star}_{\bot,\epsilon_3}\ x\ f))\end{array}$$

  – si $\epsilon_1 = \epsilon_2 = \top$ alors

$$\begin{array}{rcl}(\mathsf{star}_{\top,\epsilon_3}\ (\mathsf{unit}_{\top,\top}\ x)\ f) & = & (\mathsf{star}_{\top,\epsilon_3}\ x\ f) \\ & = & (\mathsf{unit}_{\top,\top}\ (\mathsf{star}_{\bot,\epsilon_3}\ x\ f))\end{array}$$

  – si $\epsilon_1 = \bot$, $\epsilon_2 = \top$ et $\epsilon_3 = \bot$ alors

$$\begin{array}{rcl}(\mathsf{star}_{\top,\bot}\ (\mathsf{unit}_{\bot,\top}\ x)\ f) & = & (\mathsf{star}_0\ (\mathsf{unit}_0\ x)\ \lambda y.(\mathsf{unit}_0\ (f\ y))) \\ & = & (\mathsf{unit}_0\ (f\ x)) \\ & = & (\mathsf{unit}_{\bot,\top}\ (\mathsf{star}_{\bot,\bot}\ x\ f))\end{array}$$

– si $\epsilon_1 = \bot$, $\epsilon_2 = \epsilon_3 = \top$ alors

$$\begin{array}{rcl}(\mathsf{star}_{\top,\top}\ (\mathsf{unit}_{\bot,\top}\ x)\ f) & = & (\mathsf{star}_0\ (\mathsf{unit}_0\ x)\ f) \\ & = & (f\ x) \\ & = & (\mathsf{unit}_{\top,\top}\ (\mathsf{star}_{\bot,\top}\ x\ f))\end{array}$$

□

Réciproquement, étant donnée une monade généralisée $(\mu, \mathsf{unit}, \mathsf{star})$ associée à un effet $\mathcal{E} = (E, \bot, \sqcup, \sqsubseteq)$, et étant donné un effet $\epsilon_0 \in E$, on peut définir les opérateurs suivants :

$$\begin{array}{rcl}\mu_0 & = & \mu_{\epsilon_0} \\ \mathsf{unit}_0 & = & \mathsf{unit}_{\bot,\epsilon_0} \\ \mathsf{star}_0 & = & \mathsf{star}_{\epsilon_0,\epsilon_0}\end{array}$$

et il s'agit alors d'une monade au sens classique du terme.

Lemme 6 *Le triplet* $(\mu_0, \mathsf{unit}_0, \mathsf{star}_0)$ *défini ci-dessus forme une monade i.e. satisfait les identités* (M_1) *à* (M_3)*.*

Preuve.

– (M_1) :

$$\begin{array}{rcll}(\mathsf{star}_0\ (\mathsf{unit}_0\ x)\ f) & = & (\mathsf{star}_{\epsilon_0,\epsilon_0}\ (\mathsf{unit}_{\bot,\epsilon_0}\ x)\ f) & \text{par déf.} \\ & = & (\mathsf{unit}_{\epsilon_0,\epsilon_0}\ (\mathsf{star}_{\bot,\epsilon_0}\ x\ f) & (G_6) \\ & = & (\mathsf{star}_{\bot,\epsilon_0}\ (f\ x)) & (G_2) \\ & = & (f\ x) & (G_3)\end{array}$$

– (M_2) :

$$\begin{array}{rcll}(\mathsf{star}_0\ m\ \lambda x.(\mathsf{unit}_0\ x)) & = & (\mathsf{star}_{\epsilon_0,\epsilon_0}\ m\ \lambda x.(\mathsf{unit}_{\bot,\epsilon_0}\ x)) & \text{par déf.} \\ & = & (\mathsf{unit}_{\epsilon_0,\epsilon_0}\ m) & (G_4) \\ & = & m & (G_2)\end{array}$$

– (M_3) :

$$\begin{array}{rcll}(\mathsf{star}_0\ m\ \lambda x.(\mathsf{star}_0\ (g\ x)\ f)) & = & (\mathsf{star}_{\epsilon_0,\epsilon_0}\ m\ \lambda x.(\mathsf{star}_{\epsilon_0,\epsilon_0}\ (g\ x)\ f)) & \text{par déf.} \\ & = & (\mathsf{star}_{\epsilon_0,\epsilon_0}\ (\mathsf{star}_{\epsilon_0,\epsilon_0}\ m\ g)\ f) & (G_5) \\ & = & (\mathsf{star}_0\ (\mathsf{star}_0\ m\ g)\ f) & \text{par déf.}\end{array}$$

□

Il existe une manière canonique d'étendre l'opérateur μ_ϵ en un opérateur μ sur les types τ et κ. Celui-ci se définit de la manière suivante :

$$\begin{array}{rcl}\mu((\tau_1, \ldots, \tau_n)c) & \stackrel{\text{def}}{=} & (\mu(\tau_1), \ldots, \mu(\tau_n))c \\ \mu((x:\tau) \to \kappa) & \stackrel{\text{def}}{=} & (x:\mu(\tau)) \to \mu(\kappa) \\ \mu(\tau, \epsilon) & \stackrel{\text{def}}{=} & \mu_\epsilon(\mu(\tau))\end{array}$$

On peut alors étendre également la définition de l'opérateur $\mathsf{unit}_{\epsilon_1,\epsilon_2}$ en deux opérateurs $\mathsf{unit}_{\tau_1,\tau_2} : \mu(\tau_1) \to \mu(\tau_2)$, lorsque $\tau_1 \sqsubseteq \tau_2$, et $\mathsf{unit}_{\kappa_1,\kappa_2} : \mu(\kappa_1) \to \mu(\kappa_2)$, lorsque $\kappa_1 \sqsubseteq \kappa_2$. Ces opérateurs se définissent de la manière suivante :

$$\begin{array}{lcl}
\mathsf{unit}_{\tau,\tau} & \stackrel{\text{def}}{=} & \lambda x.x \\
\mathsf{unit}_{\tau_1\to\kappa_1,\tau_2\to\kappa_2} & \stackrel{\text{def}}{=} & \lambda f.\lambda x.(\mathsf{unit}_{\kappa_1,\kappa_2}\ (f\ (\mathsf{unit}_{\tau_2,\tau_1}\ x))) \\
\mathsf{unit}_{(\tau_1,\epsilon_1),(\tau_2,\epsilon_2)} & \stackrel{\text{def}}{=} & \lambda x.(\mathsf{star}_{\epsilon_1,\epsilon_2}\ x\ \lambda v.(\mathsf{unit}_{\perp,\epsilon_2}\ (\mathsf{unit}_{\tau_1,\tau_2}\ v)))
\end{array}$$

De tels opérateurs expriment que tout objet de type τ_1 (resp. κ_1) peut être considéré comme un objet de type τ_2 (resp. κ_2) dès lors que $\tau_1 \sqsubseteq \tau_2$ (resp. $\kappa_1 \sqsubseteq \kappa_2$), c'est-à-dire que tout objet peut toujours être considéré comme ayant plus d'effets qu'il n'en a effectivement.

1.3.2 Traduction fonctionnelle

Nous sommes maintenant en mesure de définir la traduction fonctionnelle proprement dite. Dans un premier temps, nous définissons le langage cible de cette traduction. Nous aurions pu choisir le fragment purement fonctionnel du langage que nous avons introduit section 1.1. Mais nous avons préféré des notations plus proches de celles utilisées en Théorie des Types, formalisme qui sera à terme la cible de notre traduction. Dans un second temps, nous introduisons les outils de cette traduction, à savoir les opérateurs de la monade associée aux effets dont nous avons introduit l'analyse en section 1.2.1. Enfin, nous donnons la définition de la traduction, d'abord sur les types, puis sur les expressions de programmes.

Langage cible

À terme, le langage cible de notre traduction sera le Calcul des Constructions Inductives. Mais pour le moment, il suffit de choisir un λ-calcul typé polymorphe comprenant les constructions de types purement fonctionnelles de notre langage, les mêmes constantes purement fonctionnelles et un opérateur de point fixe, toujours noté Y. L'abstraction sera notée $\lambda x.e$ et l'abstraction par rapport à un type $\Lambda\alpha.e$.

Nous supposerons de plus que le langage cible comprend une notion primitive d'enregistrement, dont les types se notent $\{x_1 : \tau_1; \ldots; x_n : \tau_n\}$ et les éléments $\{x_1 = v_1; \ldots; x_n = v_n\}$. Si v est un tel enregistrement, alors $v.x$ dénote la valeur de son champ x. On suppose qu'il est possible d'abstraire un terme par rapport à une variable de champ, ainsi que d'appliquer un terme à une variable de champ, la réduction correspondante étant toujours la β-réduction[10]. On aura besoin par la suite de l'opération consistant à *mettre à jour* un enregistrement par rapport à un autre. Cette opération, notée $\oplus$, se définit formellement de la façon suivante :

Définition 24 *Si $x = \{x_1 = u_1; \ldots; x_n = u_n\}$ et $y = \{y_1 = v_1; \ldots; y_m = v_m\}$ alors $x \oplus y$ dénote l'enregistrement $\{z_1 = w_1; \ldots; z_k = w_k\}$ où $\{z_i\}_{i=1\ldots k} = \{x_i\}_{i=1\ldots n} \cup \{y_i\}_{i=1\ldots m}$ et où*

$$w_i = \begin{cases} v_j & \textit{si } z_i = y_j \\ w_j & \textit{si } z_i = x_j \textit{ et } z_i \notin \{y_j\} \end{cases}$$

[10] Cette supposition n'est pas très réaliste en pratique et nous expliquerons page 116 comment nous opérerons en pratique.

Enfin, nous supposerons le langage cible muni d'une sémantique semblable à celle du langage qui est l'objet de notre étude, c'est-à-dire d'*appel par valeur* où les abstractions sont considérées comme des valeurs — et où l'on ne réduit donc pas sous les abstractions. En effet, il est toujours possible d'écrire dans le langage cible des expressions dont l'évaluation ne termine pas, telles que $\Omega = (\mathsf{Y}\ \lambda f.\lambda x.(f\ x)\ \mathsf{void})$, et dès lors l'évaluation d'une expression telle que $(\lambda x.1\ \Omega)$ ne termine pas nécessairement suivant que l'on réduit l'application par valeur ou par nécessité. On choisit de conserver pour le langage cible la même stratégie d'évaluation que pour la partie fonctionnelle de notre langage de programmation — donnée page 18. Dès lors, on peut s'autoriser la notation $e_1 = e_2$, avec la signification suivante : « Les évaluations de e_1 et e_2 terminent et leurs formes normales sont égales ».

Cependant, on peut dès à présent garder à l'esprit que le langage cible deviendra à terme le Calcul des Constructions Inductives (voir chapitre 2) qui a la propriété de normalisation forte, laquelle nous garantit une évaluation qui termine et un résultat unique, quelle que soit la stratégie de réduction utilisée. (En contrepartie, il n'y a pas d'opérateur de point fixe général dans le Calcul des Constructions Inductives, et le prix à payer sera une justification explicite de la terminaison.)

Une monade pour les références

Nous allons appliquer ici les idées générales de la section 1.3.1 sur les monades au cas particulier de la notion d'effets que nous avons associée à notre langage dans la section 1.2. Nous avons introduit l'ensemble E des effets comme étant l'ensemble des couples (ρ, ω), où ρ et ω sont deux ensembles de variables tels que $\omega \subseteq \rho$. Nous avons désigné par $\bot$ l'élément $(\emptyset, \emptyset)$ (Définition 8 page 22) et muni E d'une opération $\sqcup$, pour laquelle $\bot$ est un élément neutre à gauche et à droite (Définition 9 page 22). Il nous reste donc, pour définir le semi-treillis des effets, à introduire une relation d'ordre $\sqsubseteq$ compatible avec $\sqcup$. Nous la définissons de la manière suivante :

$$(\rho_1, \omega_1) \sqsubseteq (\rho_2, \omega_2) \quad \stackrel{\text{def}}{=} \quad \rho_1 \subseteq \rho_2 \wedge \omega_1 \subseteq \omega_2$$

Dès lors, nous pouvons introduire la monade associée à cet effet. Suivant l'intuition que nous avons donnée jusqu'ici, la traduction d'un programme ayant un type τ et un effet (ρ, ω) sera une fonction prenant en arguments les valeurs des variables de ρ (l'entrée) et retournant d'une part une valeur de type τ (le résultat) et d'autre part les valeurs des variables de ω (la sortie). Les ensembles de valeurs d'entrée et de sortie seront représentés par des enregistrements.

Si Γ est un environnement bien formé contenant les références $x_i : \tau_i\ \mathsf{ref}$ alors l'opérateur de types μ de la monade est défini par

$$\mu_{(\rho,\omega)}(\tau) \quad \stackrel{\text{def}}{=} \quad \{x_{\phi(1)} : \tau_{\phi(1)}; \dots x_{\phi(k)} : \tau_{\phi(k)}\} \rightarrow \{x_{\psi(1)} : \tau_{\psi(1)}; \dots x_{\psi(m)} : \tau_{\psi(m)}\} \times \tau$$

où $\rho = \{x_{\phi(i)} \mid i = 1, \dots, k\}$ et $\omega = \{x_{\psi(i)} \mid i = 1, \dots, m\}$. Comme nous l'avons vu dans la section précédente, l'opérateur peut être étendu de manière canonique aux types τ et κ.

Pour définir les opérateurs unit et star, nous avons besoin d'une opération sur les enregistrements consistant à supprimer un certain nombre de champs — opération qui pourrait être rendue implicite en présence de sous-typage. Cette opération se définit formellement de la façon suivante :

$$[x]_{\{l_1,\dots,l_n\}} \quad \stackrel{\text{def}}{=} \quad \{l_1 = x.l_1; \dots; l_n = x.l_n\}$$

où x est un enregistrement comportant au moins les champs l_i. Les deux opérateurs unit et star sont alors définis de la manière suivante :

$$\begin{array}{lcl} \mathsf{unit}_{(\rho_1,\omega_1),(\rho_2,\omega_2)}\ m & \stackrel{\text{def}}{=} & \lambda x_2.\ \mathsf{let}\ (y_1, v_1) = (m\ [x_2]_{\rho_1})\ \mathsf{in} \\ & & \quad ([x_2 \oplus y_1]_{\omega_2}, v_1) \end{array}$$

et

$$\begin{array}{lcl} \mathsf{star}_{(\rho_1,\omega_1),(\rho_2,\omega_2)}\ m\ f & \stackrel{\text{def}}{=} & \lambda x.\ \mathsf{let}\ (y_1, v_1) = (m\ [x]_{\rho_1})\ \mathsf{in} \\ & & \quad \mathsf{let}\ (y_2, v_2) = (f\ v_1\ [x \oplus y_1]_{\rho_2})\ \mathsf{in} \\ & & \quad ([x \oplus y_1 \oplus y_2]_{\omega_1 \cup \omega_2}, v_2) \end{array}$$

La notion de monade généralisée nous permet d'introduire de manière rigoureuse et générale les éléments nécessaires à la traduction fonctionnelle des programmes impératifs. Mais son utilisation en pratique est assez lourde. Pour rendre le discours plus intuitif et plus clair, nous n'allons pas faire référence explicitement aux éléments de la monade par la suite, mais utiliser directement leurs définitions. Nous utiliserons en particulier une notation $\mathsf{let\ in}$ pour l'opérateur star, en se permettant de plus un certain abus de notation en omettant l'opérateur $[\_]$ — mais une fois encore cela se justifierait si nous disposions de sous-typage dans le langage cible.

Définition de la traduction

Nous commençons par redéfinir directement la traduction fonctionnelle des types et des environnements, en dépliant la définition de l'opérateur μ.

Définition 25 (traduction fonctionnelle des types) *Les traductions fonctionnelles des types de valeurs et des types de calculs, respectivement notées $\overline{\tau}$ et $\overline{\kappa}$, sont définies mutuellement récursivement, dans un environnement Γ, de la manière suivante :*

$$\begin{array}{llcll} \textit{valeurs} & \overline{(\tau_1,\dots,\tau_n)c} & = & (\overline{\tau_1},\dots,\overline{\tau_n})c & \\ & \overline{\alpha} & = & \alpha & \\ & \overline{\forall\alpha.\tau} & = & \forall\alpha.\overline{\tau} & \\ & \overline{(x : \tau\ \mathsf{ref}) \to \kappa} & = & \forall x.\overline{\kappa} & (x \textit{ est une variable d'étiquette}) \\ & \overline{(x : \tau) \to \kappa} & = & (x : \overline{\tau}) \to \overline{\kappa} & \textit{sinon} \\ & \overline{\tau\ \mathsf{ref}} & = & \overline{\tau} & \\ & & & & \\ \textit{calculs} & \overline{(\tau, (\rho, \omega))} & = & \overline{\rho} \to \overline{\omega} \times \overline{\tau} & \\ & & & & \\ \textit{effets} & \overline{\{x_1,\dots,x_n\}} & = & \{x_1 : \overline{\tau_1}; \dots; x_n : \overline{\tau_n}\} & \textit{où } x_i : \tau_i\ \mathsf{ref} \in \Gamma \end{array}$$

Si Γ est un environnement associant les variables x_i aux types τ_i, alors $\overline{\Gamma}$ dénote l'environnement associant les variables x_i aux types $\overline{\tau_i}$ pour les variables x_i n'étant pas des références.

Nous pouvons maintenant définir la traduction fonctionnelle des programmes eux-mêmes, avec l'invariant qu'une expression de programme de type κ aura une traduction fonctionnelle de type $\overline{\kappa}$.

Définition 26 (traduction fonctionnelle des programmes) *Soit Γ un environnement bien formé et e un programme tel que $\Gamma \vdash_r e : \kappa$. Alors la traduction fonctionnelle de e, notée $\overline{e}$, est un programme fonctionnel de type $\overline{\kappa}$ dans le contexte $\overline{\Gamma}$, défini par récurrence sur la structure de e de la manière suivante :*

$e \equiv c$ **:**

$$\overline{e} = c$$

$e \equiv x$ **:**

$$\overline{e} = x$$

$e \equiv (e_2\ e_1)$ **:** *On sait que $\Gamma \vdash_r e_2 : (x : \tau_1) \rightarrow^{\epsilon} \tau, \epsilon_2$. On distingue alors deux cas suivant le type de e_1 :*

– *Si $\Gamma \vdash_r e_1 : (\tau_1, \epsilon_1)$, avec $\tau_1 \neq$ _ ref, alors*

$$\begin{aligned}\overline{e} = \lambda x_0.\ & \mathsf{let}\ (x_1, a) = (\overline{e_1}\ x_0)\ \mathsf{in} \\ & \mathsf{let}\ (x_2, f) = (\overline{e_2}\ (x_0 \oplus x_1))\ \mathsf{in} \\ & \mathsf{let}\ (x_3, v) = (f\ a\ (x_0 \oplus x_1 \oplus x_2))\ \mathsf{in} \\ & (x_1 \oplus x_2 \oplus x_3, v)\end{aligned}$$

– *Si $e_1 = r : \tau_1'$ ref alors*

$$\begin{aligned}\overline{e} = \lambda x_0.\ & \mathsf{let}\ (x_1, f) = (\overline{e_2}\ x_0)\ \mathsf{in} \\ & \mathsf{let}\ (x_2, v) = (f\ r\ (x_0 \oplus x_1))\ \mathsf{in} \\ & (x_1 \oplus x_2, v)\end{aligned}$$

$e \equiv (e_1\ \tau)$ **:**

$$\overline{e} = (\overline{e_1}\ \overline{\tau})$$

$e \equiv \mathsf{fun}\ (x : \tau) \rightarrow e_1$ **:** *On distingue deux cas :*

– *Si $\tau = \tau_1$ ref alors*

$$\overline{e} = \lambda x.\overline{e_1}$$

où x désigne une variable de champ.

– *Sinon*

$$\overline{e} = \lambda x : \overline{\tau}.\overline{e_1}$$

$e \equiv \mathsf{Fun}\ \alpha \rightarrow e_1$ **:**

$$\overline{e} = \Lambda\alpha.\overline{e_1}$$

$e \equiv \mathsf{rec}\ f\ : \tau = e_1$ **:**

$$\overline{e} = (\mathsf{Y}\ \lambda f : \overline{\tau}.\overline{e_1})$$

$e \equiv \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$ **:**

$$\begin{array}{rl}\overline{e} = \lambda x_0. & \mathsf{let}\ (x_1, b) = (\overline{e_1}\ x_0)\ \mathsf{in} \\ & \mathsf{if}\ b\ \mathsf{then} \\ & \quad \mathsf{let}\ (x_2, v) = (\overline{e_2}\ (x_0 \oplus x_1))\ \mathsf{in}\ (x_0 \oplus x_1 \oplus x_2, v) \\ & \mathsf{else} \\ & \quad \mathsf{let}\ (x_2, v) = (\overline{e_3}\ (x_0 \oplus x_1))\ \mathsf{in}\ (x_0 \oplus x_1 \oplus x_2, v)\end{array}$$

$e \equiv \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ **:** *On distingue deux cas suivant le type de* e_1 *:*

- *Si* $\Gamma \vdash_r e_1 : (\tau_1, \epsilon_1)$ *avec* $\tau_1 \neq \_\ \mathsf{ref}$ *alors*

$$\begin{array}{rl}\overline{e} = \lambda x_0. & \mathsf{let}\ (x_1, x) = (\overline{e_1}\ x_0)\ \mathsf{in} \\ & \mathsf{let}\ (x_2, v) = (\overline{e_2}\ (x_0 \oplus x_1))\ \mathsf{in} \\ & (x_1 \oplus x_2, v)\end{array}$$

- *Si* $\Gamma \vdash_r e_1 : (\tau_1\ \mathsf{ref}, \epsilon_1)$ *alors*

$$\begin{array}{rl}\overline{e} = \lambda x_0. & \mathsf{let}\ (x_1, v_0) = (\overline{e_1}\ x_0)\ \mathsf{in} \\ & \mathsf{let}\ (x_2, v) = (\overline{e_2}\ (x_0 \oplus x_1 \oplus \{x = v_0\}))\ \mathsf{in} \\ & (x_1 \oplus x_2 \backslash x, v)\end{array}$$

$e \equiv e_1\ ;\ e_2$ **:** *On utilise l'équivalence*

$$e \approx \mathsf{let}\ \_ = e_1\ \mathsf{in}\ e_2$$

pour se ramener à la traduction d'un let *(avec* $e_1 : \mathsf{unit}$*).*

$e \equiv \mathsf{while}\ e_1\ \mathsf{do}\ e_2\ \mathsf{done}$ **:** *On utilise l'équivalence*

$$e \approx ((\mathsf{rec}\ w\ : \mathsf{unit} \rightarrow \mathsf{unit} = \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ ;\ (w\ \mathsf{void})\ \mathsf{else}\ \mathsf{void})\ \mathsf{void})$$

pour se ramener à des constructions pour lesquelles la traduction est déjà définie.

$e \equiv \mathsf{ref}\ e_1$ **:**

$$\overline{e} = \overline{e_1}$$

$e \equiv !x$ **:**

$$\overline{e} = \lambda u.\ u.x$$

$e \equiv x\ :=\ e_1$ **:**

$$\overline{e} = \lambda x_0.\mathsf{let}\ (x_1, v) = (\overline{e_1}\ x_0)\ \mathsf{in}\ (x_1 \oplus \{x = v\}, \mathsf{void})$$

□

Exemples. Reprenons les exemples introduits page 31 et donnons pour chacun sa traduction fonctionnelle.

Exemple 4 *Considérons tout d'abord* $ex_1 \equiv x := !x + 1$. *On avait*

$$\Gamma \vdash_r ex_1 : (\mathsf{unit}, (\{x\}, \{x\}))$$

donc $\overline{ex_1}$ *aura pour type* $\{x : \mathsf{int}\} \to \{x : \mathsf{int}\} \times \mathsf{unit}$. *Si l'on suit la définition 26 — en simplifiant directement la traduction de l'expression* $!x + 1$ *— on obtient*

$$\overline{ex_1} \quad = \quad \lambda u.\ \mathsf{let}\ v = u.x + 1\ \mathsf{in}\ (\{x = v\}, \mathsf{void})$$

que l'on peut simplifier en effectuant la réduction sous l'abstraction, ce qui donne

$$\overline{ex_1} \quad = \quad \lambda u.(\{x = u.x + 1\}, \mathsf{void})$$

Si l'on choisit de représenter les enregistrements n'ayant qu'un seul champ directement par la valeur de ce champ alors on a tout simplement

$$\overline{ex_1} \quad = \quad \lambda x.(x + 1, \mathsf{void})$$

qui est le programme que l'on aurait écrit directement. □

Exemple 5 *Considérons maintenant la fonction* $ex_2 \equiv \mathsf{fun}\ (r : \mathsf{int\ ref}) \to r := !r + 1\ ;\ !r$. *Son argument* r, *qui est une référence, devient une variable de champ, ce qui donne le programme fonctionnel suivant :*

$$\begin{array}{rcl} \overline{ex_2} & = & \lambda r.\lambda u_0.\ \mathsf{let}\ (u_1, \_) = (\{r = u_0.r + 1\}, \mathsf{void})\ \mathsf{in} \\ & & \mathsf{let}\ v = u_1.r\ \mathsf{in} \\ & & (u_1, v) \end{array}$$

soit après réductions :

$$\overline{ex_2} \quad = \quad \lambda r.\lambda u.(\{r = u.r + 1\}, u.r + 1)$$

□

Exemple 6 *Le troisième programme est* $ex_3 \equiv y := (ex_2\ x)$. *Sa traduction est donc :*

$$\begin{array}{rcl} \overline{ex_3} & = & \lambda u_0.\ \mathsf{let}\ (u_1, v) = \\ & & \phantom{\lambda u_0.\ \mathsf{let}\ } \mathsf{let}\ (u_1, v) = (\overline{ex_2}\ x\ u_0)\ \mathsf{in}\ (u_1, v) \\ & & \mathsf{in}\ (\{x = u_1.x; y = v\}, \mathsf{void}) \end{array}$$

soit après réductions :

$$\overline{ex_3} \quad = \quad \lambda u_0.\mathsf{let}\ (u_1, v) = (\overline{ex_2}\ x\ u_0)\ \mathsf{in}\ (\{x = u_1.x; y = v\}, \mathsf{void})$$

On notera comment l'argument réel x *est passé à la fonction* ex_2 *en lieu et place de son argument formel* r, *par application de* $\overline{ex_2}$ *à la variable d'étiquette* x. □

Des exemples plus conséquents seront donnés au chapitre 2, section 2.3.4 page 85.

Pour établir la correction de la traduction fonctionnelle dans la section suivante, nous aurons besoin d'un lemme de substitution, qui est à la traduction fonctionnelle ce qu'est l'auto-réduction au typage.

Lemme 7 (lemme de substitution) *Soit Γ un environnement bien formé et e un programme bien typé dans Γ. Si $z : \tau \in \Gamma$ avec $\tau \neq$ _ ref et si v est une valeur de type τ, alors*

$$\overline{e[z \leftarrow v]} = \overline{e}[z \leftarrow \overline{v}]$$

Preuve. La preuve se fait par récurrence sur la structure de e.

- $e \equiv x$
  - si $x = z$ alors $\overline{e[z \leftarrow v]} = \overline{v} = z[z \leftarrow \overline{v}]$;
  - si $x \neq z$ alors $\overline{e[z \leftarrow v]} = \overline{x} = x$ et $x[z \leftarrow \overline{v}] = x$.
- $e \equiv c$: immédiat.
- $e \equiv\ !x$: par hypothèse z n'est pas une référence, donc $z \neq x$.
- $e \equiv \mathsf{ref}\ e_1$:

$$\begin{array}{rcll} \overline{e[z \leftarrow v]} & = & \overline{\mathsf{ref}\ e_1[z \leftarrow v]} & \\ & = & \overline{e_1[z \leftarrow v]} & \\ & = & \overline{e_1}[z \leftarrow \overline{v}] & \text{par hypothèse de récurrence} \\ & = & \overline{e}[z \leftarrow \overline{v}] & \end{array}$$

- $e \equiv x := e_1$:

$$\begin{array}{rcl} \overline{e[z \leftarrow v]} & = & \overline{x := e_1[z \leftarrow v]} \\ & = & \lambda x_0.\mathsf{let}\ (x_1, v_1) = (\overline{e_1[z \leftarrow v]}\ x_0)\ \mathsf{in}\ (x_1 \oplus \{x = v_1\}, \mathsf{void}) \\ & = & \lambda x_0.\mathsf{let}\ (x_1, v_1) = (\overline{e_1}[z \leftarrow \overline{v}]\ x_0)\ \mathsf{in}\ (x_1 \oplus \{x = v_1\}, \mathsf{void}) \\ & & \qquad\qquad \text{par hypothèse de récurrence} \\ & = & \overline{e}[z \leftarrow \overline{v}] \end{array}$$

- $e \equiv \mathsf{fun}\ x \rightarrow e_1$: on peut toujours supposer que $x \neq z$ — sinon il suffit de renommer x. Dès lors, ce cas se traite de façon similaire aux cas précédents.
- $e \equiv \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$: on peut supposer que $x \neq z$, et ce cas se traite alors comme les précédents.
- $e \equiv \mathsf{rec}\ f\ \ : \tau = e_1$: là encore, on peut supposer que $f \neq z$, et ce cas se traite alors comme les précédents.
- $e \equiv (e_2\ e_1)$: on distingue là deux cas, suivant que e_1 est ou non une référence.
  - $e_1 : \tau_1 \neq$ _ ref : se traite comme les cas précédents, en prenant soin de choisir dans la traduction une variable a différente de z (voir la définition de la traduction d'une application).

– $e_1 = r : \tau_1$ ref : on a dans ce cas $e_2 : (x : \tau_1 \text{ ref}) \to^{\epsilon} \tau_2$ et

$$\begin{array}{rcl}\overline{e[z \leftarrow v]} & = & \lambda x_0.\ \mathsf{let}\ (x_1, f) = (\overline{e_2[z \leftarrow v]}\ x_0)\ \mathsf{in} \\ & & \mathsf{let}\ (x_2, v_2) = (f\ r\ (x_0 \oplus x_1))\ \mathsf{in} \\ & & (x_1 \oplus x_2, v_2) \\ & = & \lambda x_0.\ \mathsf{let}\ (x_1, f) = (\overline{e_2}[z \leftarrow \overline{v}]\ x_0)\ \mathsf{in} \\ & & \mathsf{let}\ (x_2, v_2) = (f\ r\ (x_0 \oplus x_1))\ \mathsf{in} \\ & & (x_1 \oplus x_2, v_2)\end{array}$$

Or z n'est pas une référence, donc z n'apparaît pas dans le type de f, et en particulier pas dans τ_2, le type de la valeur v_2, et l'expression ci-dessus est donc bien égale à $\overline{e}[z \leftarrow \overline{v}]$.

□

1.3.3 Correction

Nous allons maintenant montrer que la traduction fonctionnelle que nous venons de définir est *sémantiquement correcte*. Il nous faut bien sûr commencer par définir cette notion, car nous avons d'un côté des programmes impératifs et de l'autre des programmes purement fonctionnels. Une première idée consiste à exprimer seulement que, pour tout programme e, alors e et $\overline{e}$ exécutés sur un même état d'entrée donnent le même résultat. En effet, il n'est pas nécessaire de parler de l'état de sortie car on peut toujours considérer, pour toute référence r, le programme let _ $= e$ in $!r$, qui aura le même résultat que sa traduction si et seulement si la valeur de sortie de r est la même à l'issue de e et de $\overline{e}$. Néanmoins, il est plus facile d'établir directement que e et $\overline{e}$ exécutés sur le même état d'entrée donnent le même résultat et le même état de sortie, ce que l'on exprimera par le théorème 1.

Mais afin de pouvoir énoncer ce théorème, il nous faut définir comment interpréter le résultat de l'évaluation d'un programme, telle qu'elle est définie dans la section 1.1.4. Ce résultat peut être en effet de la forme $\theta.x$ avec $x \in dom(\theta)$ si le programme évalué est une référence, et une telle variable x n'a pas de contrepartie directe dans le monde fonctionnel. Puisque les seules expressions de type τ ref ne sont que des références nouvellement créées, il suffit donc d'interpréter une telle variable x par la valeur qui lui est associée dans l'état θ.

Définition 27 (interprétation des valeurs) *L'interprétation d'une valeur v dans un état θ, notée $val(\theta, v)$, est définie de la manière suivante :*

$$\begin{cases} val(\theta, x) = \theta(x) & \textit{si } x \in dom(\theta) \\ val(\theta, v) = v & \textit{sinon} \end{cases}$$

La correction de la traduction fonctionnelle peut alors se formuler ainsi :

Théorème 1 (correction) *Soit Γ un environnement bien formé dont les références sont $x_1 : \tau_1$ ref$, \dots, x_n : \tau_n$ ref et e un programme tel que $\Gamma \vdash_r e : (\tau, (\rho, \omega))$. Soient v_i des valeurs de types τ_i et θ l'état associant les v_i aux x_i. Alors*

$$\forall \theta', v.\quad \theta.e \longmapsto^{\star} \theta'.v \quad \Longrightarrow \quad (\overline{e}\ \overline{\theta(\rho)}) = (\overline{\theta'(\omega)}, \overline{val(\theta', v)})$$

PREUVE. Nous rappelons que la notion d'égalité sur les programmes fonctionnels est définie page 47. La preuve se fait par récurrence sur la longueur de la dérivation $\longmapsto^\star$ et par cas sur la structure de e. La preuve complète de ce théorème est donnée en annexe, page 157. □

Il ne suffit pas cependant de montrer que si un programme e s'évalue en v alors sa traduction fonctionnelle s'évalue bien sur le même résultat. Il faut également montrer la réciproque, à savoir que les calculs effectués par la traduction fonctionnelle — ceux sur lesquels on va raisonner — correspondent bien aux mêmes calculs effectués par le programme de départ. C'est-à-dire qu'il nous faut montrer la réciproque du théorème 1. Vu que ce dernier établit que la traduction fonctionnelle calcule bien la même chose que le programme impératif de départ, il suffit de montrer que le programme impératif termine dès lors que l'évaluation de sa traduction fonctionnelle termine. Cela peut s'énoncer de la manière suivante :

Théorème 2 (réciproque) *Soit Γ un environnement bien formé dont les références sont $x_1 : \tau_1$ ref$, \ldots, x_n : \tau_n$ ref et e un programme tel que $\Gamma \vdash_r e : (\tau, (\rho, \omega))$. Alors pour tout état θ associant des valeurs aux variables x_i on a*

$$\forall y, v_0. \quad (\overline{e}\ \overline{\theta(\rho)}) = (y, v_0) \quad \Longrightarrow \quad \exists \theta', v.\ \theta.e \longmapsto^\star \theta'.v$$

PREUVE. La preuve détaillée est donnée en annexe, page 162. □

1.4 Conclusion

Arrivés au terme de ce premier chapitre, nous allons faire un bilan rapide de la traduction fonctionnelle que nous venons de présenter, avant de poursuivre vers une méthodologie de preuve de programmes. Nous commencerons par énumérer les aspects positifs et négatifs de notre approche, avant de la comparer à des travaux similaires.

Aspects positifs

Les aspects positifs de notre traduction fonctionnelle résident dans sa relative généralité et dans son caractère modulaire.

Généralité de la traduction. La traduction fonctionnelle que nous avons définie a été introduite avec autant de généralité que possible. Nous avons introduit une notion générale d'effet (page 37) et les règles de typage associées (page 38) pour les constructions les plus générales des langages de programmation (conditionnelle, séquence, boucle, appel de fonction, etc.), indépendamment de l'effet proprement dit. Puis nous avons introduit une notion générale de monade (page 42), associée à un effet quelconque, sur laquelle nous avons bâti notre traduction fonctionnelle. Si nous avons instancié ces notions générales dans le cas précis des références, ce n'était que pour rendre le discours plus concret, et préparer la mise en pratique qui sera illustrée au chapitre 3.

En revanche, notre définition de la traduction fonctionnelle n'est pas paramétrée par la sémantique du langage de programmation considéré. Cela aurait été assez difficile, dans la mesure où notre définition des types de valeurs et de calculs (pages 23, 37) repose sur le caractère *strict* choisi pour la sémantique de notre langage. En effet, le type d'une fonction est de la forme $\tau \to \kappa$ où τ désigne un type de *valeur* et κ un type de *calcul*. Avec une sémantique d'appel par nom, on aurait un type de la forme $\kappa \to \kappa$. La plupart des définitions (passées et à venir) devraient alors être modifiées, car reposant sur la structure des types τ et κ.

Modularité. Le principal atout de notre traduction est son caractère *modulaire*. En effet, seule l'information de type avec effets d'un objet (constante ou fonction) permet la traduction des définitions qui en dépendent. De même que dans un langage de programmation modulaire les informations d'une interface permettent la compilation séparée de toute unité en dépendant, les types avec effets permettent la « traduction séparée ». Ainsi, si l'interface d'un module m déclare une référence entière r et une fonction f de type $\mathsf{int} \to (\mathsf{unit}, (\{r\}, \{r\}))$, alors on sait que la traduction fonctionnelle de f aura le type $\mathsf{int} \to \mathsf{int} \to \mathsf{int} \times \mathsf{unit}$, sans avoir besoin de connaître la *définition* de f. Et l'on peut traduire tout programme e utilisant la fonction f.

Il peut sembler, cependant, que la déclaration de fonctions avec leurs types annotés d'effets impose la déclaration simultanée des références dont elles dépendent, ce qui irait à l'encontre du principe d'encapsulation associé à tout langage de modules. Dans l'exemple ci-dessus, il semble nécessaire de déclarer forcément la référence r dans l'interface du module m. Mais il s'agit seulement de distinguer deux domaines de visibilité, celui de la programmation, et celui de la traduction fonctionnelle — qui sera plus tard celui de la preuve. La référence r ci-dessus peut très bien être abstraite du point de vue de la programmation, et visible en ce qui concerne la traduction. Pour plus de modularité encore, on pourrait envisager de pouvoir déclarer des objets mutables abstraits, auxquels seraient associés des types abstraits dans la traduction fonctionnelle. Ainsi la référence r ci-dessus pourrait n'être déclarée que comme un objet mutable abstrait, associé à un type abstrait R. À la fonction f, toujours déclarée avec le même type, serait alors associée une traduction fonctionnelle $\overline{f}$ de type $\mathsf{int} \to R \to R \times \mathsf{unit}$.

La modularité est un point essentiel, que nous ferons en sorte de conserver par la suite, de manière à obtenir une méthode de preuve de programmes elle-même modulaire.

Aspects négatifs

Les aspects négatifs de notre approche sont tous deux des conséquences de la finesse que nous avons recherchée dans notre traduction — et qui ne sera vraiment justifiée que dans le chapitre suivant.

Absence de polymorphisme d'effets. La principale limite de notre approche est l'absence de *polymorphisme d'effets*, c'est-à-dire l'impossibilité de généraliser le type d'une fonction par rapport à un effet — ce que l'on peut faire au contraire dans le formalisme de J.-P. Talpin et P. Jouvelot. On peut cependant « justifier » cette limitation, progressivement, de la façon suivante. Considérons tout d'abord la fonction d'ordre supérieur F définie par :

$$\mathsf{let}\ F\ f = (f\ 0)$$

Elle a pour type $\forall\alpha.(\mathsf{int} \rightarrow \alpha) \rightarrow \alpha$ et son type avec effets serait le suivant

$$F \quad : \quad \forall\alpha.\forall\epsilon.(\mathsf{int} \rightarrow^{\epsilon} \alpha) \rightarrow^{\epsilon} \alpha$$

en supposant que l'on peut quantifier sur une variable d'effet ϵ, à la manière de ce qui est fait dans *The Type and Effect Discipline* [55]. En instanciant par exemple F sur une fonction de type $\mathsf{int} \rightarrow^{\{r\},\{r\}} \mathsf{unit}$, on obtient un calcul de type $(\mathsf{unit}, (\{r\}, \{r\}))$, par simple instanciation de la variable ϵ. Jusque là, il n'y a pas d'obstacle a priori au polymorphisme d'effets. Mais la question est alors la suivante : quel est le type de $\overline{F}$? En toute généralité, il devrait être le suivant :

$$\forall\alpha.\ \forall\epsilon.\ (\mathsf{int} \rightarrow \mu_{\epsilon}(\alpha)) \rightarrow \mu_{\epsilon}(\alpha)$$

où μ est l'opérateur de types de la monade considérée. Comme μ_{ϵ} ne peut plus être calculé statiquement, il faut donc définir *explicitement* l'opérateur μ dans le domaine cible de la traduction, et donc la notion d'effet également. Et de même, pour traduire la définition de F il faudra expliciter la définition des opérateurs star et unit.

D'autre part, considérons la fonction d'ordre supérieur G suivante :

$$\mathsf{let}\ G\ f = (f\ r)$$

où r est par exemple une référence entière. Le type de G est $\forall\alpha.(\mathsf{int\ ref} \rightarrow \alpha) \rightarrow \alpha$, et son type avec effets devrait être le suivant :

$$G \quad : \quad \forall\alpha.\forall\epsilon.((x : \mathsf{int\ ref}) \rightarrow^{\epsilon} \alpha) \rightarrow^{\epsilon[x \leftarrow r]} \alpha$$

où la substitution $\epsilon[x \leftarrow r]$ ne peut être réalisée qu'à chaque instanciation de l'effet ϵ. On voit alors qu'il faut également expliciter la substitution sur les effets. De la même manière, on peut montrer que les différentes opérations sur les effets, telle que $\sqcup$, devraient elles aussi être rendues explicites.

Il ressort de cette analyse que le prix à payer pour avoir du polymorphisme d'effets serait une *explicitation* totale de la notion d'effet et de la monade associée. Si l'on prend le cas des références, cela impliquerait la manipulation de types dépendants assez complexes et les traductions fonctionnelles deviendraient vite très lourdes. C'est pourquoi j'ai préféré des notions *externes* d'effet et de monade. Conséquemment, on ne dispose plus que d'une forme très pauvre de polymorphisme, essentiellement réduite aux objets purement fonctionnels.

Cependant, il faut noter que même avec un système de types avec effets polymorphe comme celui évoqué ci-dessus, nous ne pourrions pas donner de type général à la fonction `map`, qui à une fonction f et une liste $l = [x_1; \dots; x_n]$ associe la liste $[f(x_1); \dots; f(x_n)]$. En effet, on imagine que son type serait de la forme

$$\mathtt{map} \quad : \quad \forall\alpha\beta.\forall\epsilon.(\alpha \rightarrow^{\epsilon} \beta) \rightarrow (\alpha\ \mathsf{list}) \rightarrow^{\epsilon} (\beta\ \mathsf{list})$$

Mais la notion d'effet que nous avons introduite est trop fine pour pouvoir être ainsi généralisée. Dans le cas de la fonction `map`, l'effet résultant de l'application de cette fonction à deux arguments f et l dépend non seulement du type de f, comme nous avons essayé de l'écrire ci-dessus, mais également de la *valeur* de l. En effet, si `incr` désigne la fonction de type int ref $\rightarrow$ unit qui incrémente une référence entière, alors on peut considérer par exemple l'expression :

$$(\texttt{map incr}\ [r_1; r_2; r_3])$$

où r_1, r_2, r_3 sont des références, et son type avec effet est $(\textsf{unit list}, (R, R))$ où $R = \{r_1, r_2, r_3\}$. De manière plus générale on peut considérer la fonction (`map incr`), qui à une liste de références entières l associe une liste de valeurs void de même longueur, après avoir incrémenté toutes les références de l. Le type de cette fonction dépend de la valeur de son argument, et ne peut donc plus être déterminé statiquement. On se rend donc compte que même instanciée sur une fonction particulière, comme `incr`, la fonction `map` ne peut se voir attribuer un type avec effets général. Cela est dû, en particulier, aux alias que l'on peut engendrer à partir d'une liste de références.

Exclusion des alias. Afin de rendre possible une analyse statique des effets et de ne pas recourir à la manipulation d'un état global, nous avons exclu les alias de nos programmes (voir page 23). Nous l'avons fait par des restrictions syntaxiques, couplées à des restrictions dans les règles de typage — voir la règle ($\textsf{APPREF}_r$) page 27 et les règles ($\textsf{VAR'}_r$) et ($\textsf{LETREF'}_r$) page 31.

Nous verrons dans le chapitre 3, dans la section 3.3, que la manipulation des tableaux est également une source d'alias dans les programmes. Nous adopterons la solution classique consistant à voir un tableau comme une unique référence sur un tableau fonctionnel (voir [10], page 931, pour une bibliographie sur la question). Ce problème n'est pas propre aux tableaux, mais à toute structure de données contenant des références *en profondeur*. La solution consisterait alors, de manière analogue à celle que nous utiliserons pour les tableaux, à voir une telle structure comme une unique référence sur la même structure de donnée fonctionnelle.

Cependant, il faut noter que les alias sont peu présents dans les programmes considérés en pratique. Du seul fait de restrictions syntaxiques, le langage Pascal ne permet pas, en l'absence de manipulation explicite de pointeurs, d'engendrer des alias — autrement que par application d'une fonction à plusieurs références identiques, ce qui est critère syntaxique aisément décidable. C'est le cas également pour le langage FORTRAN, si l'on exclut sa directive COMMON. D'autre part, l'utilisation d'un GC, qui évite dans de très nombreux cas la manipulation explicite de pointeurs, réduit considérablement les cas possibles d'aliasing.

Autres approches

Pour l'essentiel, les travaux consistant à traduire des programmes impératifs par des fonctions sont des travaux de sémantique, que l'on classifie plus précisément de sémantique dénotationnelle. C'est en un sens ce que nous venons de faire, dans la mesure où la méthode de preuve de programmes que nous allons développer dans le chapitre suivant va s'appuyer sur la traduction fonctionnelle, mais il s'agira toujours d'exécuter les programmes impératifs de départ. La traduction fonctionnelle n'aura donc servi qu'à exprimer la sémantique du programme impératif.

Si nous ne sommes pas parti de travaux existants en sémantique dénotationnelle, c'est parce que ceux-ci voient les programmes impératifs comme des fonctions manipulant des images de la mémoire i.e. prenant en argument une valeur représentant l'état actuel de la mémoire et retournant comme résultat une valeur représentant l'état de cette même mémoire à l'issue du calcul. Une telle vision des programmes impératifs convient parfaitement pour établir des résultats théoriques sur les programmes, mais s'avère impraticable en ce qui concerne la preuve de programmes. En effet, on doit alors exprimer dans la post-condition de chaque fonction les éléments de la mémoire globale qui n'ont pas été modifiés, et faire intervenir ces informations dans l'établissement des obligations de preuve. Avec notre analyse fine des effets, nous parvenons à traduire les programmes impératifs ne manipulant que les valeurs intervenant réellement dans le calcul, et les spécifications comme les obligations de preuve s'en verront amplement simplifiées, comme nous le montrerons au chapitre suivant.

Toutefois, il nous faut citer l'article de P. W. O'Hearn et J. C. Reynolds intitulé *From Algol to Polymorphic Linear Lambda-calculus* [44]. Comme son titre l'indique, les auteurs y définissent une traduction des programmes Algol vers un λ-calcul polymorphe linéaire. Ce travail a pour but de donner une sémantique aux programmes Algol et la traduction utilise un état global. Ce travail tombe donc dans la catégorie précédemment évoquée. Cependant, il possède deux avantages sur les travaux classiques de sémantique dénotationnelle. D'une part, les auteurs s'attaquent à un langage réaliste — même si leur version d'Algol est légèrement simplifiée par endroits — et traitent en particulier des constructions fonctionnelles d'une manière simple et directe. Bien que manipulant un état global, leur traduction est par endroit quasi identique à la nôtre. D'autre part, et c'est là toute l'originalité de cet article, les auteurs montrent que la traduction peut être typée dans un λ-calcul polymorphe *linéaire*, exprimant ainsi directement que l'on ne peut modifier deux fois un même état mémoire. Malheureusement, ce n'est pas le cas de notre traduction fonctionnelle, car du fait que nous ne manipulons pas d'état global, mais à chaque point de programme un ensemble de valeurs représentant un nombre fini de références, l'accès au contenu d'une référence peut faire intervenir une valeur arbitrairement haut dans la traduction, et les valeurs des références peuvent donc intervenir *plusieurs fois* après leur introduction.

Enfin, il faut citer les travaux de P. Wadler [30, 56, 57] qui ont permis d'introduire des traits impératifs, notamment des entrées-sorties, dans le langage purement fonctionnel Haskell à l'aide de monades. C'est sur cette notion de monade que nous nous sommes appuyés pour définir notre traduction fonctionnelle. Le travail de P. Wadler constitue l'un des rares exemples d'application pratique de la traduction fonctionnelle de programmes impératifs.

Chapitre 2

Preuve de programmes

« Le rêve d'hier est l'espoir d'aujourd'hui
et la réalité de demain. »
R. H. Goddard

Au chapitre précédent, nous avons présenté une traduction fonctionnelle de programmes impératifs. Elle ne constituait pas une finalité en elle-même mais une étape vers le but de notre travail : une méthode pour établir la *correction* et la *terminaison* de programmes. Ce deuxième chapitre va présenter cette méthode.

Contrairement à ce que l'on pourrait croire, ce n'est pas le langage ou le fragment de langage qu'une méthode sait traiter qui est le plus important — sauf s'il s'agit d'un langage manifestement trop limité — mais son langage de spécification et ses outils de preuve. Dans notre cas le langage de spécification sera le *Calcul des Constructions Inductives* [9, 48, 50] et l'outil de preuve sera en pratique le système Coq [1], dans lequel nous avons implanté notre méthode. Nous tenterons de justifier ces choix, s'il en est besoin, mais le plus important est d'en saisir les conséquences : nous ne définirons pas de langage de spécification particulier — du moins en ce qui concerne les parties fonctionnelles des spécifications — ni ne chercherons à construire de procédure de preuve automatique de quelque sorte que ce soit.

Nous pensons en effet que nous y gagnons à séparer clairement le langage de spécification, la méthodologie et les outils de preuve. En rendant notre méthodologie relativement indépendante du premier et du troisième, nous bénéficions de leurs progrès « gratuitement », sans avoir à modifier notre théorie, alors qu'une méthodologie — et une implantation — trop *ad hoc* devraient être modifiées perpétuellement. Ainsi, si les prochaines versions de Coq se voient dotées de définitions récursives ou de nouvelles tactiques de recherche de preuve plus puissantes alors notre implantation en bénéficiera sans aucune modification.

Ce chapitre sera organisé comme ceci. Nous commencerons par analyser diverses méthodes de preuve de programmes, dont la logique de Floyd-Hoare, ainsi que diverses formalisations de ces méthodes dans la Theorie des Types, afin d'en dégager les avantages comme les inconvénients. Nous introduirons ensuite dans la section 2.2 les définitions de spécification et d'annotations de programmes. Puis viendra dans la section 2.3 la méthode de preuve de programmes elle-même, avec sa preuve de correction et un examen détaillé des obligations de preuve qu'elle engendre. Enfin, nous discuterons de complétude dans la section 2.4.

2.1 Logique de Floyd-Hoare et autres systèmes

On ne peut parler de preuve de programmes sans parler de logique de Hoare, et ce pour deux raisons : d'une part parce que ce fut historiquement la première méthode de preuve de programmes, et d'autre part parce qu'elle est resté jusqu'à aujourd'hui la plus naturelle. Nous parlerons en fait de *logique de Floyd-Hoare*, suivant par là l'exemple de M. Gordon [18]. On peut en effet considérer que les premières idées d'une logique axiomatique pour établir la correction de programme sont autant dans l'article de R. W. Floyd *Assigning meanings to programs* [13] que dans celui de C. A. R. Hoare *An axiomatic basis for computer programming* [24], bien que ce dernier soit effectivement le plus célèbre.

La logique de Floyd-Hoare a été à l'origine d'un très grand nombre d'articles et d'ouvrages, dont il est impossible de faire ici la présentation. Nous renvoyons le lecteur intéressé au chapitre du *Handbook of Theoretical Computer Science* écrit par P. Cousot et intitulé *Methods and Logics for Proving Programs* [10]. Une grande partie de ce chapitre est consacrée à la logique de Floyd-Hoare et à ses extensions, et la bibliographie qui l'accompagne est très complète. Nous allons nous contenter ici de donner les grandes idées de la logique de Floyd-Hoare et d'en dégager certains avantages.

La logique de Floyd-Hoare introduit la notion de *formule de correction* et définit un système formel permettant d'établir cette formule. La formule de correction se présente sous la forme :

$$\{P\}\ e\ \{Q\}$$

où e est un programme et P et Q des propositions logiques, appelées respectivement précondition et post-condition, portant sur les variables du programme. Si l'on s'intéresse à la correction *totale*, l'interprétation de cette formule est alors la proposition suivante : « Pour tout état des variables du programme vérifiant la proposition P, alors l'évaluation du programme e se termine et l'état final des variables vérifie la proposition Q ». On peut formaliser cette définition dès lors que l'on se donne une sémantique formelle du langage. La logique de Floyd-Hoare est alors un système axiomatique permettant d'établir la formule ci-dessus par un certain nombre de règles d'inférence, basées sur la structure du programme e. La dérivation d'une formule $\{P\}\ e\ \{Q\}$ dans ce système sera notée $\vdash \{P\}\ e\ \{Q\}$. Ainsi, la règle concernant l'affectation pourra être de la forme :

$$\frac{P \Rightarrow Q[x \leftarrow E]}{\vdash \{P\}\ x := E\ \{Q\}}$$

où E désigne une expression purement fonctionnelle. De même, la règle concernant la conditionnelle sera la suivante :

$$\frac{\vdash \{P \wedge B\}\ e_1\ \{Q\} \qquad \vdash \{P \wedge \neg B\}\ e_2\ \{Q\}}{\vdash \{P\}\ \mathsf{if}\ B\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\ \{Q\}}$$

où B est une expression booléenne purement fonctionnelle. Notre but n'est pas de redéfinir ici toute la logique de Floyd-Hoare, mais seulement d'en donner les principales idées.

Le résultat qui justifie la logique de Floyd-Hoare est la propriété de *correction* qui exprime que toute formule de correction dérivée dans le système formel est valide, ce qui s'écrit formellement de la façon suivante :

$$\vdash \{P\}\ e\ \{Q\} \quad \Longrightarrow \quad \models \{P\}\ e\ \{Q\}$$

où le jugement $\models$ désigne la validité logique. Une question légitime est alors de savoir si la réciproque est vraie, à savoir si l'on peut dériver dans le système formel toute formule de correction valide. C'est ce que l'on appelle la propriété de *complétude*. Cette propriété n'est pas vérifiée en général car l'ensemble des formules dérivables dans la logique de Floyd-Hoare est récursivement énumérable alors que l'ensemble des formules de correction valides ne l'est pas. (Pour plus de détails, voir la section 7.4.2 de [10] consacrée aux problèmes de complétude.) Nous reparlerons du problème de la complétude dans la section 2.4.

Telle qu'elle fut présentée à la fin des années 60, la logique de Floyd-Hoare était une méthode *mathématique*, en cela que les assertions étaient écrites dans le formalisme mathématique, et les obligations de preuve prouvées « manuellement » avec l'appareil mathématique traditionnel. C'est ainsi par exemple que C. A. R. Hoare donna dans un article de 1971 la preuve d'un programme, à la manière d'un résultat de mathématique [25]. Nombreux furent ceux qui le suivirent dans cette voie (voir par exemple [19] ou [7], et la collection d'exemples donnée à la fin de [10]). Si elle n'était pas une méthode assistée par un outil mécanique, la logique de Floyd-Hoare n'en a pas moins permis une meilleure compréhension des langages de programmation. Et elle a surtout formalisé la notion d'*assertion* dans un programme, donnant un statut mathématique à ce qui n'était alors que des commentaires dans le meilleur des cas.

L'idée d'une méthode complètement formelle est plus récente, car elle nécessite deux étapes supplémentaires : la définition d'un *langage formel de spécification* et la mise en place d'un *outil de preuve*, qu'il soit automatique ou interactif. Les langages de spécification sont apparus les premiers, avec le langage Z [21] et les spécifications algébriques. Puis sont apparues des méthodes formelles proprement dites, telles que VDM [27, 28], la méthode B de J. R. Abrial [4] ou encore la méthode KIV [51] développée à l'université de Karlsruhe. Si ces méthodes sont assez différentes, notamment dans leur manière de spécifier les programmes, elles ont tout de même en commun un héritage de la logique de Floyd-Hoare : partant d'un programme annoté, ou d'un raffinement d'une spécification, ces méthodes engendrent un certain nombre d'obligations de preuve, déléguées alors à un prouveur assisté par l'utilisateur. La justification de ces méthodes a un caractère *méta* : elle repose sur un résultat de correction de nature mathématique — quand celui-ci existe. Ce qu'il est important de signaler ici, c'est que ces méthodes viennent avec leur propre formalisme logique et leur propre prouveur, développés de façon *ad hoc*. (Le lecteur désirant découvrir plus avant ces différentes méthodes pourra consulter le livre de J. F. Monin *Comprendre les méthodes formelles* [40].)

Parallèlement se sont développées des *logiques formelles*, sous la forme d'assistants de preuve. On peut citer par exemple les systèmes Nqthm, PVS, Coq, LEGO ou encore HOL. Ces outils permettent de définir des types et des propositions, d'énoncer des théorèmes et de prouver ces derniers de manière essentiellement interactive — mais avec l'aide d'un système automatique lorsque les buts deviennent véritablement simples. Ces systèmes implantent des logiques d'ordre supérieur très expressives, dans lesquelles il est possible de représenter une grande partie des mathématiques traditionnelles.

Mais paradoxalement, ces systèmes n'ont pas, ou peu, été utilisés comme support pour le développement de méthodes formelles de preuve de programmes. Plus précisément, ces systèmes ont surtout servi jusqu'à présent à *valider* certaines méthodes formelles. Ainsi, on peut citer les travaux de P. Chartier qui a formalisé les machines B dans Isabelle/HOL [8] ou ceux de T. Kleymann qui a justifié certaines règles de la logique de Floyd-Hoare et de VDM dans le système LEGO [52, 31]. Ces formalisations permettent d'établir des propriétés méta-théoriques de systèmes formels concernés (correction, complétude relative, justification d'étapes de raffinement, etc.) mais elles se révèlent inutilisables en pratique car basées sur une représentation concrète du système formel dans la logique par des codages assez lourds. C'est ce que l'on pourrait appeler l'internalisation (*deep embedding* en anglais).

Une autre approche, plus souple, consiste à ne pas internaliser le système formel dans la logique, mais à le voir comme du « sucre syntaxique ». Ainsi une formule de correction ou une machine abstraite n'est plus codée dans le formalisme logique, mais directement représentée par une proposition ou un type dépendant. C'est par opposition à l'internalisation un codage *superficiel* (*shallow embedding* en anglais). C'est l'approche adoptée par M. Gordon dans son cours *Hardware and Software Verification* à l'université de Cambridge [18], dans lequel il expose les idées de la logique de Hoare et les met en pratique dans HOL. D'une certaine manière, c'est également l'approche de C. Muñoz qui représente les machines B dans PVS directement par des types dépendants [43]. Mais ces deux derniers exemples restent des formalisations académiques que l'on ne peut pas considérer réellement comme des méthodes de preuve de programme.

C'est cependant dans cette dernière approche que nous nous reconnaissons, car elle consiste à utiliser une logique formelle existante pour spécifier et prouver, au lieu de coder tout un système formel dans un autre. La logique formelle que nous avons choisie est le Calcul des Constructions Inductives [9, 48, 50], et son implantation le système Coq [1]. Notre contribution consiste donc en l'intégration d'une méthode de preuve de programmes dans ce cadre précis. C'est cette méthode que nous allons présenter maintenant.

2.2 Spécifier

Dans cette section nous allons introduire une notion de *spécification* pour nos programmes, comme une extension de la notion de type avec effets, et une notion d'*annotations* de programme.

2.2.1 Types et spécifications

Lorsque l'on spécifie une procédure ou une fonction, ou plus généralement une expression de programme quelconque, on peut se contenter de donner une *pré-condition* et une *post-condition*. Ce sont des assertions logiques portant sur les variables du programmes — qu'il s'agisse de références ou non — et éventuellement sur le résultat du programme. Nous adopterons la notation traditionnelle de la logique de Floyd-Hoare, à savoir

$$\{P\}\ e\ \{Q\}$$

où P est la pré-condition et Q la post-condition. Précisément, P est une proposition portant sur les valeurs des variables de e *avant* son évaluation, dénotées par leur nom, et Q une proposition portant sur les valeurs des variables de e *à l'issue* de l'évaluation, toujours dénotées par leur nom, ainsi que leurs valeurs *avant* l'évaluation, que l'on dénotera par $\overleftarrow{x}$ pour une référence x[1]. De plus, la post-condition Q peut porter sur le résultat de e, et l'on adoptera le cas échéant la notation

$$\{P\}\ e\ \{r \mid Q\}$$

où la variable r lie le résultat de l'évaluation de e dans Q. Toutefois, la notation $\{P\}\ e\ \{Q\}$ sera utilisée lorsqu'il n'est pas fait mention du résultat dans la post-condition.

Mais on doit aller plus loin et dire que l'on *doit* se contenter d'une pré- et d'une post-condition pour spécifier un programme, en plus des informations de type et d'effets déjà introduites. En aucun cas une information contenue *dans* le programme (assertion, invariant de boucle, etc.) ne doit apparaître dans sa spécification *publique*. Car de même que le type d'un objet permet de typer toute expression y faisant référence, la spécification d'un objet doit être une information suffisante pour *prouver* la correction de toute expression impliquant cet objet. Nous pouvons faire l'analogie suivante :

partie publique	nécessaire et suffisante pour
type	le typage
spécification	la preuve

Dès lors, nous pouvons faire mieux qu'une analogie et introduire une notion de spécification qui soit l'extension d'un système de type par des pré- et post-conditions. Et nous pouvons l'introduire *avant même* de parler de programme annotés, de même que l'on peut introduire un système de types avant même de parler de programmes. La définition suivante introduit la notion de *type annoté*, qui sera pour nous la notion de spécification.

Définition 28 (types annotés) *Les notions de types de valeurs et de calculs sont étendues de la manière suivante :*

$$\begin{cases} \textit{valeurs} & \tau & ::= & (\tau, \dots, \tau)c \mid \alpha \mid (x : \tau) \to \kappa \mid \forall\alpha.\tau \mid \tau\ \mathsf{ref} \\ \textit{calculs} & \kappa & ::= & (r : \tau, \epsilon, P, Q) \end{cases}$$

où P désigne une pré-condition et Q une post-condition.

Pour pouvoir définir la notion de types annotés bien formés, il faut être en mesure de vérifier que les assertions contenues dans un type sont elles-mêmes bien formées, c'est-à-dire correctement typées et portant sur le bon ensemble de variables. Pour exprimer cette deuxième condition, nous introduisons les définitions suivantes :

Définition 29 (variables de pre- et post-conditions) *Étant donnés un environnement Γ, une pré-condition P et une post-condition Q, les ensembles de variables $PreV(\Gamma, P)$,*

[1]Dans le langage de spécification de VDM la valeur de sortie est notée $\overleftarrow{x}$, de façon similaire, mais dans le langage de spécification Z, en revanche, la valeur de sortie est notée x et la valeur *initiale* x'. Nous préférons, à l'instar de VDM, une notation où x désigne dans tous les cas la valeur *courante* de la variable x.

$PostV(\Gamma, Q)$ et $Post\overleftarrow{V}(\Gamma, Q)$ sont définis de la manière suivante :

$$\begin{array}{lcl} PreV(\Gamma, P) & \stackrel{def}{=} & \{\ x \mid x \in FV(P) \cap Refs(\Gamma)\ \} \\ PostV(\Gamma, Q) & \stackrel{def}{=} & \{\ x \mid x \in FV(Q) \cap Refs(\Gamma)\ \} \\ Post\overleftarrow{V}(\Gamma, Q) & \stackrel{def}{=} & \{\ x \mid \overleftarrow{x} \in FV(Q) \wedge x \in Refs(\Gamma)\ \} \end{array}$$

où $FV(R)$ désigne l'ensemble des variables libres d'une proposition R arbitraire.

Nous allons maintenant introduire l'interprétation des types τ et κ dans le Calcul des Constructions, afin de pouvoir construire des environnements dans lesquels on puisse typer les assertions. La définition suivante est une extension de la Définition 25 page 48.

Définition 30 (interprétation des types) *Les interprétations des types de valeurs et de calculs dans le Calcul des Constructions, respectivement notées $\widehat{\tau}$ et $\widehat{\kappa}$, sont définies mutuellement récursivement, dans un environnement Γ, de la manière suivante :*

$$\begin{array}{llcll} \textit{valeurs} & \widehat{(\tau_1, \ldots, \tau_n)c} & = & (\widehat{\tau_1}, \ldots, \widehat{\tau_n})c & \\ & \widehat{\alpha} & = & \alpha & \\ & \widehat{\forall\alpha.\tau} & = & \forall\alpha.\widehat{\tau} & \\ & \widehat{(x : \tau\ \mathsf{ref}) \to \kappa} & = & \forall x.\widehat{\kappa} & (x \textit{ est une étiquette}) \\ & \widehat{(x : \tau) \to \kappa} & = & \forall x : \widehat{\tau}.\ \widehat{\kappa} & \textit{sinon} \\ & \widehat{\tau\ \mathsf{ref}} & = & \widehat{\tau} & \\ & & & & \\ \textit{calculs} & (r : \tau, \widehat{(\rho, \omega)}, P, Q) & = & \forall x : \widehat{\rho}.\ P(x) \Rightarrow \exists (y, r) : \widehat{\omega} \times \widehat{\tau}.\ Q(x, y, r) & \\ & & & & \\ \textit{effets} & \widehat{\{x_1, \ldots, x_n\}} & = & \{x_1 : \widehat{\tau_1}; \ldots; x_n : \widehat{\tau_n}\} & \textit{où } x_i : \tau_i\ \mathsf{ref} \in \Gamma \end{array}$$

où les propositions $P(x)$ et $Q(x, y, r)$ sont formellement définies par :

$$P(x) \stackrel{def}{=} P[v \leftarrow x.v]$$

pour $v \in PreV(\Gamma, P)$, et

$$Q(x, y, r) \stackrel{def}{=} Q[\overleftarrow{v} \leftarrow x.v][w \leftarrow x.w][z \leftarrow y.z]$$

pour $v \in Post\overleftarrow{V}(\Gamma, Q)$, $w \in PostV(\Gamma, Q) \backslash \omega$ et $z \in PostV(\Gamma, Q) \cap \omega$.

Si Γ est un environnement associant les variables x_i aux types τ_i, alors $\widehat{\Gamma}$ dénote l'environnement associant les variables x_i aux types $\widehat{\tau_i}$ pour les variables x_i n'étant pas des références.

On notera que les définitions de $P(x)$ et de $Q(x, y, r)$ dépendent toutes deux de l'environnement Γ, et que celle de $Q(x, y, r)$ dépend également de l'effet, par l'intermédiaire de ω. En effet, une référence v apparaissant dans Q est substituée soit par $y.v$ si elle est potentiellement modifiée par le programme (i.e. appartient à ω), soit directement par $x.v$ si elle ne l'est pas, car elle a alors conservé sa valeur initiale. On a ainsi la liberté de mentionner dans les post-conditions la valeur courante d'une référence, que celle-ci soit modifiée ou non par le programme.

Nous pouvons maintenant définir les environnements dans lesquels les pré- et post-conditions seront typées. Ils seront respectivement notés $Pre(\Gamma)$ et $Post(\Gamma, r : \tau)$ où r désigne le résultat de l'évaluation du programme, de type τ.

Définition 31 (environnements de pré- et post-condition) *Soit un environnement* Γ*, dont les références sont* $x_1 : \tau_1$ ref$, \dots, x_n : \tau_n$ ref*. Alors on définit les environnements de pré- et post-conditions de la manière suivante :*

$$Pre(\Gamma) \stackrel{def}{=} \widehat{\Gamma}, x_1 : \widehat{\tau_1}, \dots, x_n : \widehat{\tau_n}$$

et

$$Post(\Gamma, r : \tau) \stackrel{def}{=} Pre(\Gamma), \overleftarrow{x_1} : \widehat{\tau_1}, \dots, \overleftarrow{x_n} : \widehat{\tau_n}, r : \widehat{\tau}$$

Les environnements de pré- et post-conditions étant définis, nous sommes en mesure de définir la notion de type annoté bien formé. Elle étend la Définition 11 page 25.

Définition 32 (types annotés bien formés) *Les jugements* $\Gamma \vdash_a \tau$ bf *et* $\Gamma \vdash_a \kappa$ bf *sont définis inductivement sur la structure de* τ *et* κ *par l'ensemble des règles d'inférence suivantes :*

$$\frac{\forall i \in 1 \dots n \quad \Gamma \vdash_a \tau_i \ \mathsf{bf}}{\Gamma \vdash_a (\tau_1, \dots, \tau_n)c \ \mathsf{bf}} \qquad \frac{\Gamma \vdash_a \tau_1 \ \mathsf{bf} \qquad \Gamma, x : \tau_1 \vdash_a \kappa_2 \ \mathsf{bf}}{\Gamma \vdash_a (x : \tau_1) \to \kappa_2 \ \mathsf{bf}}$$

$$\frac{\alpha \in \Gamma}{\Gamma \vdash_a \alpha \ \mathsf{bf}} \qquad \frac{\Gamma, \alpha \vdash_a \tau \ \mathsf{bf}}{\Gamma \vdash_a \forall \alpha . \tau \ \mathsf{bf}} \qquad \frac{\Gamma \vdash_a \tau \ \mathsf{bf}}{\Gamma \vdash_a \tau \ \mathsf{ref} \ \mathsf{bf}}$$

$$\frac{\begin{array}{c}\Gamma \vdash_a \tau \ \mathsf{bf} \qquad \omega \subseteq \rho \subseteq Refs(\Gamma) \\ PreV(\Gamma, P) \cup PostV(\Gamma, Q) \cup Post\overleftarrow{V}(\Gamma, Q) \subseteq \rho \\ Pre(\Gamma) \vdash_{\mathrm{CIC}} P : \mathsf{Prop} \qquad Post(\Gamma, r : \tau) \vdash_{\mathrm{CIC}} Q : \mathsf{Prop}\end{array}}{\Gamma \vdash_a (r : \tau, (\rho, \omega), P, Q) \ \mathsf{bf}}$$

La définition de Γ bf *reste la même.*

2.2.2 Programmes annotés

Maintenant que nous avons introduit la notion de spécification, que nous confondons avec celle de type annoté, venons-en aux programmes eux-mêmes, et à la manière dont ils seront annotés. Comme nous l'avons expliqué dans la section précédente, un programme peut se voir annoté d'une pré- et d'une post-condition, et celles-ci apparaissent dans son type de calcul κ. Dès lors, il est naturel qu'il en soit de même pour chaque expression du programme. Ainsi, il n'y a pas de différence, au niveau de la spécification, entre une sous-expression et le programme lui-même, et chaque expression pourra être annotée avec la syntaxe déjà introduite

$$\{P\} \ e \ \{r \mid Q\}$$

Ainsi, par exemple, les annotations pertinentes d'une fonction fun $(x : \tau) \to e$ ne sont pas les pré- et post-conditions portant sur la fonction elle-même mais celles qui portent sur son corps e. Elles seront donc introduites comme telles, c'est-à-dire avec la syntaxe

$$\mathsf{fun} \ (x : \tau) \to \{P_f\} \ e \ \{r \mid Q_f\}$$

Invariants de boucle. Parmi les annotations de programmes, on donne généralement une importance particulière aux *invariants de boucle*. En effet, par le caractère même de la boucle, qui ne renvoie pas de résultat mais modifie des données en place, il se dégage le plus souvent une propriété qui persiste à chaque itération de la boucle, que l'on nomme *invariant de boucle*. En fait, c'est même l'idée de cet invariant qui guide le programmeur dans l'écriture d'une boucle, même si parfois cette idée reste informelle. En toute généralité, on pourrait utiliser les annotations déjà introduites pour annoter une boucle de la façon suivante

$$\mathsf{while}\ e_1\ \mathsf{do}\ \{P\}\ e_2\ \{Q\}\ \mathsf{done}$$

mais comme le plus souvent P et Q sont de la forme $I \wedge T_1$ et I, où T_1 est une proposition exprimant que le test e_1 est satisfait, on introduira une notation particulière de la forme

$$\mathsf{while}\ e_1\ \mathsf{do}\ \{\mathsf{invariant}\ I\}\ e_2\ \mathsf{done}$$

On remarquera qu'à la différence d'une boucle, une fonction récursive a généralement des pré- et post-conditions distinctes et son annotation sera simplement de la forme

$$\mathsf{rec}\ f\ (x : \tau) : \kappa = \{P_f\}\ e\ \{r \mid Q_f\}$$

à la manière d'une fonction non récursive.

Terminaison. Puisque nous nous intéressons exclusivement à la correction totale, nous devons être en mesure de justifier la terminaison des programmes, c'est-à-dire des boucles et des fonctions récursives. Une telle justification passe généralement par la donnée d'une quantité qui décroit strictement pour un relation bien fondée ; cette quantité est appelée *variant*. Plutôt que de se limiter à une quantité entière, voire un ordinal, comme c'est généralement le cas, nous laisserons la possibilité à l'utilisateur de spécifier un variant quelconque, sous la forme d'un couple $\nu = (\phi, R)$ où ϕ est le variant proprement dit et R la relation, que l'on devra montrer être bien fondée.

Définition 33 (variant) *Étant donné un environnement* Γ*, un* variant *dans* Γ *est un couple* $\nu = (\phi, R)$ *bien typé dans le Calcul des Constructions dans l'environnement* $Pre(\Gamma)$ *et ayant un type de la forme :*

$$\mathit{Variant}(A) \quad \stackrel{def}{=} \quad A \times (A \rightarrow A \rightarrow \mathsf{Prop})$$

où A *est un type quelconque.*

Les boucles et fonctions récursives se verront donc explicitement annotées par un variant. Dans le cas d'une fonction récursive, le variant dépend le plus souvent des arguments de la fonction — dans le cas contraire on aurait écrit une boucle — et nous allons donc modifier la syntaxe des fonctions récursives afin d'accepter plusieurs arguments, de manière à pouvoir insérer le variant après ceux-ci. On abandonne donc la vision de la construction rec comme opérateur de point fixe pour la vision plus traditionnelle de la construction let rec de ML.

Nous pouvons maintenant donner la syntaxe des programmes annotés.

$$
\begin{array}{rcl}
e & ::= & \{P\}\ s\ \{r \mid Q\} \\
s & ::= & c \\
& | & x \\
& | & (e\ e) \\
& | & (e\ \tau) \\
& | & \mathsf{fun}\ (x : \tau) \rightarrow e \\
& | & \mathsf{Fun}\ \alpha \rightarrow e \\
& | & \mathsf{rec}\ f\ (\vec{x} : \vec{\tau}) : \kappa\ \{\mathsf{variant}\ \nu\} = e \\
& | & \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
& | & \mathsf{let}\ x = e\ \mathsf{in}\ e \\
& | & e\ ;\ e \\
& | & \mathsf{while}\ e\ \mathsf{do}\ \{\mathsf{invariant}\ P\ \mathsf{variant}\ \nu\}\ e\ \mathsf{done} \\
& | & \mathsf{ref}\ e \\
& | & !x \\
& | & x := e
\end{array}
$$

FIG. 2.1: Programmes annotés

Définition 34 (programmes annotés) *La syntaxe des programmes annotés est donnée Figure 2.1, où P désigne une pré-condition, Q une post-condition et ν un variant.*

On remarque que la syntaxe des programmes annotés impose une pré- et une post-condition à *chaque* sous-expression. En pratique, bien entendu, on ne souhaite pas annoter toutes les sous-expressions, mais au contraire ne donner qu'un nombre minimum d'annotations. Nous aurions pu introduire des pré- et post-conditions optionnelles dans la syntaxe ci-dessus, mais cela aurait compliqué par la suite toutes les définitions s'appuyant sur la syntaxe des programmes. Nous préférons donc supposer pour l'instant une annotation complète des programmes, et nous montrerons par la suite, dans la section 2.4 dédiée à la complétude, comment un grand nombre d'annotations peuvent être insérées de manière automatique — sans affecter la prouvabilité des obligations de preuve.

Si nous avons introduit les types annotés τ et κ (Définition 28 page 63), c'est pour typer les programmes annotés, à la manière de ce que nous avons déjà fait pour les effets dans la section 1.2. On introduit donc le jugement « dans l'environnement Γ, le programme annoté e a le type annoté κ », que l'on notera $\Gamma \vdash_a e : \kappa$.

Définition 35 (typage des programmes annotés) *Le jugement $\Gamma \vdash_a e : \kappa$ est défini par la règle d'inférence suivante :*

$$
\frac{\textit{soit } \epsilon' \stackrel{\mathit{def}}{=} (\mathit{PreV}(\Gamma, P) \cup \mathit{PostV}(\Gamma, Q) \cup \mathit{Post}\overleftarrow{V}(\Gamma, Q), \emptyset) \qquad \mathit{Pre}(\Gamma) \vdash_{\mathrm{CIC}} P : \mathsf{Prop} \qquad \Gamma \vdash_a s : (\tau, \epsilon) \qquad \mathit{Post}(\Gamma, r : \tau) \vdash_{\mathrm{CIC}} Q : \mathsf{Prop}}{\Gamma \vdash_a \{P\}\ s\ \{r \mid Q\} : (r : \tau, \epsilon \sqcup \epsilon', P, Q)}
$$

où le jugement $\Gamma \vdash_a s : (\tau, \epsilon)$, qui étend le jugement $\vdash_r$ introduit section 1.2.1 aux programmes annotés, est défini par les règles d'inférences données Figure 2.2. Le jugement

$\Gamma \vdash_a e : \kappa$ *y apparaît dans une forme « affaiblie » obtenue par la règle suivante :*

$$\frac{\Gamma \vdash_a e : (r : \tau, \epsilon, P, Q)}{\Gamma \vdash_a e : (\tau, \epsilon)}$$

Exemples. Reprenons une fois encore les exemples de la page 31 et donnons pour chacun une ou plusieurs spécifications possibles. Lorsqu'une expression de programme ne comporte qu'une des deux annotations (seule la pré-condition ou seule la post-condition), on fera tout de même apparaître la paire d'accolades correspondant à l'annotation vide, de manière à ce que la portée de l'annotation existante soit sans ambiguïté. Ainsi dans le programme annoté $\{\}\ x := 0\ \{Q\}$, il est clair que Q porte sur l'affectation et non sur la constante 0.

Exemple 7 *Le premier programme $ex_1 \equiv x := !x + 1$ peut donner lieu à la spécification suivante :*

$$ex'_1 \quad \equiv \quad \{\}\ x := !x + 1\ \{x > \overleftarrow{x}\}$$

ou encore

$$ex'_1 \quad \equiv \quad \{x > 0\}\ x := !x + 1\ \{x > 1\}$$

Une spécification « complète », exprimant exactement ce que réalise le programme ex_1, est évidemment la suivante :

$$ex'_1 \quad \equiv \quad \{\}\ x := !x + 1\ \{x = \overleftarrow{x} + 1\}$$

Dans ce dernier cas, on a le jugement suivant

$$\Gamma \vdash_a ex'_1 : (v : \mathsf{unit}, (\{x\}, \{x\}), \mathsf{True}, x = \overleftarrow{x} + 1)$$

ce qui permet de déduire le type de l'interprétation de ex'_1 dans le Calcul des Constructions :

$$\widehat{ex'_1} : \forall u : \{x : \mathsf{int}\}.\ \{(y, v) : \{x : \mathsf{int}\} \times \mathsf{unit} \mid y.x = u.x + 1\}$$

□

Exemple 8 *Une façon d'annoter le second programme ex_2 est la suivante :*

$$ex'_2 \quad \equiv \quad \mathsf{fun}\ (r : \mathsf{int\ ref}) \to \{\}\ r := !r + 1\ ;\ !r\ \{v \mid r = \overleftarrow{r} + 1 \wedge v = r\}$$

et on a

$$\Gamma \vdash_a ex'_2 : ((r : \mathsf{int\ ref}) \to (v : \mathsf{int}, (\{r\}, \{r\}), \mathsf{True}, r = \overleftarrow{r} + 1 \wedge v = r), \bot, \mathsf{True}, \mathsf{True})$$

□

Exemple 9 *Une manière d'annoter le dernier programme ex_3 pourrait être*

$$ex'_3 \quad \equiv \quad \{\}\ y := (ex_2\ x)\ \{y = \overleftarrow{x} + 1\}$$

□

Des exemples plus conséquents seront donnés section 2.3.4, page 85.

$$\frac{\mathit{Type}(c) = \tau}{\Gamma \vdash_a c : (\tau, \bot)}(\mathsf{CONST}_a) \qquad \frac{x : \tau \in \Gamma \qquad \tau \neq \_\ \mathsf{ref}}{\Gamma \vdash_a x : (\tau, \bot)}(\mathsf{VAR}_a)$$

$$\frac{\Gamma, x : \tau \vdash_a e : \kappa \qquad \Gamma \vdash_a \tau\ \mathsf{bf}}{\Gamma \vdash_a \mathsf{fun}\ (x : \tau) \to e : ((x : \tau) \to \kappa, \bot)}(\mathsf{FUN}_a) \qquad \frac{\Gamma, \alpha \vdash_a e : (\tau, \epsilon) \qquad \alpha \notin \Gamma}{\Gamma \vdash_a \mathsf{Fun}\ \alpha \to e : (\forall \alpha.\tau, \epsilon)}(\mathsf{FUNT}_a)$$

$$\frac{\Gamma \vdash_a e : (\tau_1 \to (\tau, \epsilon), \epsilon_2) \qquad \Gamma \vdash_a e_1 : (\tau_1, \epsilon_1) \qquad \tau_1 \neq \_\ \mathsf{ref}}{\Gamma \vdash_a (e\ e_1) : (\tau, \epsilon_2 \sqcup \epsilon_1 \sqcup \epsilon)}(\mathsf{APP}_a)$$

$$\frac{\Gamma \vdash_a e : ((x : \tau_1\ \mathsf{ref}) \to (\tau, \epsilon), \epsilon_2) \qquad r : \tau_1\ \mathsf{ref} \in \Gamma \qquad r \notin (\tau, \epsilon)}{\Gamma \vdash_a (e\ r) : (\tau[x \leftarrow r], \epsilon_2 \sqcup \epsilon[x \leftarrow r])}(\mathsf{APPREF}_a)$$

$$\frac{\Gamma \vdash_a e : (\forall \alpha.\tau, \epsilon) \qquad \Gamma \vdash_a \tau_1\ \mathsf{bf}}{\Gamma \vdash_a (e\ \tau_1) : (\tau[\alpha \leftarrow \tau_1], \epsilon)}(\mathsf{APPT}_a)$$

$$\frac{\begin{array}{c}\Gamma \vdash_a \vec{\tau}\ \mathsf{bf} \qquad \Gamma, \vec{x} : \vec{\tau} \vdash_a \kappa\ \mathsf{bf} \\ \Gamma, f : (\vec{x} : \vec{\tau}) \to \kappa, \vec{x} : \vec{\tau} \vdash_a e : \kappa \qquad \mathit{Pre}(\Gamma, \vec{x} : \vec{\tau}) \vdash_{\mathrm{CIC}} \nu : \mathit{Variant}(A)\end{array}}{\Gamma \vdash_a \mathsf{rec}\ f\ (\vec{x} : \vec{\tau}) : \kappa\ \{\mathsf{variant}\ \nu\} = e : ((\vec{x} : \vec{\tau}) \to \kappa, \bot)}(\mathsf{REC}_a)$$

$$\frac{\Gamma \vdash_a e_1 : (\mathsf{bool}, \epsilon_1) \qquad \Gamma \vdash_a e_2 : (\tau, \epsilon_2) \qquad \Gamma \vdash_a e_3 : (\tau, \epsilon_3)}{\Gamma \vdash_a \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : (\tau, \epsilon_1 \sqcup \epsilon_2 \sqcup \epsilon_3)}(\mathsf{COND}_a)$$

$$\frac{\Gamma \vdash_a e_1 : (\tau_1, \epsilon_1) \qquad \tau_1 \neq \_\ \mathsf{ref} \qquad \Gamma, x : \tau_1 \vdash_a e_2 : (\tau, \epsilon)}{\Gamma \vdash_a \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : (\tau, \epsilon_1 \sqcup \epsilon)}(\mathsf{LET}_a)$$

$$\frac{\Gamma \vdash_a e_1 : (\tau_1\ \mathsf{ref}, \epsilon_1) \qquad \Gamma, x : \tau_1\ \mathsf{ref} \vdash_a e_2 : (\tau, \epsilon) \qquad x \notin \tau}{\Gamma \vdash_a \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : (\tau, \epsilon_1 \sqcup \epsilon \backslash x)}(\mathsf{LETREF}_a)$$

$$\frac{\Gamma \vdash_a e_1 : (\mathsf{unit}, \epsilon_1) \qquad \Gamma \vdash_a e_2 : (\tau, \epsilon_2)}{\Gamma \vdash_a e_1\ ;\ e_2 : (\tau, \epsilon_1 \sqcup \epsilon_2)}(\mathsf{SEQ}_a)$$

$$\frac{\begin{array}{c}\Gamma \vdash_a e_1 : (\mathsf{bool}, \epsilon_1) \qquad \Gamma \vdash_a e_2 : (\mathsf{unit}, \epsilon_2) \\ \mathit{Pre}(\Gamma) \vdash_{\mathrm{CIC}} P : \mathsf{Prop} \qquad \mathit{Pre}(\Gamma) \vdash_{\mathrm{CIC}} \nu : \mathit{Variant}(A)\end{array}}{\Gamma \vdash_a \mathsf{while}\ e_1\ \mathsf{do}\ \{\mathsf{invariant}\ P\ \mathsf{variant}\ \nu\}\ e_2\ \mathsf{done} : (\mathsf{unit}, \epsilon_1 \sqcup \epsilon_2)}(\mathsf{LOOP}_a)$$

$$\frac{\Gamma \vdash_a e : (\tau, \epsilon) \qquad \tau\ \mathsf{pur}}{\Gamma \vdash_a \mathsf{ref}\ e : (\tau\ \mathsf{ref}, \epsilon)}(\mathsf{REF}_a) \qquad \frac{x : \tau\ \mathsf{ref} \in \Gamma}{\Gamma \vdash_a !x : (\tau, (\{x\}, \emptyset))}(\mathsf{DEREF}_a)$$

$$\frac{x : \tau\ \mathsf{ref} \in \Gamma \qquad \Gamma \vdash_a e : (\tau, (\rho, \omega))}{\Gamma \vdash_a x := e : (\mathsf{unit}, (\{x\} \cup \rho, \{x\} \cup \omega))}(\mathsf{AFF}_a)$$

FIG. 2.2: Typage des programmes annotés

2.2.3 Variables auxiliaires

À ce point, le lecteur peut être surpris que nous n'ayons pas parlé des *variables auxiliaires*. Encore appelées variables *logiques*, elles sont introduites dans les spécifications pour relier entre elles les valeurs successives des références. En effet, si l'on souhaite par exemple spécifier un programme calculant dans la référence y la factorielle de la valeur contenue dans la référence x on pourrait penser à la spécification suivante :

$$\{x \geq 0\} \ \ldots \ \{y = x!\}$$

mais celle-ci est trivialement réalisée par le programme suivant :

$$\{x \geq 0\} \ x := 1 \ ; \ y := 1 \ \{y = x!\}$$

Il faut exprimer dans la spécification que l'on calcule dans y la factorielle de la valeur *initiale* de x. Si l'on suppose que l'on ne dispose pas de prédicats avant-après, une telle spécification devra faire intervenir une variable auxiliaire, par exemple de la manière suivante :

$$\{x = X \geq 0\} \ \ldots \ \{y = X!\}$$

On peut objecter sur cet exemple que la notion de post-condition que nous avons introduite permet de spécifier correctement un tel programme, en lui donnant la post-condition $y = \overleftarrow{x}!$. Mais dans certains cas, même l'utilisation de prédicats avant-après dans les post-conditions ne permet pas de spécifier le programme correctement — voir par exemple la procédure F page 114 — et l'utilisation de variables auxiliaires devient nécessaire.

Il est possible de formaliser la notion de variable auxiliaire, ainsi que l'a montré T. Schreiber dans sa thèse [31, 52][2]. Ces variables apparaissent alors comme quantifiées universellement dans la formule de correction, ce qui correspond bien à l'idée intuitive que l'on en a. Cependant, nous proposons un autre moyen que les variables auxiliaires pour dénoter les valeurs successives des références dans les annotations. Très simplement, il s'agit d'un moyen *syntaxique* permettant de désigner ces différentes valeurs, à l'aide d'étiquettes placées dans le programme pour en désigner les points pertinents pour la spécification. Essentiellement technique, ce point sera détaillé dans le paragraphe *De l'inutilité des variables auxiliaires*, page 114.

2.3 Prouver

Nous en arrivons au point où nous allons pouvoir introduire la méthodologie de preuve de programmes proprement dite. Nous venons de définir ce qui sera notre notion de spécification, sous la forme d'un jugement de typage pour une notion étendue de types de valeurs et de calculs. L'étape suivante consiste maintenant à engendrer à partir d'une spécification et d'un programme annoté un ensemble d'*obligations de preuve*, dont la validité établit la correction du programme vis-à-vis de la spécification donnée.

[2] T. Schreiber et T. Kleymann ne sont en fait qu'une seule et même personne !

Notre méthodologie s'appuie sur les notions d'*extraction* et de *réalisabilité* dans le Calcul des Constructions Inductives, et nous introduirons ces notions avant de donner l'idée intuitive de notre méthode de preuve de programmes. Ce sera l'objet de la première section. La deuxième section définit formellement la méthodologie, sous la forme d'une interprétation des programmes annotés dans le Calcul des Constructions Inductives. La troisième section établit ensuite la correction de la méthode, qui repose sur le fait que l'interprétation des programmes a été construite au dessus de leur traduction fonctionnelle. Enfin, la dernière section étudie les comparaisons avec la logique de Floyd-Hoare, en s'appuyant sur le détail des obligations de preuves dans de nombreux cas de figures.

2.3.1 Méthodologie

Il y a dans le Calcul des Constructions Inductives une distinction entre propositions *informatives* et propositions *logiques*, introduite de manière syntaxique par les sortes respectives Set et Prop. Pour simplifier, on peut considérer que la première contient les types de données et la seconde les prédicats. Cette distinction est justifiée par le mécanisme d'*extraction*. Introduite dans le Calcul des Constructions par C. Paulin [48, 47], l'extraction exploite le caractère constructif des preuves pour en extraire le contenu calculatoire sous la forme d'une fonction. Ainsi, d'une preuve de la proposition

$$\forall x.\ P(x) \Rightarrow \exists y.\ Q(x,y)$$

où les prédicats P et Q sont logiques et x et y des objets informatifs, on peut extraire une fonction f calculant y en fonction de x. Le calcul de f est justifié par le théorème de *réalisabilité*, qui exprime que f réalise bien la spécification donnée par le type de la preuve dont elle est extraite. En l'occurrence, la réalisabilité s'exprime ici par la proposition

$$\forall x.\ P(x) \Rightarrow Q(x, f(x))$$

Outre dans les deux références données ci-dessus, on pourra trouver une présentation de l'extraction et de la réalisabilité dans l'ouvrage *Logique et informatique : une introduction* [49].

L'idée de base de notre méthode de preuve de programmes est alors la suivante : à partir d'un programme e et d'une spécification κ, on va construire une preuve $\widehat{e}$ de la proposition $\widehat{\kappa}$, qui aura pour caractéristique d'avoir un contenu extrait égal à la traduction fonctionnelle $\overline{e}$. De façon informelle, la justification de la méthode est alors la suivante : la spécification κ étant de la forme (τ, ϵ, P, Q) on a par définition

$$\widehat{\kappa} \quad = \quad \forall x.\ P(x) \Rightarrow \exists y.\ Q(x,y)$$

et si $\widehat{e}$ est une preuve de $\widehat{\kappa}$ dont le contenu extrait est $\overline{e}$ alors la réalisabilité exprimera que

$$\forall x.\ P(x) \Rightarrow Q(x, (\overline{e}\ x))$$

c'est-à-dire que pour toute entrée x vérifiant la pré-condition P alors le résultat du programme, à savoir $(\overline{e}\ x)$, satisfait bien la post-condition Q. C'est exactement ce que l'on entend par la correction du programme e vis-à-vis de la spécification κ.

Bien entendu, il n'est pas possible de construire le terme de preuve $\widehat{e}$ intégralement — il s'agit clairement d'un problème indécidable. Ce terme sera donc *incomplet*, ses « trous » étant autant d'obligations de preuve laissées à la charge de l'utilisateur. Seule la partie informative de $\widehat{e}$ peut être construire, puisqu'elle est égale $\overline{e}$. Il ne subsistera donc pour l'utilisateur que des propositions *logiques*, portant sur les différentes variables introduites par la traduction fonctionnelle. En pratique, notre tactique déclarera la proposition $\widehat{\kappa}$ comme but à prouver et on en donnera pour preuve le terme $\widehat{e}$ par l'intermédiaire d'une tactique particulière, appelée Refine, qui engendre un sous-but pour chaque morceau manquant dans le terme de preuve.

Notre méthodologie est donc proche de celle de C. Parent, qui a défini dans sa thèse une méthode de preuve de programmes fonctionnels dans le Calcul des Constructions Inductives basée sur la construction partielle d'un terme de preuve à partir de la donnée d'un programme fonctionnel et d'une spécification [45, 46]. Cette méthodologie est implantée dans le système Coq, sous la forme d'une tactique appelée Program. Bien que C. Parent ait implanté cette méthode d'une façon directe, en engendrant directement les obligations de preuve, on peut tout à fait décomposer sa méthode en deux étapes, dont la première serait la construction d'un terme de preuve incomplet, et la deuxième la simple application de la tactique Refine, de manière identique à ce que nous allons faire. Dès lors, notre approche peut être vue comme une extension de la méthodologie de C. Parent aux programmes impératifs, dont la clé réside dans la traduction fonctionnelle définie au premier chapitre. Cependant, il faut relativiser cette comparaison, dans la mesure où nous ne considérons dans nos programmes que des objets purement informatifs, alors que la tactique Program tire pleinement profit de l'extraction en permettant de manipuler dans les programmes des objets contenant éventuellement des parties logiques. Nous évoquerons une telle extension de notre travail dans les perspectives, page 152.

Terminaison. Nous avons expliqué dans la section précédente que nous nous intéressons exclusivement à la correction *totale* des programmes. La terminaison d'un programme dont on fera la preuve sera garantie par le fait que l'on ne peut définir dans le Calcul des Constructions Inductives que des fonctions totales, et donc qu'en particulier les termes $\widehat{e}$ et $\overline{e}$ seront totaux. Bien entendu, le prix à payer pour l'utilisateur est la donnée explicite d'une justification de cette terminaison, sous la forme d'un variant, pour permettre la construction de $\widehat{e}$ par récurrence bien fondée, et la preuve d'un certain nombre d'obligations concernant le variant et la relation qui l'accompagne.

Avant de pouvoir définir le terme de preuve $\widehat{e}$, il nous faut introduire formellement les notions d'extraction et de réalisabilité, telles qu'elles sont définies par C. Paulin dans [48]. On peut ici se limiter à la classe des propositions qui nous intéressent. Celles-ci sont formées de propositions *logiques*, notées L, et de propositions *informatives*, notées I, construites de la manière suivante :

$$\begin{array}{rcl} I & ::= & A, \text{ proposition atomique informative} \\ & | & \{x : I; \ldots; x : I\} \\ & | & \forall x : I.\ I \\ & | & \exists x : I.\ L \\ & | & L \Rightarrow I \end{array}$$

L'opérateur d'extraction, noté $\mathcal{E}$, est alors défini par récurrence sur les propositions informatives de la manière suivante ([48] page 77) :

$$\begin{array}{lcl} \mathcal{E}(A) & \stackrel{\text{def}}{=} & A \\ \mathcal{E}(\{x_i : I_i\}) & \stackrel{\text{def}}{=} & \{x_i = \mathcal{E}(I_i)\} \\ \mathcal{E}(\forall x : I_1.\ I_2) & \stackrel{\text{def}}{=} & \mathcal{E}(I_1) \rightarrow \mathcal{E}(I_2) \\ \mathcal{E}(\exists x : I.\ L) & \stackrel{\text{def}}{=} & \mathcal{E}(I) \\ \mathcal{E}(L \Rightarrow I) & \stackrel{\text{def}}{=} & \mathcal{E}(I) \end{array}$$

Ainsi, si l'on considère la proposition S suivante exprimant l'existence d'un quotient et d'un reste dans la division euclidienne des entiers :

$$S \quad \equiv \quad \forall a, b : \mathsf{int}.\ b > 0 \Rightarrow \exists c : \mathsf{int} \times \mathsf{int}.\ \mathsf{let}\ (q, r) = c\ \mathsf{in}\ a = b \times q + r \wedge 0 \leq r < b \qquad (2.1)$$

alors les quantités a, b et c consituent les parties informatives et les propositions $b > 0$ et $\mathsf{let}\ (q, r) = c\ \mathsf{in}\ a = b \times q + r \wedge 0 \leq r < b$ les parties logiques. Dès lors, l'extraction de S est :

$$\mathcal{E}(S) \quad = \quad \mathsf{int} \rightarrow \mathsf{int} \rightarrow \mathsf{int} \times \mathsf{int}$$

c'est-à-dire le type d'une fonction prenant en argument deux entiers, à savoir a et b, et retournant une paire d'entiers, à savoir (q, r).

Si t est un terme informatif de type I, alors son terme extrait $\mathcal{E}(t)$ a pour type $\mathcal{E}(I)$ et est défini récursivement sur la structure de t par les équations données figure 2.3 ([48] page 77).

$$\begin{array}{lcl} \mathcal{E}(x) & \stackrel{\text{def}}{=} & x \\ \mathcal{E}(c) & \stackrel{\text{def}}{=} & c \\ \mathcal{E}(\{x_i = t_i\}) & \stackrel{\text{def}}{=} & \{x_i = \mathcal{E}(t_i)\} \\ \mathcal{E}(\lambda x : I.t) & \stackrel{\text{def}}{=} & \lambda x : \mathcal{E}(I).\mathcal{E}(t) \\ \mathcal{E}(\lambda x : L.t) & \stackrel{\text{def}}{=} & \mathcal{E}(t) \\ \mathcal{E}(\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2) & \stackrel{\text{def}}{=} & \left\{ \begin{array}{ll} \mathsf{let}\ x = \mathcal{E}(t_1)\ \mathsf{in}\ \mathcal{E}(t_2) & \text{si } t_1 \text{ est informatif} \\ \mathcal{E}(t_2) & \text{sinon} \end{array} \right. \\ \mathcal{E}((t_1\ t_2)) & \stackrel{\text{def}}{=} & \left\{ \begin{array}{ll} (\mathcal{E}(t_1)\ \mathcal{E}(t_2)) & \text{si } t_2 \text{ est informatif} \\ \mathcal{E}(t_1) & \text{sinon} \end{array} \right. \end{array}$$

FIG. 2.3: Définition du terme extrait

Nous pouvons donner maintenant la définition de la réalisabilité fonctionnelle dans le Calcul des Constructions Inductives, telle qu'elle est introduite par C. Paulin dans [48] page 81, sur le fragment de propositions informatives que nous considérons ici :

Définition 36 (réalisabilité fonctionnelle, C. Paulin [48]) *La proposition $x \ \mathsf{r}_f \ I$ est définie par récurrence sur la formule I de la manière suivante :*

$$\begin{array}{llll} x \ \mathsf{r}_f \ A & \stackrel{def}{=} & \mathsf{True} & \textit{pour } A \textit{ proposition atomique} \\ x \ \mathsf{r}_f \ \forall y : I.I' & \stackrel{def}{=} & \forall y : \mathcal{E}(I).\ y \ \mathsf{r}_f \ I \Rightarrow (x \ y) \ \mathsf{r}_f \ I' & \\ x \ \mathsf{r}_f \ \exists y : I.L & \stackrel{def}{=} & x \ \mathsf{r}_f \ I \wedge \overline{L}[y \leftarrow x] & \\ x \ \mathsf{r}_f \ L \Rightarrow I & \stackrel{def}{=} & \overline{L} \Rightarrow x \ \mathsf{r}_f \ I & \end{array}$$

où $\overline{L}$ désigne l'interprétation de la proposition logique L dans le monde extrait, c'est-à-dire la proposition L appliquée aux termes extraits des objets informatifs et non aux objets informatifs eux-mêmes (voir [48] page 82).

Si nous reprenons l'exemple (2.1) ci-dessus, alors la proposition $f \ \mathsf{r}_f \ S$ est par définition la suivante :

$$\forall a, b : \mathsf{int}.b > 0 \Rightarrow \mathsf{let}\ (q, r) = (f \ a \ b)\ \mathsf{in}\ a = b \times q + r \wedge 0 \leq r < b$$

Elle exprime exactement que la fonction f, appliquée à deux entiers a et b, retourne une paire (q, r) qui satisfait bien la « spécification » S.

Cette définition de la réalisabilité fonctionnelle va de paire avec le théorème de réalisabilité ([48] page 85), qui s'énonce ainsi :

Théorème 3 (de réalisabilité, C. Paulin [48]) *Soit Γ un environnement de preuve dans le* Cic. *Alors*

$$\Gamma \vdash_{\mathrm{CIC}} e : T \quad \Longrightarrow \quad \mathcal{E}(e) \ \mathsf{r}_f \ T$$

2.3.2 Interprétation des programmes annotés

Nous allons maintenant définir l'interprétation d'un programme e dans le Calcul des Constructions, que l'on note $\widehat{e}$. Il s'agit d'un terme de preuve incomplet, où les « morceaux manquants » sont notés par des points d'interrogation. Ces morceaux manquants, qui donneront lieu à des obligations de preuves, sont bien entendu typés, grâce aux informations provenant du typage des programmes annotés défini dans la section précédente. Pour plus de clarté nous indiçons les points d'interrogation et donnons leurs types séparément, pour ne pas alourdir la définition de $\widehat{e}$.

Pour interpréter les boucles et les fonctions récursives, nous allons utiliser un opérateur de récurrence bien fondée, WF, dont le type est le suivant :

$$\begin{array}{ll} \mathsf{WF}: & \forall A : \mathsf{Set}.\ \forall R : A \rightarrow A \rightarrow \mathsf{Prop}.\ (\textit{well\_founded}\ R) \Rightarrow \\ & \quad \forall P : A \rightarrow \mathsf{Set}. \\ & \quad (\forall x : A.(\forall y : A.(R\ y\ x) \Rightarrow P(y)) \Rightarrow P(x)) \Rightarrow \\ & \qquad \forall a : A.\ P(a) \end{array}$$

La proposition $(\textit{well\_founded}\ R)$ exprime que la relation R est bien fondée, et il n'est pas nécessaire de préciser ici la définition de ce prédicat. On remarquera que ce principe n'exige pas que R soit une relation d'ordre, mais uniquement que R soit bien fondée.

Définition 37 (interprétation des programmes) *Soit Γ un environnement bien formé et e un programme tel que $\Gamma \vdash_a e : \kappa$. Alors l'interprétation de e dans le Calcul des Constructions, notée $\widehat{e}$, est un terme incomplet de type $\widehat{\kappa}$ dans le contexte $\widehat{\Gamma}$, défini par récurrence sur la structure de e de la manière suivante :*

$e \equiv \{P\}\ s\ \{Q\}$ **avec** $s = c \mid x$ **:**

$$\widehat{e} = \lambda x_0.\lambda h : P(x_0).(\{\}, s, ?_1)$$

avec $?_1 : Q(x_0, \{\}, s)$.

$e \equiv \{P\}\ (e_2\ e_1)\ \{Q\}$ **:** *On sait que* $\Gamma \vdash_a e_2 : ((x : \tau_1) \to (\tau, \epsilon, P', Q'), \epsilon_2, P_2, Q_2)$. *On distingue alors deux cas suivant le type de* e_1 *:*

- *Si* $\Gamma \vdash_a e_1 : (\tau_1, \epsilon_1, P_1, Q_1)$, *avec* $\tau_1 \neq \_\ \mathsf{ref}$, *alors*

$$\begin{array}{rl}\widehat{e} = \lambda x_0.\lambda h : P(x_0). & \mathsf{let}\ (x_1, a, q_1) = (\widehat{e_1}\ x_0\ ?_1)\ \mathsf{in} \\ & \mathsf{let}\ (x_2, f, q_2) = (\widehat{e_2}\ (x_0 \oplus x_1)\ ?_2)\ \mathsf{in} \\ & \mathsf{let}\ (x_3, v, q) = (f\ a\ (x_0 \oplus x_1 \oplus x_2)\ ?_3)\ \mathsf{in} \\ & (x_1 \oplus x_2 \oplus x_3, v, ?_4)\end{array}$$

  avec $?_1 : P_1(x_0)$, $?_2 : P_2(x_0 \oplus x_1)$, $?_3 : P'(x_0 \oplus x_1 \oplus x_2)$ *et* $?_4 : Q(x_0, x_1 \oplus x_2 \oplus x_3, v)$.

- *Si* $e_1 = r : \tau_1'\ \mathsf{ref}$ *alors*

$$\begin{array}{rl}\widehat{e} = \lambda x_0.\lambda h : P(x_0). & \mathsf{let}\ (x_1, f, q_2) = (\widehat{e_2}\ x_0\ ?_1)\ \mathsf{in} \\ & \mathsf{let}\ (x_2, v, q) = (f\ r\ (x_0 \oplus x_1)\ ?_2)\ \mathsf{in} \\ & (x_1 \oplus x_2, v, ?_3)\end{array}$$

  avec $?_1 : P_2(x_0)$, $?_2 : P'(x_0 \oplus x_1)$, *et* $?_3 : Q(x_0, x_1 \oplus x_2, v)$.

$e \equiv \{P\}\ (e_1\ \tau)\ \{Q\}$ **:**

$$\widehat{e} = \lambda x_0.\lambda h : P(x_0).((\widehat{e_1}\ \widehat{\tau}), ?_1)$$

avec $?_1 : Q(x_0, \{\}, (\widehat{e_1}\ \widehat{\tau}))$.

$e \equiv \{P\}\ \mathsf{fun}\ (x : \tau) \to e_1\ \{Q\}$ **:** *On distingue deux cas :*

- *Si* $\tau = \tau_1\ \mathsf{ref}$ *alors*

$$\widehat{e} = \lambda x_0.\lambda h : P(x_0).(\{\}, \lambda x.\widehat{e_1}, ?_1)$$

  avec $?_1 : Q(x_0, \{\}, \lambda x.\widehat{e_1})$.

- *Sinon*

$$\widehat{e} = \lambda x_0.\lambda h : P(x_0).(\{\}, \lambda x : \widehat{\tau}.\widehat{e_1}, ?_1)$$

  avec $?_1 : Q(x_0, \{\}, \lambda x : \widehat{\tau}.\widehat{e_1})$.

$e \equiv \{P\}\ \mathsf{Fun}\ \alpha \to e_1\ \{Q\}$ **:**

$$\widehat{e} = \lambda x_0.\lambda h : P(x_0).(\{\}, \Lambda\alpha.\widehat{e_1}, ?_1)$$

avec $?_1 : Q(x_0, \{\}, \Lambda\alpha.\widehat{e_1})$.

$e \equiv \{P\}$ rec $f\ (\vec{x} : \vec{\tau}) : \kappa$ $\{$variant $\nu\} = e_1\ \{Q\}$ **:** *On a* $\nu = (\phi, R) : Variant(A)$. *Alors*

$$\widehat{e} = \lambda x_0.\lambda h : P(x_0).(\{\}, (\mathsf{WF}\ A\ R\ ?_1\ \lambda\phi.\lambda f.\lambda\vec{x}.\widehat{e_1}), ?_2)$$

avec $?_1 : (well\_founded\ R)$ *et* $?_2 : Q(x_0, \{\}, (\mathsf{WF}\ A\ R\ ?_1\ \lambda\phi.\lambda f.\lambda\vec{x}.\widehat{e_1}))$.

Remarque. *Pour les trois dernières constructions, à savoir* fun, Fun *et* rec, *on n'a généralement ni pré-condition ni post-condition, ce qui simplifie beaucoup leur interprétation. En effet, on ne s'intéresse pas souvent à la fonction en tant que valeur, mais on spécifie plutôt son* corps, *en tant que calcul, les annotations correspondantes étant alors celles de* e_1.

$e \equiv \{P\}$ if e_1 then e_2 else e_3 $\{Q\}$ **:** *On a* $\Gamma \vdash_a e_1 : (\mathsf{bool}, \epsilon_1, P_1, Q_1)$, $\Gamma \vdash_a e_2 : (\tau, \epsilon_2, P_2, Q_2)$ *et* $\Gamma \vdash_a e_3 : (\tau, \epsilon_3, P_3, Q_3)$. *Alors*

$$\begin{array}{ll} \widehat{e} = \lambda x_0.\lambda h : P(x_0). & \mathsf{let}\ (x_1, b, q_1) = (\widehat{e_1}\ x_0\ ?_1)\ \mathsf{in} \\ & \mathsf{if}\ b\ \mathsf{then} \\ & \quad \mathsf{let}\ (x_2, v, q_2) = (\widehat{e_2}\ (x_0 \oplus x_1)\ ?_2)\ \mathsf{in}\ (x_0 \oplus x_1 \oplus x_2, v, ?_3) \\ & \mathsf{else} \\ & \quad \mathsf{let}\ (x_2, v, q_3) = (\widehat{e_3}\ (x_0 \oplus x_1)\ ?_4)\ \mathsf{in}\ (x_0 \oplus x_1 \oplus x_2, v, ?_3) \end{array}$$

avec $?_1 : P_1(x_0)$, $?_2 : P_2(x_0 \oplus x_1)$, $?_3 : Q(x_0, x_1 \oplus x_2, v)$ *et* $?_4 : P_3(x_0 \oplus x_1)$.

$e \equiv \{P\}$ let $x = e_1$ in e_2 $\{Q\}$ **:** *On sait que l'on a* $\Gamma \vdash_a e_1 : (\tau_1, \epsilon_1, P_1, Q_1)$ *et* $\Gamma, x : \tau_1 \vdash_a e_2 : (\tau, \epsilon_2, P_2, Q_2)$. *On distingue deux cas suivant le type de* e_1 *:*

- *Si* $\tau_1 \neq \_$ ref *alors*

$$\begin{array}{ll} \widehat{e} = \lambda x_0.\lambda h : P(x_0). & \mathsf{let}\ (x_1, x, q_1) = (\widehat{e_1}\ x_0\ ?_1)\ \mathsf{in} \\ & \mathsf{let}\ (x_2, v, q_2) = (\widehat{e_2}\ (x_0 \oplus x_1)\ ?_2)\ \mathsf{in} \\ & (x_1 \oplus x_2, v, ?_3) \end{array}$$

  avec $?_1 : P_1(x_0)$, $?_2 : P_2(x_0 \oplus x_1)$ *et* $?_3 : Q(x_0, x_1 \oplus x_2, v)$.

- *Si* $\tau_1 = \_$ ref *alors*

$$\begin{array}{ll} \widehat{e} = \lambda x_0.\lambda h : P(x_0). & \mathsf{let}\ (x_1, v_0, q_1) = (\widehat{e_1}\ x_0\ ?_1)\ \mathsf{in} \\ & \mathsf{let}\ (x_2, v, q_2) = (\widehat{e_2}\ (x_0 \oplus x_1 \oplus \{x = v_0\})\ ?_2)\ \mathsf{in} \\ & (x_1 \oplus x_2 \backslash x, v, ?_3) \end{array}$$

  avec $?_1 : P_1(x_0)$, $?_2 : P_2(x_0 \oplus x_1 \oplus \{x = v_0\})$ *et* $?_3 : Q(x_0, x_1 \oplus x_2 \backslash x, v)$.

$e \equiv \{P\}\ e_1\ ;\ e_2\ \{Q\}$ **:** *On utilise l'équivalence*

$$e \approx \mathsf{let}\ \_ = e_1\ \mathsf{in}\ e_2$$

pour se ramener à la traduction d'un let *(avec* e_1 : unit*).*

$e \equiv \{P\}$ while e_1 do $\{$invariant I variant $\nu\}$ e_2 done $\{Q\}$ **:** *On a* $\Gamma \vdash_a e_1 : (\mathsf{bool}, \epsilon_1, P_1, Q_1$ *et* $\Gamma \vdash_a e_1 : (\mathsf{unit}, \epsilon_2, P_2, Q_2$. *On a de plus* $\nu = (\phi, R) : Variant(A)$.

$$\begin{array}{l}
\widehat{e} = \lambda x_0.\lambda h : P(x_0).\ \mathsf{let}\ (x_1, _, q) = \\
\quad (\mathsf{WF}\ A\ R\ ?_1 \\
\quad\ \lambda\varphi.\lambda w.\lambda x_0.\lambda h_1 : I. \\
\quad\quad \mathsf{let}\ (x_1, b, q_1) = (\widehat{e_1}\ x_0\ ?_2)\ \mathsf{in} \\
\quad\quad \mathsf{if}\ b\ \mathsf{then} \\
\quad\quad\quad \mathsf{let}\ (x_2, _, q_2) = (\widehat{e_2}\ (x_0 \oplus x_1)\ ?_3)\ \mathsf{in} \\
\quad\quad\quad (w\ \phi(x_0 \oplus x_1 \oplus x_2)\ ?_4\ (x_0 \oplus x_1 \oplus x_2)\ ?_5) \\
\quad\quad \mathsf{else} \\
\quad\quad\quad (x_0 \oplus x_1, \mathsf{void}, (?_6, q_1)) \\
\quad \phi(x_0)\ x_0\ ?_7)\ \mathsf{in} \\
(x_1, \mathsf{void}, ?_8)
\end{array}$$

avec $?_1 : (well\_founded\ R)$, $?_2 : P_1(x_0)$, $?_3 : P_2(x_0 \oplus x_1)$, $?_4 : (R\ \phi(x_0 \oplus x_1 \oplus x_2)\ \varphi)$, $?_5 : I(x_0 \oplus x_1 \oplus x_2)$, $?_6 : I(x_0 \oplus x_1)$, $?_7 : I(x_0)$ *et* $?_8 : Q(x_0, x_1, \mathsf{void})$.

$e \equiv \{P\}$ ref e_1 $\{Q\}$ **:** *On a* $\Gamma \vdash_a e_1 : (\tau, \epsilon_1, P_1, Q_1)$. *Alors*

$$\widehat{e} = \lambda x_0.\lambda h : P(x_0).\mathsf{let}\ (x_1, v, q_1) = (\widehat{e_1}\ x_0\ ?_1)\ \mathsf{in}\ (x_1, v, ?_2)$$

avec $?_1 : P_1(x_0)$ *et* $?_2 : Q(x_0, x_1, v)$.

$e \equiv \{P\}\ !x\ \{Q\}$ **:**

$$\widehat{e} = \lambda x_0.\lambda h : P(x_0).(\{\}, x_0.x, ?_1)$$

avec $?_1 : Q(x_0, \{\}, x_0.x)$.

$e \equiv \{P\}\ x\ :=\ e_1\ \{Q\}$ **:** *On a* $\Gamma \vdash_a e_1 : (\tau, \epsilon_1, P_1, Q_1)$. *Alors*

$$\widehat{e} = \lambda x_0.\lambda h : P(x_0).\mathsf{let}\ (x_1, v, q_1) = (\widehat{e_1}\ x_0\ ?_1)\ \mathsf{in}\ (x_1 \oplus \{x = v\}, \mathsf{void}, ?_2)$$

avec $?_1 : P_1(x_0)$ *et* $?_2 : Q(x_0, x_1 \oplus \{x = v\}, \mathsf{void})$.

□

Remarque. On peut constater qu'une boucle while est traduite par une fonction récursive *terminale*. De manière générale, lorsque l'on traduit une boucle en une fonction récursive, on a le choix *a priori* entre la récursivité terminale ou non-terminale. Considérons par exemple le programme suivant calculant $x!$:

$$\begin{array}{l}
f := 1; \\
\mathsf{while}\ !x \neq 0\ \mathsf{do} \\
\quad f := !x \times !f; \\
\quad x := !x - 1; \\
\mathsf{done}
\end{array}$$

Une traduction par une fonction récursive terminale peut être la suivante :

$$\begin{array}{l} \mathsf{let\ rec}\ w\ f\ x\ = \\ \quad \mathsf{if}\ x \neq 0\ \mathsf{then}\ (w\ (x \times f)\ (x-1))\ \mathsf{else}\ f \\ \mathsf{in}\ (w\ 1) \end{array}$$

et une traduction par une fonction récursive non-terminale, plus naturelle, peut être :

$$\begin{array}{l} \mathsf{let\ rec}\ w\ x\ = \\ \quad \mathsf{if}\ x \neq 0\ \mathsf{then}\ x \times (w\ (x-1))\ \mathsf{else}\ 1 \end{array}$$

Mais bien que la deuxième version soit plus facile à saisir sur cet exemple — c'est exactement la définition mathématique de la factorielle — il n'y a pas de méthode générale pour l'obtenir à partir de la boucle impérative. De plus, cette seconde écriture n'est équivalente à la première que grâce à la commutativité et à l'associativité de la multiplication.

D'autre part, si les annotations de la boucle impérative sont les mêmes que celle de la fonction récursive terminale, une version récursive non-terminale a généralement des annotations complètement différentes. Pour s'en convaincre, le lecteur peut essayer d'annoter les trois programmes ci-dessus.

Enfin, la récursivité terminale est la plus efficace des deux, car elle ne nécessite pas de conserver sur la pile les arguments (et variables locales) de la fonction, et peut donc être compilée *par une boucle.* □

La justification de la définition de $\widehat{e}$ sera donnée dans la section suivante. Elle repose en grande partie sur le fait que la partie informative de $\widehat{e}$ est bien la traduction fonctionnelle $\overline{e}$ introduite section 1.3.2. Nous l'énonçons par la proposition suivante :

Proposition 2 *Soit Γ un environnement et e un programme tel que $\Gamma \vdash_a e : \kappa$. Alors on a l'égalité suivante :*

$$\mathcal{E}(\widehat{e}) = \overline{e}$$

Preuve. La preuve se fait par récurrence sur la structure de e et est immédiate par construction de $\widehat{e}$. Le seul point délicat concerne la boucle while et la construction let rec pour lesquelles on utilise l'égalité suivante :

$$\mathcal{E}(\mathsf{WF}\ A\ R\ ?\ F) = (\mathsf{Y}\ \mathcal{E}(F))$$

□

2.3.3 Correction

Lorsque l'on souhaite montrer la validité d'un formalisme tel que la logique de Floyd-Hoare, on commence par se donner une interprétation de la formule de correction $\{P\}\ e\ \{Q\}$, puis on établit un théorème disant en substance :

$$\vdash \{P\}\ e\ \{Q\} \quad \Longrightarrow \quad \models \{P\}\ e\ \{Q\}$$

c'est-à-dire « Si l'on peut dériver dans le système la formule de correction $\{P\}\ e\ \{Q\}$, alors l'interprétation de cette formule est valide ». Dans un formalisme du premier ordre où l'on s'intéresse à la correction totale, l'interprétation de la formule $\{P\}\ e\ \{Q\}$ peut être la suivante :

$$\{P\}\ e\ \{Q\} \stackrel{\text{def}}{=} \forall s. P(s) \Rightarrow \exists s'. eval(e,s) = s' \wedge Q(s,s') \tag{2.2}$$

Mais dans notre cas, l'ordre supérieur impose un raffinement de l'interprétation ci-dessus. En effet, le résultat d'un programme peut être une fonction, et l'on doit exprimer dans ce cas qu'il vérifie bien lui-même sa spécification. Inversement, lorsque l'on exprime qu'une fonction vérifie bien sa spécification, on doit le faire en supposant qu'il en est de même pour ses arguments, et ceux-ci peuvent être à leur tour des fonctions sur lesquelles portent des spécifications.

D'autre part, dans l'interprétation traditionnelle (2.2) ci-dessus, les prédicats P et Q portent sur les valeurs des états s et s', qui ne sont composés que d'objets vivant dans les types de bases (entiers, booléens et flottants généralement). Mais dans notre cas une référence peut contenir également une fonction, et les prédicats P et Q peuvent donc porter sur une valeur fonctionnelle. Si ces prédicats ont été introduits dans un contexte où chaque référence de type τ ref était représentée par une variable de type $\widehat{\tau}$, il convient d'*interpréter* ces prédicats comme portant plutôt sur les traductions fonctionnelles des références i.e. chaque référence de type τ ref étant représentée par une variable de type $\overline{\tau}$. Un prédicat P sera donc interprété par la proposition $\overline{P}$, qui désigne l'interprétation de P dans le monde extrait (voir page 74).

Nous sommes donc amenés à introduire une double interprétation des programmes et des valeurs, définie par récurrence sur leur type.

Définition 38 *Les propositions $[\![\_]\!]_\kappa$ et $[\![\_]\!]_\tau$ sont définies mutuellement récursivement de la façon suivante :*

$$\begin{array}{lcl}
[\![e]\!]_{(r:\tau,(\rho,\omega),P,Q)} & \stackrel{def}{=} & \forall\theta.\ [\![\theta]\!]_\rho \Rightarrow \overline{P}(\overline{\theta(\rho)}) \Rightarrow \exists\theta', v. \\
& & \quad \theta.e \longmapsto^\star \theta'.v \wedge [\![\theta']\!]_\omega \wedge [\![val(\theta',v)]\!]_\tau \wedge \overline{Q}(\overline{\theta(\rho)}, \overline{\theta'(\omega)}, \overline{val(\theta',v)}) \\
 & & \\
[\![v]\!]_{(\tau_1,\ldots,\tau_n)c} & \stackrel{def}{=} & \mathsf{True} \\
[\![v]\!]_{\tau\ \mathsf{ref}} & \stackrel{def}{=} & [\![v]\!]_\tau \\
[\![v]\!]_{(x:\tau\ \mathsf{ref})\to\kappa} & \stackrel{def}{=} & \forall r:\tau\ \mathsf{ref}.\ [\![(v\ r)]\!]_{\kappa[x\leftarrow r]} \\
[\![v]\!]_{(x:\tau)\to\kappa} & \stackrel{def}{=} & \forall x:\tau.\ [\![x]\!]_\tau \Rightarrow [\![(v\ x)]\!]_\kappa
\end{array}$$

avec $[\![\theta]\!]_{\{x_1,\ldots,x_n\}} \stackrel{def}{=} \forall i \in \{1,\ldots,n\}.\ [\![\theta(x_i)]\!]_{\Gamma(x_i)}$

On constate que dans le cas d'un programme du premier ordre on retrouve exactement la formulation (2.2). En effet, les prédicats $[\![\theta]\!]_\rho$, $[\![\theta']\!]_\omega$ et $[\![val(\theta',v)]\!]_\tau$ valent tous les trois True et d'autre part $\overline{P}(\overline{\theta(\rho)}) = P(\theta(\rho))$ et $\overline{Q}(\overline{\theta(\rho)}, \overline{\theta'(\omega)}, \overline{val(\theta',v)}) = Q(\theta(\rho), \theta'(\omega), val(\theta',v))$.

Afin de parvenir au résultat final de correction de la méthode, qui utilisera le théorème de réalisabilité, on va se rapprocher de la notion classique de réalisabilité fonctionnelle en introduisant la notion suivante de *réalisabilité impérative*, définie sur les termes extraits.

Définition 39 (réalisabilité impérative) *Soit Γ un environnement bien formé et τ et κ des types de valeur et de calcul bien formés dans Γ. Pour tout v de type $\overline{\tau}$ et tout e de type $\overline{\kappa}$ on définit les prédicats $v \ \mathsf{r}_i \ \tau$ et $e \ \mathsf{r}_i \ \kappa$ mutuellement récursivement de la manière suivante :*

$$
\begin{array}{lcl}
e \ \mathsf{r}_i \ (r : \tau, (\rho, \omega), P, Q) & \stackrel{def}{=} & \forall x : \overline{\rho}.\ x \ \mathsf{r}_i \ \rho \Rightarrow \overline{P}(x) \Rightarrow \\
& & \quad \mathsf{let}\ (y, r) = (e\ x)\ \mathsf{in}\ y \ \mathsf{r}_i \ \omega \wedge r \ \mathsf{r}_i \ \tau \wedge \overline{Q}(x, y, r)
\end{array}
$$

$$
\begin{array}{lcl}
v \ \mathsf{r}_i \ (\tau_1, \ldots, \tau_n)c & \stackrel{def}{=} & \mathsf{True} \\
v \ \mathsf{r}_i \ \tau \ \mathsf{ref} & \stackrel{def}{=} & v \ \mathsf{r}_i \ \tau \\
v \ \mathsf{r}_i \ (x : \tau \ \mathsf{ref}) \rightarrow \kappa & \stackrel{def}{=} & \forall x.\ (v\ x) \ \mathsf{r}_i \ \kappa \\
v \ \mathsf{r}_i \ (x : \tau) \rightarrow \kappa & \stackrel{def}{=} & \forall x : \overline{\tau}.\ x \ \mathsf{r}_i \ \tau \Rightarrow (v\ x) \ \mathsf{r}_i \ \kappa
\end{array}
$$

avec $x \ \mathsf{r}_i \ \{x_1, \ldots, x_n\} \stackrel{def}{=} \forall i \in \{1, \ldots, n\}.\ x.x_i \ \mathsf{r}_i \ \Gamma(x_i)$

On aura besoin du lemme de substitution suivant :

Lemme 8 *Soit Γ un environnement bien formé et κ un type de calcul bien formé dans Γ. Alors pour tout e de type $\overline{\kappa}$ on a*

$$e \ \mathsf{r}_i \ \kappa \quad \Longleftrightarrow \quad e[x_1 \leftarrow x_2] \ \mathsf{r}_i \ \kappa[x_1 \leftarrow x_2]$$

Preuve. Il est important de noter que les environnements ne sont pas les mêmes à gauche et à droite de l'équivalence. Plus précisément, si Γ dénote l'environnement dans lequel on a $e \ \mathsf{r}_i \ \kappa$ alors la proposition $e[x_1 \leftarrow x_2] \ \mathsf{r}_i \ \kappa[x_1 \leftarrow x_2]$ doit être considérée dans l'environnement $\Gamma[x_1 \leftarrow x_2]$. La preuve se fait par récurrence sur la structure du type. Examinons chaque construction :

$(\tau_1, \ldots, \tau_n)c$:

Le résultat est immédiat car la proposition vaut True de part et d'autre.

$\tau \ \mathsf{ref}$:

On a $v \ \mathsf{r}_i \ \tau \ \mathsf{ref} \stackrel{\text{def}}{=} v \ \mathsf{r}_i \ \tau$ d'une part et

$$
\begin{array}{rcl}
v[x_1 \leftarrow x_2] \ \mathsf{r}_i \ (\tau \ \mathsf{ref})[x_1 \leftarrow x_2] & = & v[x_1 \leftarrow x_2] \ \mathsf{r}_i \ \tau[x_1 \leftarrow x_2] \ \mathsf{ref} \\
& \stackrel{\text{def}}{=} & v[x_1 \leftarrow x_2] \ \mathsf{r}_i \ \tau[x_1 \leftarrow x_2]
\end{array}
$$

d'autre part, et le résultat est immédiat par hypothèse de récurrence.

$(x : \tau \ \mathsf{ref}) \rightarrow \kappa$:

On a d'une part $v \ \mathsf{r}_i \ (x : \tau \ \mathsf{ref}) \rightarrow \kappa \stackrel{\text{def}}{=} (v\ x) \ \mathsf{r}_i \ \kappa$. D'autre part, on peut toujours supposer x différent de x_1 (par α-conversion, car x est une variable liée), et dès lors on a

$$
\begin{array}{rl}
& v[x_1 \leftarrow x_2] \ \mathsf{r}_i \ ((x : \tau \ \mathsf{ref}) \rightarrow \kappa)[x_1 \leftarrow x_2] \\
= & v[x_1 \leftarrow x_2] \ \mathsf{r}_i \ (x : \tau[x_1 \leftarrow x_2] \ \mathsf{ref}) \rightarrow \kappa[x_1 \leftarrow x_2] \\
\stackrel{\text{def}}{=} & (v[x_1 \leftarrow x_2]\ x) \ \mathsf{r}_i \ \kappa[x_1 \leftarrow x_2] \\
= & (v\ x)[x_1 \leftarrow x_2] \ \mathsf{r}_i \ \kappa[x_1 \leftarrow x_2]
\end{array}
$$

et le résultat est immédiat par hypothèse de récurrence.

$(x : \tau) \to \kappa$ **:**

On a $v \;\mathsf{r}_i\; (x : \tau) \to \kappa \stackrel{\text{def}}{=} \forall x : \overline{\tau}.\; x \;\mathsf{r}_i\; \tau \Rightarrow (v\; x) \;\mathsf{r}_i\; \kappa$. Comme pour le cas précédent, on peut supposer par α-conversion que x est différent de x_1. Dès lors, on a

$$\begin{array}{ll} & v[x_1 \leftarrow x_2] \;\mathsf{r}_i\; ((x : \tau) \to \kappa)[x_1 \leftarrow x_2] \\ = & v[x_1 \leftarrow x_2] \;\mathsf{r}_i\; (x : \tau[x_1 \leftarrow x_2]) \to \kappa[x_1 \leftarrow x_2] \\ \stackrel{\text{def}}{=} & \forall x : \overline{\tau[x_1 \leftarrow x_2]}.\; x \;\mathsf{r}_i\; \tau[x_1 \leftarrow x_2] \Rightarrow (v\; x) \;\mathsf{r}_i\; \kappa[x_1 \leftarrow x_2] \end{array}$$

Montrons la première implication. Soit x de type $\overline{\tau[x_1 \leftarrow x_2]}$ tel que $x \;\mathsf{r}_i\; \tau[x_1 \leftarrow x_2]$. Alors par hypothèse de récurrence on a $x \;\mathsf{r}_i\; \tau$ (x est vu maintenant comme faisant partie de l'environnement Γ, et non $\Gamma[x_1 \leftarrow x_2]$). Dès lors on a $(v\; x) \;\mathsf{r}_i\; \kappa$, et donc $(v\; x) \;\mathsf{r}_i\; \kappa[x_1 \leftarrow x_2]$ par hypothèse de récurrence. La seconde implication se montre de manière similaire.

$(r : \tau, (\rho, \omega), P, Q)$ **:**

Par définition, on a

$$\begin{array}{lll} e \;\mathsf{r}_i\; (r : \tau, (\rho, \omega), P, Q) & \stackrel{\text{def}}{=} & \forall x : \overline{\rho}.\; x \;\mathsf{r}_i\; \rho \Rightarrow \overline{P}(x) \Rightarrow \\ & & \quad \mathsf{let}\; (y, r) = (e\; x)\; \mathsf{in}\; y \;\mathsf{r}_i\; \omega \wedge r \;\mathsf{r}_i\; \tau \wedge \overline{Q}(x, y, r) \end{array}$$

D'autre part, on a

$$\begin{array}{ll} & e[x_1 \leftarrow x_2] \;\mathsf{r}_i\; ((r : \tau, (\rho, \omega), P, Q))[x_1 \leftarrow x_2] \\ \stackrel{\text{def}}{=} & \forall x : \overline{\rho[x_1 \leftarrow x_2]}.\; x \;\mathsf{r}_i\; \rho[x_1 \leftarrow x_2] \Rightarrow \overline{P[x_1 \leftarrow x_2]}(x) \Rightarrow \\ & \quad \mathsf{let}\; (y, r) = (e[x_1 \leftarrow x_2]\; x)\; \mathsf{in} \\ & \quad\quad y \;\mathsf{r}_i\; \omega[x_1 \leftarrow x_2] \wedge r \;\mathsf{r}_i\; \tau[x_1 \leftarrow x_2] \wedge \overline{Q[x_1 \leftarrow x_2]}(x, y, r) \end{array}$$

Montrons la première implication. Soit x de type $\overline{\rho[x_1 \leftarrow x_2]}$ tel que $x \;\mathsf{r}_i\; \rho[x_1 \leftarrow x_2]$ et $\overline{P[x_1 \leftarrow x_2]}(x)$. Posons $x' = x[x_2 \leftarrow x_1]$. Alors par hypothèse de récurrence on a $x' \;\mathsf{r}_i\; \rho$, et d'autre part on a $\overline{P}(x')$. Dès lors, si $(y', r') = (e\; x')$ on a $y' \;\mathsf{r}_i\; \omega \wedge r' \;\mathsf{r}_i\; \tau \wedge \overline{Q}(x', y', r')$. Alors par hypothèse de récurrence on a $y'[x_1 \leftarrow x_2] \;\mathsf{r}_i\; \omega[x_1 \leftarrow x_2]$ et $r'[x_1 \leftarrow x_2] \;\mathsf{r}_i\; \tau[x_1 \leftarrow x_2]$. D'autre part on a également $\overline{Q[x_1 \leftarrow x_2]}(x'[x_1 \leftarrow x_2], y'[x_1 \leftarrow x_2], r'[x_1 \leftarrow x_2])$. Or $(y', r')[x_1 \leftarrow x_2] = (e\; x')[x_1 \leftarrow x_2] = (e[x_1 \leftarrow x_2]\; x)$. Donc si $(y, r) = (e[x_1 \leftarrow x_2]\; x)$ on a $y \;\mathsf{r}_i\; \omega[x_1 \leftarrow x_2]$, $r \;\mathsf{r}_i\; \tau[x_1 \leftarrow x_2]$ et $\overline{Q[x_1 \leftarrow x_2]}(x, y, r)$, ce qui est la proposition attendue. La seconde implication se montre de façon similaire.

□

Cette notion de réalisabilité impérative est rattachée à l'interprétation $[\![\_]\!]$ précédemment définie par le résultat suivant.

Lemme 9 *Soit Γ un environnement et κ un type de calcul bien formé dans Γ. Alors*

$$\overline{e} \;\mathsf{r}_i\; \kappa \quad\Longleftrightarrow\quad [\![e]\!]_\kappa$$

PREUVE. On montre par récurrence mutuelle sur la structure des types que pour tout type de calcul κ et pour tout programme e de type κ on a :

$$\overline{e}\ \mathsf{r}_i\ \kappa \quad \Longleftrightarrow \quad [\![e]\!]_\kappa$$

et que pour tout type de valeur τ et toute *valeur* v de type τ on a :

$$\overline{v}\ \mathsf{r}_i\ \tau \quad \Longleftrightarrow \quad [\![v]\!]_\tau$$

Examinons chaque construction de type :

$(\tau_1, \ldots, \tau_n)c$ **:**

Le résultat est immédiat car $[\![v]\!]_c = \overline{v}\ \mathsf{r}_i\ c = \mathsf{True}$.

$\tau\ \mathsf{ref}$ **:**

On a $[\![v]\!]_{\tau\ \mathsf{ref}} = [\![v]\!]_\tau$ et $\overline{v}\ \mathsf{r}_i\ \tau\ \mathsf{ref} = \overline{v}\ \mathsf{r}_i\ \tau$ et la récurrence est donc immédiate.

$(x : \tau\ \mathsf{ref}) \rightarrow \kappa$ **:**

On doit montrer l'équivalence

$$(\overline{v}\ x)\ \mathsf{r}_i\ \kappa \quad \Longleftrightarrow \quad \forall r : \tau\ \mathsf{ref}.\ [\![(v\ r)]\!]_{\kappa[x \leftarrow r]}$$

Supposons que $(\overline{v}\ x)\ \mathsf{r}_i\ \kappa$ et soit r une référence. Par le lemme de substitution 8 on a $(\overline{v}\ x)[x \leftarrow r]\ \mathsf{r}_i\ \kappa[x \leftarrow r]$. Or $(\overline{v}\ x)[x \leftarrow r] = (\overline{v[x \leftarrow r]}\ r) = \overline{(v\ r)}$ donc on a $\overline{(v\ r)}\ \mathsf{r}_i\ \kappa[x \leftarrow r]$. D'où par hypothèse de récurrence $[\![(v\ r)]\!]_{\kappa[x \leftarrow r]}$. La réciproque se montre en utilisant les mêmes équivalences et les mêmes identités.

$(x : \tau) \rightarrow \kappa$ **:**

Montrons la première implication. On suppose $\overline{v}\ \mathsf{r}_i\ (x : \tau) \rightarrow \kappa$ c'est-à-dire :

$$\forall x : \overline{\tau}.\ x\ \mathsf{r}_i\ \tau \Rightarrow (\overline{v}\ x)\ \mathsf{r}_i\ \kappa \tag{2.3}$$

et on cherche à montrer $[\![v]\!]_{(x:\tau)\rightarrow\kappa}$ c'est-à-dire :

$$\forall x : \tau.\ [\![x]\!]_\tau \Rightarrow [\![(v\ x)]\!]_\kappa \tag{2.4}$$

Soit donc x de type τ tel que $[\![x]\!]_\tau$. Par hypothèse de récurrence on a $x\ \mathsf{r}_i\ \tau$, où x désigne ici la variable de $\overline{\Gamma}$, et donc $(\overline{v}\ x)\ \mathsf{r}_i\ \kappa$ d'après (2.3). Or $(\overline{v}\ x) = \overline{(v\ x)}$ car v est une valeur et x une variable (voir la définition de la traduction, page 49). Dès lors on a $[\![(v\ x)]\!]_\kappa$ par hypothèse de récurrence.

Réciproquement, supposons (2.4) et montrons (2.3). Soit donc x de type $\overline{\tau}$ tel que $x\ \mathsf{r}_i\ \tau$. Alors par hypothèse de récurrence on a $[\![x]\!]_\tau$ d'où $[\![(v\ x)]\!]_\kappa$ par (2.4). Par hypothèse de récurrence on a alors $\overline{(v\ x)}\ \mathsf{r}_i\ \kappa$, d'où $(\overline{v}\ x)\ \mathsf{r}_i\ \kappa$ par le même argument que ci-dessus.

$(r : \tau, (\rho, \omega), P, Q)$ **:**

Montrons la première implication. On suppose $\overline{e}\ \mathsf{r}_i\ (r : \tau, (\rho, \omega), P, Q)$ c'est-à-dire :

$$\forall x : \overline{\rho}.\ x\ \mathsf{r}_i\ \rho \Rightarrow \overline{P}(x) \Rightarrow \mathsf{let}\ (y, r) = (\overline{e}\ x)\ \mathsf{in}\ y\ \mathsf{r}_i\ \omega \wedge r\ \mathsf{r}_i\ \tau \wedge \overline{Q}(x, y, r) \tag{2.5}$$

et on cherche à montrer $[\![e]\!]_{(r:\tau,(\rho,\omega),P,Q)}$, c'est-à-dire :

$$\begin{aligned}\forall\theta.\ [\![\theta]\!]_\rho \Rightarrow \overline{P}(\overline{\theta(\rho)}) \Rightarrow \exists\theta', v.&\\ \theta.e \longmapsto^\star \theta'.v \wedge [\![\theta']\!]_\omega \wedge [\![val(\theta',v)]\!]_\tau \wedge \overline{Q}(\overline{\theta(\rho)}, \overline{\theta'(\omega)}, \overline{val(\theta',v)})&\end{aligned} \tag{2.6}$$

Soit donc θ tel que $[\![\theta]\!]_\rho$ et $\overline{P}(\overline{\theta(\rho)})$. Soit $x = \overline{\theta(\rho)}$. On a $[\![\theta]\!]_\rho$ donc par hypothèse de récurrence appliquée aux types des références de ρ on a $x\ \mathsf{r}_i\ \rho$. Les hypothèses de (2.5) sont donc vérifiées. Soient alors $(y, r) = (\overline{e}\ x)$. D'après le Théorème 2 page 54 on a

$$\exists\theta', v.\ \theta.e \longmapsto^\star \theta'.v \quad \text{avec } y = \overline{\theta'(\omega)} \text{ et } r = \overline{val(\theta',v)}$$

Or $r\ \mathsf{r}_i\ \tau$ d'après (2.5) donc $\overline{val(\theta',v)}\ \mathsf{r}_i\ \tau$, d'où $[\![val(\theta',v)]\!]_\tau$ par hypothèse de récurrence. De même on a d'après (2.5) $y\ \mathsf{r}_i\ \omega$ i.e. $\overline{\theta'(\omega)}\ \mathsf{r}_i\ \omega$ donc par hypothèse de récurrence appliquée aux types des références de ω, on a $[\![\theta']\!]_\omega$. Enfin on a $\overline{Q}(x, y, r)$ d'après (2.5) donc $\overline{Q}(\overline{\theta(\rho)}, \overline{\theta'(\omega)}, \overline{val(\theta',v)})$, c'est-à-dire $[\![\theta, \theta', val(\theta',v)]\!]_{r:\tau,Q}$.

Réciproquement, supposons (2.6) et montrons (2.5). Soit donc x de type $\overline{\rho}$ tel que $x\ \mathsf{r}_i\ \rho$ et $\overline{P}(x)$. Soit θ un état tel que $\overline{\theta(\rho)} = x$. On a $x\ \mathsf{r}_i\ \rho$ donc par hypothèse de récurrence appliquée aux types des références de ρ, on a $[\![\theta]\!]_\rho$. D'autre part on a $\overline{P}(\overline{\theta(\rho)})$. Donc d'après (2.6) on a $\theta.e \longmapsto^\star \theta'.v$. Dès lors, par le Théorème 1 page 53 on a $(\overline{e}\ x) = (\overline{\theta'(\omega)}, \overline{val(\theta',v)})$. Notons (y, r) ce couple. D'après (2.6) on a $[\![\theta']\!]_\omega$ donc par hypothèse de récurrence appliquée aux types des références de ω, on a $\overline{\theta'(\omega)}\ \mathsf{r}_i\ \omega$ i.e. $y\ \mathsf{r}_i\ \omega$. De même d'après (2.6) on a $[\![val(\theta',v)]\!]_\tau$ donc par hypothèse de récurrence on a $\overline{val(\theta',v)}\ \mathsf{r}_i\ \tau$, c'est-à-dire $r\ \mathsf{r}_i\ \tau$. Enfin par (2.6) on a $\overline{Q}(\overline{\theta(\rho)}, \overline{\theta'(\omega)}, \overline{val(\theta',v)})$ c'est-à-dire $\overline{Q}(x, y, r)$.

□

Notre notion de réalisabilité impérative est rattachée à la notion de réalisabilité fonctionnelle (voir page 74) par le résultat suivant.

Lemme 10 *Soit Γ un environnement et κ un type de calcul bien formé dans Γ. Alors pour tout x de type $\overline{\kappa}$ on a*

$$x\ \mathsf{r}_f\ \widehat{\kappa} \quad\Longleftrightarrow\quad x\ \mathsf{r}_i\ \kappa$$

Preuve. Comme pour le Lemme 9, généralisons le résultat : montrons par récurrence sur la structure du type que pour tout e de type $\overline{\kappa}$ on a :

$$e\ \mathsf{r}_f\ \widehat{\kappa} \quad\Longleftrightarrow\quad e\ \mathsf{r}_i\ \kappa$$

et que pour tout v de type $\overline{\tau}$ on a :

$$v\ \mathsf{r}_f\ \widehat{\tau} \quad\Longleftrightarrow\quad v\ \mathsf{r}_i\ \tau$$

$(\tau_1, \ldots, \tau_n)c$:
Le résultat est immédiat car $v\ \mathsf{r}_f\ \widehat{c} = v\ \mathsf{r}_i\ c = \mathsf{True}$.

τ ref :
Soit v de type $\overline{\tau\ \mathsf{ref}}$, donc de type $\overline{\tau}$. On a $v\ \mathsf{r}_f\ \widehat{\tau\ \mathsf{ref}} = v\ \mathsf{r}_f\ \widehat{\tau}$ et $v\ \mathsf{r}_i\ \tau\ \mathsf{ref} = v\ \mathsf{r}_i\ \tau$ et la récurrence est donc immédiate.

$(x : \tau \ \mathsf{ref}) \rightarrow \kappa$ **:**

Soit v de type $\overline{(x : \tau \ \mathsf{ref}) \rightarrow \kappa}$, donc de type $\forall x.\overline{\kappa}$. On a $v \ \mathsf{r}_f \ \widehat{(x : \tau \ \mathsf{ref}) \rightarrow \kappa} = (v \ x) \ \mathsf{r}_f \ \widehat{\kappa}$ et $v \ \mathsf{r}_i \ (x : \tau \ \mathsf{ref}) \rightarrow \kappa = (v \ x) \ \mathsf{r}_i \ \kappa$, et la récurrence est donc immédiate.

$(x : \tau) \rightarrow \kappa$ **:**

Soit v de type $\overline{(x : \tau) \rightarrow \kappa}$, donc de type $(x : \overline{\tau}) \rightarrow \overline{\kappa}$. Montrons la première implication. On suppose $v \ \mathsf{r}_f \ \widehat{(x : \tau) \rightarrow \kappa}$ i.e. $v \ \mathsf{r}_f \ \forall x : \widehat{\tau}.\ \widehat{\kappa}$, c'est-à-dire :

$$\forall x : \overline{\tau}.\ x \ \mathsf{r}_f \ \widehat{\tau} \Rightarrow (v \ x) \ \mathsf{r}_f \ \widehat{\kappa} \tag{2.7}$$

et l'on veut montrer $v \ \mathsf{r}_i \ (x : \tau) \rightarrow \kappa$ c'est-à-dire :

$$\forall x : \overline{\tau}.\ x \ \mathsf{r}_i \ \tau \Rightarrow (v \ x) \ \mathsf{r}_i \ \kappa \tag{2.8}$$

Soit donc x de type $\overline{\tau}$ tel que $x \ \mathsf{r}_i \ \tau$. Alors par hypothèse de récurrence on a $x \ \mathsf{r}_f \ \widehat{\tau}$. Dès lors par (2.7) on a $(v \ x) \ \mathsf{r}_f \ \widehat{\kappa}$ et par hypothèse de récurrence on a donc $(v \ x) \ \mathsf{r}_i \ \kappa$, d'où (2.8).

On montre de même que (2.8) implique (2.7).

$(r : \tau, (\rho, \omega), P, Q)$ **:**

Notons κ ce type. Soit e de type $\overline{\kappa}$, donc de type $\overline{\rho} \rightarrow \overline{\omega} \times \overline{\tau}$. Montrons la première implication. On suppose $e \ \mathsf{r}_f \ \widehat{\kappa}$ i.e. $e \ \mathsf{r}_f \ \forall x : \widehat{\rho}.P(x \Rightarrow \{(y, r) \mid Q(x, y, r)\}$ c'est-à-dire :

$$\forall x : \overline{\rho}.\ x \ \mathsf{r}_f \ \widehat{\rho} \Rightarrow \overline{P}(x) \Rightarrow \mathsf{let} \ (y, r) = (e \ x) \ \mathsf{in} \ y \ \mathsf{r}_f \ \widehat{\omega} \wedge r \ \mathsf{r}_f \ \widehat{\tau} \wedge \overline{Q}(x, y, r) \tag{2.9}$$

et l'on veut montrer que $e \ \mathsf{r}_i \ \kappa$ c'est-à-dire :

$$\forall x : \overline{\rho}.\ x \ \mathsf{r}_i \ \rho \Rightarrow \overline{P}(x) \Rightarrow \mathsf{let} \ (y, r) = (e \ x) \ \mathsf{in} \ y \ \mathsf{r}_i \ \omega \wedge r \ \mathsf{r}_i \ \tau \wedge \overline{Q}(x, y, r) \tag{2.10}$$

Soit donc x de type $\overline{\rho}$ tel que $x \ \mathsf{r}_i \ \rho$ et $\overline{P}(x)$. On a $x \ \mathsf{r}_i \ \rho$ donc par hypothèse de récurrence on a $x \ \mathsf{r}_f \ \widehat{\rho}$. Les hypothèses de (2.9) sont donc vérifiées. Soit alors $(y, r) = (e \ x)$. Par (2.9) on a $y \ \mathsf{r}_f \ \widehat{\omega}$ donc par hypothèse de récurrence on a $y \ \mathsf{r}_i \ \omega$. De même on a par (2.9) $r \ \mathsf{r}_f \ \widehat{\tau}$ donc par hypothèse de récurrence on a $r \ \mathsf{r}_i \ \tau$. D'où finalement (2.10).

On montre que (2.10) implique (2.9) de manière similaire.

□

On en déduit immédiatement le corollaire suivant.

Corollaire 1 *Soit* Γ *un environnement et* κ *un type de calcul bien formé dans* Γ. *Alors pour toute preuve* e *dans* CIC*,*

$$\widehat{\Gamma} \vdash_{\mathrm{CIC}} e : \widehat{\kappa} \quad \Longrightarrow \quad \mathcal{E}(e) \ \mathsf{r}_i \ \kappa$$

PREUVE. La preuve est immédiate : si e est une preuve de $\widehat{\kappa}$, alors par le théorème de réalisabilité on a $\mathcal{E}(e) \ \mathsf{r}_f \ \widehat{\kappa}$ d'où, par le Lemme 10, $\mathcal{E}(e) \ \mathsf{r}_i \ \kappa$. □

En combinant les éléments précédents, on obtient donc finalement le théorème qui valide notre méthode.

Théorème 4 *Soit Γ un environnement bien formé et e un programme tel que $\Gamma \vdash_a e : \kappa$. Si $\widehat{e}$ peut être complétée en une preuve de $\widehat{\kappa}$, alors la proposition $[\![e]\!]_\kappa$ est valide.*

PREUVE. Supposons que $\widehat{e}$ soit complétée en une preuve π de $\widehat{\kappa}$. Alors par le Corollaire 1 ci-dessus on a $\mathcal{E}(\pi)\ \mathsf{r}_i\ \kappa$. Or d'après la Proposition 2 page 78 on a $\mathcal{E}(\pi) = \overline{e}$, d'où $\overline{e}\ \mathsf{r}_i\ \kappa$. Et donc finalement $[\![e]\!]_\kappa$ par le Lemme 9. □

Validité des assertions internes

Telle que nous l'avons définie, la notion de correction ne prend en compte que les assertions les plus *externes* du programme. Plus précisément, ce sont celles qui apparaissent dans son type de calcul. Rien n'est spécifié dans l'interprétation $[\![e]\!]_\kappa$ concernant les pré- et post-conditions portant sur les *sous-expressions* de e. Ainsi dans le programme suivant

$$\mathsf{begin}\ e_1\ \{Q_1\};\ e_2\ \mathsf{end}\ \{Q\}$$

la formule de correction ne fait mention que de la post-condition Q et en aucune façon de la post-condition Q_1. Cela est cohérent avec le fait que le type annoté d'un programme doit exprimer complètement sa spécification, indépendamment de la manière dont il est codé, et donc en particulier de la manière dont ses sous-expressions sont annotées. Cela ne signifie pas que les annotations internes soient superflues. Elles sont mêmes parfois nécessaires, comme nous le montrerons dans la section 2.4.

Toutefois, il peut s'avérer que l'on veuille établir une propriété à un certain point du programme. L'utilisation d'une pré-condition sur la sous-expression correspondante convient alors parfaitement. Mais pour plus de clarté nous avons introduit en pratique une construction assert permettant d'engendrer une obligation de preuve à un endroit quelconque du programme — la plupart des langages de programmation permettent de tester une condition booléenne grâce à une instruction du même nom. La construction assert est décrite page 113.

2.3.4 Exemples et comparaison avec la logique de Floyd-Hoare

Afin de bien illustrer les points communs et les différences entre notre méthodologie et la logique de Floyd-Hoare, nous allons détailler les obligations de preuves obtenues sur quelques cas particuliers. Des exemples complets seront détaillés dans la section 4 consacrée aux études de cas.

Dans les exemples qui suivent, les annotations des programmes ne sont pas complètes. En conséquence, les termes de preuve partiels que nous donnons comportent parfois des constructions let in qui peuvent être réduites, entraînant des substitutions. Nous montrerons dans la section suivante consacrée à la complétude que ces substitutions peuvent apparaître de façon équivalente dans une annotation complète des programmes fondée sur un calcul de plus faibles pré-conditions.

Affectations

Prenons pour commencer le cas de l'affectation, car il joue un rôle important dans la logique de Floyd-Hoare en tant qu'unique axiome. Nous distinguerons deux cas, l'un donnant une analogie avec la logique de Floyd-Hoare, l'autre une différence.

1. Le premier cas est celui d'une affectation simple

$$e \quad \equiv \quad \{P\}\ x := E\ \{Q\} \tag{2.11}$$

où E est une expression purement fonctionnelle quelconque, comprenant éventuellement des déréférences, soit $!y_1, !y_2, \dots, !y_k$. Alors l'interprétation de e est la suivante :

$$\begin{aligned}\widehat{e} &= \lambda x_0.\lambda h : P(x_0).\mathsf{let}\ v = (\widehat{E}\ x_0)\ \mathsf{in}\ (\{x = v\}, \mathsf{void}, ?) \\ &= \lambda x_0.\lambda h : P(x_0).\mathsf{let}\ v = E[!y \leftarrow x_0.y]\ \mathsf{in}\ (\{x = v\}, \mathsf{void}, ?) \\ &= \lambda x_0.\lambda h : P(x_0).(\{x = E[!y \leftarrow x_0.y]\}, \mathsf{void}, ?)\end{aligned}$$

où x_0 est un enregistrement contenant les valeurs de x et des y_i, et où y décrit l'ensemble des variables y_i pour $i = 1, \dots, k$. L'obligation de preuve est donc :

$$\forall x_0.P(x_0) \Rightarrow Q(x_0, x_0 \oplus \{x = E[!y \leftarrow x_0.y]\}, \mathsf{void})$$

Si par exemple l'expression E est $!x + 1$ alors $\widehat{e}$ se réduit à $\lambda x.\lambda h : P.(x + 1, \mathsf{void}, ?)$ et l'obligation de preuve à $P(x) \Rightarrow Q(x, x + 1, \mathsf{void})$, soit l'obligation obtenue avec l'axiomatique de Floyd-Hoare en combinant les règles de conséquence et d'affectation :

$$\dfrac{P(x) \Rightarrow Q(x+1) \qquad \dfrac{}{\vdash \{Q(x+1)\}\ x := !x+1\ \{Q(x)\}}(AFF)}{\vdash \{P(x)\}\ x := !x+1\ \{Q(x)\}}(CONS)$$

2. Le deuxième cas est celui d'une affectation de la forme

$$e \quad \equiv \quad \{P\}\ x := (f\ a)\ \{Q\} \tag{2.12}$$

où f est une fonction quelconque. Supposons de plus que la fonction f ait une spécification — c'est-à-dire un type — de la forme :

$$f \quad : \quad (x : A) \rightarrow (r : B, \epsilon, P_f, Q_f)$$

Supposons enfin que l'argument a soit purement fonctionnel, pour simplifier, et que l'on annote l'appel de fonction avec la post-condition Q_f, c'est-à-dire :

$$e \quad \equiv \quad \{P\}\ x := \{P_f[x \leftarrow a]\}\ (f\ a)\ \{Q_f[x \leftarrow a]\}\ \{Q\}$$

une telle annotation pouvant être insérée automatiquement. Alors l'interprétation $\widehat{e}$ est de la forme :

$$\begin{aligned}\widehat{e} \quad = \quad \lambda x_0.\lambda h : P(x_0).\ &\mathsf{let}\ (x_1, v, q_f) = \\ &\qquad \mathsf{let}\ (x_1, r, q_f) = (f\ a\ x_0\ ?)\ \mathsf{in}\ (x_1, r, ?) \\ &\mathsf{in}\ (x_1 \oplus \{x = v\}, \mathsf{void}, ?)\end{aligned}$$

Comme on a annoté l'appel de fonction avec la post-condition de la fonction f, la seconde obligation de preuve est directement établie par q_f. Les deux obligations de preuve restantes sont alors :

$$\forall x_0.\ P(x_0) \Rightarrow P_f(x_0)[x \leftarrow a]$$

et

$$\begin{array}{l}\forall x_0.\ P(x_0) \Rightarrow P_f(x_0)[x \leftarrow a] \Rightarrow \\ \qquad \forall x_1.\ \forall v.\ Q_f(x_0, x_1, v)[x \leftarrow a] \Rightarrow Q(x_0, x_1 \oplus \{x = v\}, \mathsf{void})\end{array}$$

On doit donc montrer la pré-condition de la fonction f pour l'argument a, puis la post-condition Q sous l'hypothèse que l'on a la post-condition de la fonction f. En particulier, et c'est là un point très important, la preuve de l'établissement de Q_f est faite une fois pour toute à la définition de f et n'a pas besoin d'être refaite pour chaque appel de la fonction f.

On constate également que, contrairement au cas précédent, il n'y a plus substitution de la valeur affectée dans la post-condition Q, mais que valeur est abstraite sous la forme d'une variable v à propos de laquelle notre unique information est la preuve q_f de la proposition $Q_f(x_0, x_1, v)[x \leftarrow a]$. Une manière de le voir est de proposer une variante de la règle d'affectation de la logique de Floyd-Hoare dans le cas où l'expression affectée est annotée, qui serait la suivante :

$$\frac{\vdash \{P\}\ e\ \{r \mid Q\}}{\vdash \{P\}\ x := \{P\}\ e\ \{r \mid Q\}\ \{Q[r \leftarrow x]\}}(AFF')$$

En la combinant avec la règle de conséquence, elle permet alors de retrouver l'obligation de preuve ci-dessus.

Les deux exemples (2.11) et (2.12) montrent donc une grande différence entre les expressions qui ne sont pas annotées et celles qui le sont : dans le premier cas, les expressions peuvent être *substituées* dans le terme de preuve, donnant alors des obligations de preuve analogues à celles de la logique de Floyd-Hoare ; dans le deuxième cas, au contraire, les expressions sont *abstraites* et seules les annotations qui les accompagnaient interviennent alors dans les obligations de preuves. Nous reviendrons plus tard sur les intérêts d'une telle abstraction, notamment lorsque nous parlerons de modularité et de raffinement dans les perspectives qui seront évoquées page 152.

Boucles

Prenons maintenant le cas d'une boucle et montrons quelles sont les obligations de preuve qui lui sont rattachées. Plutôt que d'étudier le cas général, prenons un exemple concret de programme bâti sur une boucle et faisons-en la preuve de correction. Pour faire simple, nous avons choisi un programme calculant la somme des n premiers entiers. Un tel programme est donné Programme 1 page 88, où l'on suppose qu'une référence entière n est déclarée globalement. ($\overleftarrow{n_0}$ est une notation pour la valeur initiale de n, qui sera introduite dans la section 3.3.) Pour le spécifier, nous supposons que le terme $somme : Z \rightarrow Z$ a été introduit, avec par exemple l'axiomatisation suivante :

$$\begin{array}{ll} (1) & (somme\ 0) = 0 \\ (2) & \forall n : Z.\ n \geq 0 \Rightarrow (somme\ n) = (somme\ (n-1)) + n \end{array}$$

Programme 1 Boucle calculant la somme $\sum_{i=1}^{n} i$

```
1.   { n ≥ 0 }
2.  (let s = ref 0 in
3.   begin
4.     while !n > 0 do
5.       { invariant (somme ←n₀) = (somme n) + s ∧ 0 ≤ n variant n }
6.       s := !s + !n;
7.       n := !n − 1
8.     done;
9.     !s
10.  end)
11.  { result = (somme ←n) }
```

Appelons e ce programme annoté. Pour comprendre quelles sont les obligations de preuve qui résulteront de ce programme, observons son interprétation $\widehat{e}$ i.e. le terme de preuve incomplet que l'on construit à partir de e. Ce terme de preuve étant assez gros, observons-le en plusieurs étapes. À chaque fois on encadrera les obligations de preuve. Le terme $\widehat{e}$ s'écrit sous la forme :

$$\begin{array}{l} \widehat{e} = \lambda\{n = n_0\}.\ \lambda h : n_0 \geq 0. \\ \quad \mathsf{let}\ s_0 = 0\ \mathsf{in} \\ \quad \mathsf{let}\ (\{s = s_1; n = n_1\}, v, q_1) = \\ \quad\quad \mathsf{let}\ (\{s = s_1; n = n_1\}, \_, q_1) = (\widehat{e_1}\ \{s = s_0; n = n_0\}\ \boxed{1})\ \mathsf{in} \\ \quad\quad \mathsf{let}\ v = s_1\ \mathsf{in} \\ \quad\quad (\{s = s_1; n = n_1\}, v, \boxed{2}) \\ \quad \mathsf{in} \\ \quad (\{n = n_1\}, v, \boxed{2}) \end{array}$$

où l'on reconnaît la traduction de la construction let in, puis en dessous celle de la séquence. (Nous avons directement réduit la traduction de la déréférenciation $!s$ de la ligne 9 du programme.) L'interprétation de la boucle while est pour l'instant dénotée par $\widehat{e_1}$. Dès à présent, on peut observer deux obligations de preuves. La première correspond à l'application de $\widehat{e_1}$ i.e. à l'entrée de la boucle. On doit montrer l'invariant, pour les valeurs de n et s valant respectivement n_0 et s_0, sous l'hypothèse h i.e. $n_0 \geq 0$. Étant donné que le rédexe let $s_0 = 0$ in peut être réduit, on obtient finalement l'obligation suivante :

$$\frac{h \quad : \quad n_0 \geq 0}{(somme\ n_0) = (somme\ n_0) + 0 \wedge 0 \leq n_0} \tag{2.13}$$

La deuxième obligation de preuve provient de la post-condition du programme (ligne 11). Si elle apparaît dupliquée dans le terme $\widehat{e}$, c'est parce que cette post-condition est également celle de la séquence begin end des lignes 3 à 10. (Nous reviendrons sur ce point dans la section 2.4 où il sera question de propagations d'annotations.) Mais elle n'a besoin d'être prouvée qu'une seule fois, bien heureusement, car sa deuxième occurrence est immédiatement établie par q_1. Pour cette deuxième obligation, on doit montrer la post-condition portant sur les valeurs n_1 et s_1 des variables, sous l'hypothèse h initiale, qui n'est plus utile, et l'hypothèse q_1 de sortie de boucle, qui exprime que l'invariant de boucle est vérifié et que le test de la boucle n'est pas satisfait. Une fois le rédexe let $v = s_1$ in réduit, on obtient finalement l'obligation :

$$\frac{\begin{array}{lcl} h & : & n_0 \geq 0 \\ q_1 & : & ((\mathit{somme}\ n_0) = (\mathit{somme}\ n_1) + s_1 \wedge 0 \leq n_1) \wedge n_1 \leq 0 \end{array}}{s_1 = (\mathit{somme}\ n_0)} \tag{2.14}$$

Observons maintenant le corps de la boucle $\widehat{e_1}$. Afin d'alléger les notations, notons $I(s,n)$ l'invariant de boucle, c'est-à-dire

$$I(s,n) \quad \stackrel{\text{def}}{=} \quad (\mathit{somme}\ n_0) = (\mathit{somme}\ n) + s \wedge 0 \leq n$$

On a alors :

$$\begin{array}{ll}
\widehat{e_1} = & (\mathsf{WF}\ Z\ <_n\ \boxed{3} \\
& \quad \lambda\varphi.\lambda w.\lambda\{n = n_1; s = s_1\}.\lambda h_1 : I(n_1, s_1). \\
& \qquad \mathsf{let}\ (b, q_1) = \widehat{n_1 > 0}\ \mathsf{in} \\
& \qquad \mathsf{if}\ b\ \mathsf{then} \\
& \qquad\quad \mathsf{let}\ s_2 = s_1 + n_1\ \mathsf{in} \\
& \qquad\quad \mathsf{let}\ n_2 = n_1 - 1\ \mathsf{in} \\
& \qquad\quad (w\ n_2\ \boxed{4}\ \{n = n_2; s = s_2\}\ \boxed{5}) \\
& \qquad \mathsf{else} \\
& \qquad\quad (\{n = n_1; s = s_1\}, \mathsf{void}, \boxed{(h_1, q_1)}) \\
& \quad n_0)
\end{array}$$

Ce terme est l'application de l'opérateur de récursion bien fondée sur l'ordre usuel sur les entiers naturels, défini sur Z par $x <_n y \stackrel{\text{def}}{=} 0 \leq x < y$. L'obligation de preuve 3 exprime que l'ordre $<_n$ est bien fondé :

$$\frac{}{(\mathit{well\_founded}\ <_n)} \tag{2.15}$$

Examinons maintenant le corps de la boucle proprement dit. Il faut s'arrêter un instant sur le test $!n > 0$. Si dans le programme ce test est un booléen, il est plus pratique de manipuler des *propositions* lorsque l'on doit prouver les obligations de preuve. En l'occurrence, il est plus agréable de récupérer dans chaque branche du test les propositions $n_1 > 0$ et $n_1 \leq 0$, plutôt qu'un booléen exprimant cette dichotomie. C'est pourquoi le test $!n > 0$ est interprété par un terme $\widehat{n_1 > 0}$ retournant un booléen *et* la proposition qu'il représente. On lie alors les variables b et q_1 à ces deux objets. La section 3.4 donnera plus de détails sur l'interprétation des booléens dans les tests.

Considérons tout d'abord le cas où la boucle se termine, le booléen b valant false. Dans ce cas, on retourne un état inchangé, $\{n = n_1; s = s_1\}$, et on doit retourner une preuve de la post-condition de la boucle, à savoir $I(n_1, s_1) \wedge n_1 \leq 0$. Or on a justement une preuve h_1 de $I(n_1, s_1)$ et une preuve q_1 de $n_1 \leq 0$. On renvoie donc directement le terme de preuve (h_1, q_1), évitant ainsi d'engendrer une obligation de preuve inutile.

Dans l'autre cas, lorsque le booléen b vaut true, on exécute le corps de la boucle, ce qui est traduit par les deux constructions let in introduisant respectivement s_2 et n_2, puis on exécute l'appel récursif sur w. On doit passer à w quatre arguments :

- tout d'abord le variant, en l'occurrence n_2 ;
- puis une preuve que ce variant est strictement plus petit que le variant φ. C'est l'obligation de preuve 4. Il faut ici faire un commentaire important : tel qu'est écrit le terme $\widehat{e_1}$ ci-dessus, rien ne permet de savoir que le variant d'entrée φ vaut n_1. Dès lors, il n'est pas possible de montrer que $n_2 = n_1 - 1$ est strictement plus petit que φ pour $<_n$. Dans la pratique, on modifie donc légèrement l'argument φ pour lui adjoindre une preuve de $\varphi = \phi(x)$ où ϕ est l'expression du variant en fonction des variables du programme, en l'occurrence n. Dès lors, après réduction des rédexes let in, l'obligation 4 se trouve être de la forme :

$$\frac{\begin{array}{lcl} \varphi & : & Z \\ \pi & : & \varphi = n_1 \\ h_1 & : & I(n_1, s_1) \\ q_1 & : & n_1 > 0 \end{array}}{n_1 - 1 <_n \varphi} \tag{2.16}$$

  Dans la pratique, φ est automatiquement réécrite en $\phi(x)$ et disparaît du contexte ;
- le troisième argument de w est le nouvel état des variables, à savoir $\{n = n_2; s = s_2\}$, c'est-à-dire après réduction des rédexes let in l'état $\{n = n_1 - 1; s = s_1 + n_1\}$;
- enfin le dernier argument de w est une preuve de l'invariant i.e. de $I(n_2, s_2)$, soit après réduction une preuve de $I(n_1 - 1, s_1 + n_1)$. Cette obligation exprime la préservation de l'invariant au cours de l'exécution du corps de la boucle. L'obligation de preuve 5 est donc la suivante :

$$\frac{\begin{array}{lcl} h_1 & : & (\mathit{somme}\ n_0) = (\mathit{somme}\ n_1) + s_1 \wedge 0 \leq n_1 \\ q_1 & : & n_1 > 0 \end{array}}{(\mathit{somme}\ n_0) = (\mathit{somme}\ (n_1 - 1)) + (s_1 + n_1) \wedge 0 \leq (n_1 - 1)} \tag{2.17}$$

De manière générale, il y a donc quatre obligations de preuve attachées à une boucle while :

- le caractère bien fondé de l'ordre servant à justifier la terminaison (ici 2.15) ;
- la proposition exprimant que l'invariant de boucle est initialement vérifié (ici 2.13) ;
- la décroissance stricte du variant à l'issue du corps de la boucle (ici 2.16) ;
- la préservation de l'invariant de boucle par le corps de la boucle (ici 2.17).

Ce sont bien les mêmes que celles que l'on obtient avec la logique de Floyd-Hoare.

Appel de fonctions

Pour illustrer le cas des appels de fonctions, et en particulier des fonctions récursives, nous allons reprendre l'exemple précédent du calcul de la somme des n premiers entiers, en l'écrivant sous la forme d'une fonction récursive. Le programme est donné ci-dessous (Programme 2) et est toujours spécifié avec la même fonction *somme*.

Programme 2 Fonction récursive calculant la somme $\sum_{i=1}^{n} i$

```
1. let rec f_somme (n : Z) : Z { variant n } =
2.   { n ≥ 0 }
3.   (if n = 0 then
4.      0
5.    else
6.      n + (f_somme (n − 1)))
7.   { result = (somme n) }.
```

Appelons e ce programme et, comme précédemment, observons son interprétation $\widehat{e}$. Celle-ci est de la forme :

$$\widehat{e} = (\mathsf{WF}\ Z\ <_n\ \boxed{1}\ \lambda\varphi.\lambda f.\lambda n.\widehat{e_1})$$

où $\widehat{e_1}$ est l'interprétation du corps de la fonction (lignes 2 à 7). On a déjà une obligation de preuve, qui exprime le caractère bien fondé de l'ordre $<_n$:

$$\frac{}{(well\_founded\ <_n)} \tag{2.18}$$

Observons maintenant les obligations de preuve résultant des lignes 2 à 7 du programme i.e. du corps de la fonction. Le terme de preuve incomplet correspondant est le suivant :

$$\begin{array}{ll}
\widehat{e_1} = \lambda h : n \geq 0. & \mathsf{let}\ (b, q_1) = \widehat{n = 0}\ \mathsf{in} \\
& \mathsf{if}\ b\ \mathsf{then} \\
& \quad (0, \boxed{2}) \\
& \mathsf{else} \\
& \quad \mathsf{let}\ (a, q_2) = (f_somme\ (n-1)\ \boxed{3}\ (n-1)\ \boxed{4})\ \mathsf{in} \\
& \quad (n + a, \boxed{5})
\end{array}$$

où l'on reconnaît la traduction de la conditionnelle. (Nous avons volontairement réduit certains rédexes provenant de la traduction de l'application des opérations arithmétiques, par souci de clarté.)

La première obligation de preuve, notée 2, correspond au cas de base de la récurrence lorsque n vaut 0. On doit alors prouver la post-condition de f_somme dans laquelle *result* est substitué par 0, soit :

$$\frac{\begin{array}{lll} h & : & n \geq 0 \\ q_1 & : & n = 0 \end{array}}{0 = (somme\ n)} \tag{2.19}$$

Dans l'autre branche de la conditionnelle, où le test $n = 0$ est faux, on retourne le résultat $n + (f\_somme\ (n-1))$. Cette valeur résulte de l'application de deux fonctions, à savoir f_somme tout d'abord, puis $+$. L'application de la fonction récursive f_somme introduit deux obligations de preuve. La première concerne le variant : celui-ci vaut $n-1$ dans l'appel récursif et l'on doit montrer qu'il diminue strictement. Tout comme pour les boucles, la variable φ désignant le variant se voit adjoindre une preuve que celui-ci est égal à n. On obtient donc finalement l'obligation suivante :

$$\frac{\begin{array}{lcl} h & : & n \geq 0 \\ \varphi & : & Z \\ \pi & : & \varphi = n \\ q_1 & : & n \neq 0 \end{array}}{n - 1 <_n \varphi} \tag{2.20}$$

L'autre obligation de preuve rattachée à l'appel récursif concerne la pré-condition de f_somme, dont on doit montrer qu'elle est satisfaite sur les arguments de l'appel récursif, en l'occurrence $n-1$. D'où l'obligation :

$$\frac{\begin{array}{lcl} h & : & n \geq 0 \\ q_1 & : & n \neq 0 \end{array}}{n - 1 \geq 0} \tag{2.21}$$

Enfin la dernière obligation de preuve consiste à montrer que la post-condition de f_somme est satisfaite à l'issue de la deuxième branche de la conditionnelle i.e. lorsque le résultat est $n + a$. L'obligation est alors la suivante :

$$\frac{\begin{array}{lcl} h & : & n \geq 0 \\ q_1 & : & n \neq 0 \\ a & : & Z \\ q_2 & : & a = (somme\ (n-1)) \end{array}}{n + a = (somme\ n)} \tag{2.22}$$

On constate que l'on peut bien prouver cette obligation, car l'on a récupéré de l'appel récursif non seulement une valeur a mais également une preuve q_2 que $a = (somme\ (n-1))$.

On constate sur cet exemple que la spécification d'une fonction récursive est plus directe que celle d'une boucle impérative réalisant le même calcul et que, de même, les obligations de preuve sont plus simple. Il s'avère que c'est souvent le cas. De manière générale, les obligations de preuve liées à une fonction récursive sont les suivantes :

- le caractère bien fondé de l'ordre justifiant la terminaison de la fonction (ici 2.18) ;
- la (ou les) obligation(s) de preuve visant à montrer la post-condition de la fonction (ici il y en a deux, 2.19 et 2.22) ;
- et pour chaque appel récursif :
  - la décroissance stricte du variant (ici 2.20),
  - l'établissement de la pré-condition de la fonction (ici 2.21).

Appel par variable

Pour compléter cette section, terminons par un exemple d'appel par variable. Pour cela, spécifions une fonction *reverse* impérative qui retourne une liste passée en argument comme une référence sur une valeur du type *list*, qui n'a pas besoin d'être précisé ici. C'est se donner une fonction ayant le type suivant :

$$\mathit{reverse} \quad : \quad (x : \mathit{list}\ \mathsf{ref}) \rightarrow (\mathsf{unit}, (\{x\}, \{x\}), \mathsf{True}, x = (\mathit{rev}\ \overleftarrow{x}))$$

où *rev* est une l'opération fonctionnelle consistant à retourner les éléments d'une liste, qui peut être définie explicitement ou seulement axiomatisée. Utilisons alors la fonction *reverse* ci-dessus dans un programme supprimant le dernier élément d'une liste d'une façon assez stupide. Ce programme est donné ci-dessous (Programme 3).

Programme 3 Fonction supprimant en place le dernier élément d'une liste

1. $\mathsf{fun}\ (l : \mathit{list}\ \mathsf{ref}) \rightarrow$
2. $\quad\mathsf{begin}$
3. $\qquad(\mathit{reverse}\ l);$
4. $\qquad l := (\mathit{tl}\ !l);$
5. $\qquad(\mathit{reverse}\ l)$
6. $\quad\mathsf{end}$
7. $\quad\{\ (\mathit{rev}\ l) = (\mathit{tl}\ (\mathit{rev}\ \overleftarrow{l}))\ \}$

Appelons e ce programme, et observons son interprétation comme terme de preuve incomplet $\widehat{e}$. On a :

$$\begin{array}{rcl} \widehat{e} & = & \lambda\{l = l_0\}.\ \mathsf{let}\ (\{l = l_1\}, \_, q_1) = (\widehat{\mathit{reverse}}\ l\ \{l = l_0\})\ \mathsf{in} \\ & & \phantom{\lambda\{l = l_0\}.\ }\mathsf{let}\ (\{l = l_2\}, \_) = \mathsf{let}\ v = (\mathit{tl}\ l_1)\ \mathsf{in}\ (\{l = v\}, \mathsf{void})\ \mathsf{in} \\ & & \phantom{\lambda\{l = l_0\}.\ }\mathsf{let}\ (\{l = l_3\}, v, q_2) = (\widehat{\mathit{reverse}}\ l\ \{l = l_2\})\ \mathsf{in} \\ & & \phantom{\lambda\{l = l_0\}.\ }(\{l = l_3\}, v, \boxed{1}) \end{array}$$

Pour comprendre comment l'appel par nom a été traduit, il faut observer le type de l'interprétation de la fonction *reverse*. En effet, cette fonction que nous avons spécifiée donne lieu dans le Calcul des Constructions Inductives à une variable $\widehat{\mathit{reverse}}$ de type :

$$\widehat{\mathit{reverse}} \quad : \quad \forall x.\ \forall s : \{x : \mathit{list}\}.\ \{(s', v) : \{x : \mathit{list}\} \times \mathsf{unit} \mid s'.x = (\mathit{rev}\ s.x)\}$$

Dès lors, les deux applications de la fonction *reverse* lignes 3 et 5 dans le programme e donnent lieu dans $\widehat{e}$ à deux applications de $\widehat{\mathit{reverse}}$. À chaque fois, deux arguments sont passés, le premier étant le champ désignant l'argument réel, à savoir l, et le second un enregistrement donnant la valeur courante de l, et le résultat est un enregistrement donnant la valeur de l à l'issue de l'appel de la fonction. Cet enregistrement mentionne le champ l, car l'application au premier argument, soit $(\widehat{\mathit{reverse}}\ l)$, a substitué l'argument réel l à l'argument formel x.

En fin de compte, une fois le rédexe de la deuxième ligne de $\widehat{e}$ réduit, on se retrouve avec l'obligation de preuve suivante :

$$\begin{array}{lcl} l_0 & : & \mathit{list} \\ l_1 & : & \mathit{list} \\ q_1 & : & l_1 = (\mathit{rev}\ l_0) \\ l_3 & : & \mathit{list} \\ q_2 & : & l_3 = (\mathit{rev}\ (\mathit{tl}\ l_1)) \\ \hline \multicolumn{3}{c}{(\mathit{rev}\ l_3) = (\mathit{tl}\ (\mathit{rev}\ l_0))} \end{array}$$

On constate que l'on a bien récupéré la post-condition de la fonction *reverse* à chaque fois, par l'intermédiaire de q_1 et q_2, et que ces propositions portent bien sur les valeurs successives de l. L'appel par nom a donc bien été traduit.

2.4 Complétude

Nous avons déjà évoqué la notion de complétude dans la section 2.1. En substance, elle exprime que « Si le programme vérifie la spécification alors je peux en faire la preuve dans mon système. » Avec les notations déjà introduites pour la logique de Hoare, cela s'écrit formellement de la façon suivante :

$$\models \{P\}\ e\ \{Q\} \quad \Longrightarrow \quad \vdash \{P\}\ e\ \{Q\}$$

Dans notre système, la complétude s'exprimerait par une proposition de la forme :

$$[\![e]\!]_\kappa \quad \Longrightarrow \quad \widehat{e} \text{ peut être complétée en une preuve de } \widehat{\kappa}$$

et constituerait donc la réciproque du théorème de correction (Théorème 4 page 85).

Essayons d'illustrer de façon simple les problèmes liés à la complétude. Prenons par fragment le fragment de programme suivant :

```
1. begin
2.    m := (min x y);
3.    m := !m - 1
4. end
5. { m < x }
```

Sa traduction donne lieu à un terme de la forme

$$\begin{array}{l} \mathsf{let}\ (\{m = m_1\}, \_) = (\{m = (\mathit{min}\ x\ y)\}, \mathsf{void})\ \mathsf{in} \\ \mathsf{let}\ (\{m = m_2\}, v) = (\{m = m_1 - 1\}, \mathsf{void})\ \mathsf{in} \\ (\{m = m_2\}, v, ?) \end{array}$$

ce qui donne après réductions une obligation de preuve de la forme $(\mathit{min}\ x\ y) - 1 < x$, parfaitement prouvable. Et cela grâce aux *substitutions* amenées par les réductions. Mais supposons maintenant que l'on modifie la ligne 2 du code ci-dessus de la façon suivante :

$$
\begin{array}{rl}
1. & \mathsf{begin} \\
2. & \quad \mathsf{if}\ x < y\ \mathsf{then}\ m := x\ \mathsf{else}\ m := y; \\
3. & \quad m := !m - 1 \\
4. & \mathsf{end} \\
5. & \{\ m < x\ \}
\end{array}
$$

Alors sa traduction sera de la forme :

$$
\begin{array}{ll}
\mathsf{let}\ (\{m = m_1\}, \_) = & \mathsf{let}\ (b, q) = x \widehat{<} y\ \mathsf{in} \\
& \mathsf{if}\ b\ \mathsf{then}\ (\{m = x\}, \mathsf{void})\ \mathsf{else}\ (\{m = y\}, \mathsf{void})\ \ \mathsf{in} \\
\multicolumn{2}{l}{\mathsf{let}\ (\{m = m_2\}, v) = (\{m = m_1 - 1\}, \mathsf{void})\ \mathsf{in}} \\
\multicolumn{2}{l}{(\{m = m_2\}, v, ?)}
\end{array}
$$

et cette fois seul le deuxième let in sera réduit — le premier n'est plus un rédexe — ce qui donnera une obligation de preuve de la forme :

$$
\frac{m_1 \quad : \quad Z}{m_1 - 1 < x}
$$

que l'on ne peut pas prouver car l'on ne dispose d'aucune information sur la variable m_1, qui représente ici la valeur de m après la première affectation. Ici il n'y a pas eu substitution de la valeur de m_1 mais *abstraction* de celle-ci. Nous avons illustré ces deux alternatives au début de la section 2.3.4, dans le paragraphe consacré aux affectations.

Comme la valeur m_1 sera toujours abstraite dans le programme ci-dessus, la seule manière d'aboutir à une obligation de preuve que l'on puisse prouver est de récupérer avec m_1 une propriété portant sur cette variable. La solution consiste donc à annoter la conditionnelle avec une post-condition. On peut, par exemple, modifier la ligne 2 de la façon suivante :

$$
\begin{array}{rl}
1. & \mathsf{begin} \\
2. & \quad (\mathsf{if}\ x < y\ \mathsf{then}\ m := x\ \mathsf{else}\ m := y)\ \{\ m \leq x\ \}; \\
3. & \quad m := !m - 1 \\
4. & \mathsf{end} \\
5. & \{\ m < x\ \}
\end{array}
$$

Dès lors le terme de preuve devient le suivant :

$$
\begin{array}{ll}
\mathsf{let}\ (\{m = m_1\}, \_, q_1) = & \mathsf{let}\ (b, q) = x \widehat{<} y\ \mathsf{in} \\
& \mathsf{if}\ b\ \mathsf{then}\ (\{m = x\}, \mathsf{void}, ?)\ \mathsf{else}\ (\{m = y\}, \mathsf{void}, ?)\ \ \mathsf{in} \\
\multicolumn{2}{l}{\mathsf{let}\ (\{m = m_2\}, v) = (\{m = m_1 - 1\}, \mathsf{void})\ \mathsf{in}} \\
\multicolumn{2}{l}{(\{m = m_2\}, v, ?)}
\end{array}
$$

et on obtiendra, outre deux obligations de preuve visant à établir que $m \leq x$ à l'issue de chaque branche de la conditionnelle, l'obligation maintenant parfaitement prouvable suivante :

$$
\frac{\begin{array}{lcl} m_1 & : & Z \\ q_1 & : & m_1 \leq x \end{array}}{m_1 - 1 < x}
$$

L'annotation $m \leq x$ que nous avons rajoutée peut paraître *ad-hoc*. Il y avait en fait une annotation plus générale, qui elle ne l'est pas, appelée *plus faible pré-condition* et calculée à partir de la post-condition $m < x$ et de l'affectation $m := !m - 1$. Il s'agissait en l'occurence de $m - 1 < x$.

De manière plus générale, on peut montrer que, pour un certain fragment de notre langage, il existe une manière canonique d'annoter les programmes qui conduit à des obligations de preuve prouvables dès lors que le programme vérifie la spécification. Pour établir ce résultat de complétude, il nous faut commencer par définir formellement la notion de plus faible pré-condition.

2.4.1 Plus faible pré-condition

Afin d'alléger les définitions et preuves à venir, nous allons introduire quelques simplifications, sans réelle perte de généralité. D'une part nous allons considérer par la suite que les post-conditions ne portent plus que sur les valeurs des variables à l'issue de l'évaluation — comme c'est le cas dans les présentations traditionnelles de la logique de Floyd-Hoare. Dès lors, la notion de variable auxiliaires devient nécessaire à la spécification des programmes. Cependant, le résultat de complétude que nous allons établir dans cette section et la suivante pourrait être généralisé au cas des post-conditions précédemment introduites, et c'est pourquoi il n'est pas réellement nécessaire d'introduire formellement la notion de variables auxiliaires.

La deuxième simplification porte sur les fonctions contenues dans les références et retournées comme résultat d'un programme. Nous allons en effet supposer par la suite que les types de ces fonctions ne contiennent pas d'annotation logique. Ainsi, une telle fonction pourra avoir le type $\mathsf{int} \rightarrow \mathsf{int}$ mais pas un type de la forme $(x : \mathsf{int}) \rightarrow (y : \mathsf{int}, \bot, P, Q)$. Dès lors, la définition de la réalisabilité impérative (Définition 39 page 80) est simplifiée et peut s'écrire simplement comme

$$e \ \mathsf{r}_i \ (r : \tau, (\rho, \omega), P, Q) \quad \stackrel{\text{def}}{=} \quad \forall x. P(x) \Rightarrow \mathsf{let}\ (y, r) = (\overline{e}\ x)\ \mathsf{in}\ Q(y, r)$$

Sous ces hypothèses, nous pouvons maintenant introduire la notion de plus faible pré-condition d'un programme e et d'une post-condition Q, notée $\mathrm{wp}(e)(Q)$. Historiquement, elle est dûe à E. W. Dijkstra [11], qui la définit comme la plus petite pré-condition P telle que $\{P\}\ e\ \{Q\}$, et dont il montre ensuite un certain nombre de propriétés. Une autre manière de procéder consiste à commencer par définir le prédicat $\mathrm{wp}(e)(Q)$ par récurrence sur e, puis à montrer qu'il s'agit de la plus faible pré-condition P entraînant $\{P\}\ e\ \{Q\}$. C'est cette deuxième formalisation que nous avons choisie ici.

De manière générale, il n'est pas possible de déterminer $\mathrm{wp}(e)(Q)$ lorsque e est une boucle, car cela nécessiterait de déterminer l'invariant de boucle, qui est le plus souvent une généralisation de Q. Nous considérons donc ici uniquement le fragment de notre langage qui ne contient pas de boucle. Nous excluons également pour l'instant les appels de fonctions, dont nous reparlerons à la fin de cette section.

Définition 40 (plus faible pré-condition) *Soit Γ un environnement bien formé et e un programme bien typé dans Γ de type τ. Soit $\lambda r : \tau.Q$ une post-condition bien formée pour e.*

Alors la pré-condition $\mathrm{wp}(e)(\lambda r.Q)$*, appelé* plus faible pré-condition de e et de Q*, est définie récursivement sur la structure de* e *par les égalités données figure 2.4.*

$$
\begin{array}{lcl}
\mathrm{wp}(c)(\lambda r.Q) & \stackrel{\mathrm{def}}{=} & Q[r \leftarrow c] \\
\mathrm{wp}(x)(\lambda r.Q) & \stackrel{\mathrm{def}}{=} & Q[r \leftarrow x] \\
\mathrm{wp}(!x)(\lambda r.Q) & \stackrel{\mathrm{def}}{=} & Q[r \leftarrow x] \\
\mathrm{wp}(e_1 \ ; \ e_2)(\lambda r.Q) & \stackrel{\mathrm{def}}{=} & \mathrm{wp}(e_1)(\lambda\_.\mathrm{wp}(e_2)(\lambda r.Q)) \\
\mathrm{wp}(x \ := \ e_1)(\lambda r.Q) & \stackrel{\mathrm{def}}{=} & \mathrm{wp}(e_1)(\lambda x.Q[r \leftarrow \mathsf{void}]) \\
\mathrm{wp}(\mathsf{ref}\ e_1)(\lambda r.Q) & \stackrel{\mathrm{def}}{=} & \mathrm{wp}(e_1)(\lambda r.Q) \\
\mathrm{wp}(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2)(\lambda r.Q) & \stackrel{\mathrm{def}}{=} & \mathrm{wp}(e_1)(\lambda x.\mathrm{wp}(e_2)(\lambda r.Q)) \\
\mathrm{wp}(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3)(\lambda r.Q) & \stackrel{\mathrm{def}}{=} & \mathrm{wp}(e_1)(\lambda r. \quad r = \mathsf{true} \Rightarrow \mathrm{wp}(e_2)(\lambda r.Q) \\
& & \qquad\qquad \wedge \quad r = \mathsf{false} \Rightarrow \mathrm{wp}(e_3)(\lambda r.Q))
\end{array}
$$

FIG. 2.4: Définition de la plus faible pré-condition

Remarque. Si E désigne une expression purement fonctionnelle du premier ordre, telle que $!x+1$, c'est-à-dire une expression formée de constantes, de variables, d'accès à des références et d'application de constantes, alors on constate que

$$\mathrm{wp}(E)(\lambda r.Q) \quad = \quad Q[r \leftarrow E[!x \leftarrow x]]$$

Dès lors, lorsque nous ferons par la suite une preuve par récurrence sur la structure d'un programme, nous traiterons directement du cas d'une telle expression, regroupant ainsi plusieurs cas quasi identiques.

Le lemme suivant justifie la définition du prédicat $\mathrm{wp}(e)(\lambda r.Q)$ en montrant que sous l'hypothèse que la plus faible pré-condition est vérifiée alors la post-condition est établie par le programme.

Lemme 11 *Soit* Γ *un environnement bien formé et* e *un programme tel que* $\Gamma \vdash_r e : (\tau, \epsilon)$*. Alors le programme* e *vérifie la spécification* $(r : \tau, \epsilon, \mathrm{wp}(e)(\lambda r.Q), Q)$*, c'est-à-dire*

$$[\![e]\!]_{(r:\tau,\epsilon,\mathrm{wp}(e)(\lambda r.Q),Q)}$$

PREUVE. En vertu du Lemme 9 page 81, cette proposition peut également s'écrire

$$\overline{e}\ \mathsf{r}_i\ (r : \tau, \epsilon, \mathrm{wp}(e)(\lambda r.Q), Q)$$

c'est-à-dire

$$\forall x.\ \mathrm{wp}(e)(\lambda r.Q)(x) \Rightarrow \mathsf{let}\ (y, r) = (\overline{e}\ x)\ \mathsf{in}\ Q(y, r)$$

Montrons cette dernière proposition par récurrence sur la structure de e :

$e \equiv E$ **:** Par définition on a $\mathrm{wp}(e)(\lambda r.Q) = Q[r \leftarrow E[!z \leftarrow z]]$ et $(\overline{e}\ x) = (\{\}, E[!z \leftarrow x.z])$. Le résultat est donc immédiat.

$e \equiv e_1 \ ; \ e_2$ **:** Soit x_0 tel que $\mathrm{wp}(e)(\lambda r.Q)(x_0)$ et soit $(y,r) = (\overline{e}\ x_0)$. On a d'une part

$$\mathrm{wp}(e)(\lambda r.Q) = \mathrm{wp}(e_1)(\lambda\_.\mathrm{wp}(e_2)(\lambda r.Q))$$

et d'autre part

$$\begin{array}{rl} (\overline{e}\ x_0) = & \mathsf{let}\ (x_1,\_) = (\overline{e_1}\ x_0)\ \mathsf{in} \\ & \mathsf{let}\ (x_2,v) = (\overline{e_2}\ (x_0 \oplus x_1))\ \mathsf{in} \\ & (x_1 \oplus x_2, v) \end{array}$$

Soit $(x_1,\_) = (\overline{e_1}\ x_0)$. Par hypothèse de récurrence appliquée à e_1, à l'état x_0 et au prédicat $\lambda\_.\mathrm{wp}(e_2)(\lambda r.Q)$ on a

$$\mathrm{wp}(e_2)(\lambda r.Q)(x_1)$$

Soit alors $(x_2,v) = (\overline{e_2}\ (x_0 \oplus x_1))$. Par hypothèse de récurrence appliquée à e_2, à l'état $x_0 \oplus x_1$ et au prédicat $\lambda r.Q$ on a

$$Q(x_0 \oplus x_1 \oplus x_2, v)$$

c'est-à-dire $Q(y,v)$.

$e \equiv x\ :=\ e_1$ **:** Soit x_0 tel que $\mathrm{wp}(e)(\lambda r.Q)(x_0)$ et soit $(y,r) = (\overline{e}\ x_0)$. On a d'une part

$$\mathrm{wp}(x\ :=\ e_1)(\lambda r.Q) = \mathrm{wp}(e_1)(\lambda x.Q[r \leftarrow \mathsf{void}])$$

et d'autre part

$$(\overline{e}\ x_0) = \mathsf{let}\ (x_1,v) = (\overline{e_1}\ x_0)\ \mathsf{in}\ (x_1 \oplus \{x = v\}, \mathsf{void})$$

Soit $(x_1,v) = (\overline{e_1}\ x_0)$. Par hypothèse de récurrence appliquée à e_1, à l'état x_0 et au prédicat $\lambda x.Q[r \leftarrow \mathsf{void}]$ on a

$$\begin{array}{rl} & Q[r \leftarrow \mathsf{void}](x_1,v) \\ = & Q(x_1 \oplus \{x = v\}, \mathsf{void}) \\ = & Q(y,r) \end{array}$$

$e \equiv \mathsf{ref}\ e_1$ **:** La récurrence est immédiate car $\overline{e} = \overline{e_1}$ et $\mathrm{wp}(e)(\lambda r.Q) = \mathrm{wp}(e_1)(\lambda r.Q)$.

$e \equiv \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ **:** Soit x_0 tel que $\mathrm{wp}(e)(\lambda r.Q)(x_0)$ et soit $(y,r) = (\overline{e}\ x_0)$. On a

$$\mathrm{wp}(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2)(\lambda r.Q) = \mathrm{wp}(e_1)(\lambda x.\mathrm{wp}(e_2)(\lambda r.Q))$$

Distinguons deux cas suivant que e_1 est ou non une référence :

- si e_1 n'est pas une référence, alors on a

$$\begin{array}{rl} (\overline{e}\ x_0) = & \mathsf{let}\ (x_1,x) = (\overline{e_1}\ x_0)\ \mathsf{in} \\ & \mathsf{let}\ (x_2,v) = (\overline{e_2}\ (x_0 \oplus x_1))\ \mathsf{in} \\ & (x_1 \oplus x_2, v) \end{array}$$

Soit $(x_1, x) = (\overline{e_1}\ x_0)$. Par hypothèse de récurrence appliquée à e_1, à l'état x_0 et au prédicat $\lambda x.\mathrm{wp}(e_2)(\lambda r.Q)$ on a

$$\mathrm{wp}(e_2)(\lambda r.Q)(x_1)$$

Soit alors $(x_2, v) = (\overline{e_2}\ (x_0 \oplus x_1))$. Par hypothèse de récurrence appliquée à e_2, à l'état $x_0 \oplus x_1$ et au prédicat $\lambda r.Q$ on a

$$Q(x_0 \oplus x_1, v)$$

c'est-à-dire $Q(y, r)$.

– si e_1 est une référence, alors on a

$$\begin{array}{rl} (\overline{e}\ x_0) = & \mathsf{let}\ (x_1, v_0) = (\overline{e_1}\ x_0)\ \mathsf{in} \\ & \mathsf{let}\ (x_2, v) = (\overline{e_2}\ (x_0 \oplus x_1 \oplus \{x = v_0\}))\ \mathsf{in} \\ & (x_1 \oplus x_2 \backslash x, v) \end{array}$$

Soit $(x_1, v_0) = (\overline{e_1}\ x_0)$. Par hypothèse de récurrence appliquée à e_1, à l'état x_0 et au prédicat $\lambda x.\mathrm{wp}(e_2)(\lambda r.Q)$ on a

$$\begin{array}{rl} & \mathrm{wp}(e_2)(\lambda r.Q)(x_1)[x \leftarrow v_0] \\ = & \mathrm{wp}(e_2)(\lambda r.Q)(x_1 \oplus \{x = v_0\}) \end{array}$$

Soit alors $(x_2, v) = (\overline{e_2}\ (x_0 \oplus x_1 \oplus \{x = v_0\}))$. Par hypothèse de récurrence appliquée à e_2, à l'état $x_0 \oplus x_1 \oplus \{x = v_0\}$ et au prédicat $\lambda r.Q$ on a

$$\begin{array}{rll} & Q(x_1 \oplus \{x = v_0\} \oplus x_2, v) & \\ = & Q(x_1 \oplus x_2 \backslash x, v) & \text{car } x \notin Q \\ = & Q(y, r) & \end{array}$$

$e \equiv \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$ **:** Soit x_0 tel que $\mathrm{wp}(e)(\lambda r.Q)(x_0)$ et soit $(y, r) = (\overline{e}\ x_0)$. On a d'une part

$$\begin{array}{rl} \mathrm{wp}(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3)(\lambda r.Q) = \mathrm{wp}(e_1)(\lambda r. & r = \mathsf{true} \Rightarrow \mathrm{wp}(e_2)(\lambda r.Q) \\ \wedge & r = \mathsf{false} \Rightarrow \mathrm{wp}(e_3)(\lambda r.Q)) \end{array}$$

et d'autre part

$$\begin{array}{rl} (\overline{e}\ x_0) = & \mathsf{let}\ (x_1, b) = (\overline{e_1}\ x_0)\ \mathsf{in} \\ & \mathsf{if}\ b\ \mathsf{then} \\ & \quad \mathsf{let}\ (x_2, v) = (\overline{e_2}\ (x_0 \oplus x_1))\ \mathsf{in}\ (x_0 \oplus x_1 \oplus x_2, v) \\ & \mathsf{else} \\ & \quad \mathsf{let}\ (x_2, v) = (\overline{e_3}\ (x_0 \oplus x_1))\ \mathsf{in}\ (x_0 \oplus x_1 \oplus x_2, v) \end{array}$$

Soit $(x_1, b) = (\overline{e_1}\ x_0)$. Par hypothèse de récurrence appliquée à e_1, à l'état x_0 et au prédicat $\lambda r.r = \mathsf{true} \Rightarrow \mathrm{wp}(e_2)(\lambda r.Q) \wedge r = \mathsf{false} \Rightarrow \mathrm{wp}(e_3)(\lambda r.Q)$ on a

$$b = \mathsf{true} \Rightarrow \mathrm{wp}(e_2)(\lambda r.Q) \wedge b = \mathsf{false} \Rightarrow \mathrm{wp}(e_3)(\lambda r.Q)$$

Supposons que $b = \mathsf{true}$. On a alors $\mathrm{wp}(e_2)(\lambda r.Q)(x_0 \oplus x_1)$. Soit $(x_2, v) = (\overline{e_2}\ (x_0 \oplus x_1))$. Par hypothèse de récurrence appliquée à e_2, à l'état $x_0 \oplus x_1$ et au prédicat $\lambda r.Q$ on a

$$\begin{array}{rl} & Q(x_0 \oplus x_1 \oplus x_2, v) \\ = & Q(y, r) \end{array}$$

Idem si $b = \mathsf{false}$.

□

De ce lemme on déduit immédiatement le corollaire suivant :

Corollaire 2 *Soit Γ un environnement bien formé et e un programme tel que $\Gamma \vdash_r e : (\tau, \epsilon)$. Alors pour toutes pré- et post-conditions P et Q, on a*

$$(P \Rightarrow \mathrm{wp}(e)(\lambda r.Q)) \quad \Rightarrow \quad [\![e]\!]_{(r:\tau,\epsilon,P,Q)}$$

Appels de fonctions. Il est possible d'exprimer la plus faible pré-condition dans le cas d'un appel de fonction. Donnons-nous une fonction f, dont la spécification est

$$f \quad : \quad \mathsf{fun}\ (\vec{x_i} : \vec{\tau_i}) \rightarrow (v : \tau, \epsilon, P_f, Q_f)$$

Alors la plus faible pré-condition du programme $e \equiv (f\ e_1\ \ldots\ e_n)$ et de la post-condition $\lambda r.Q$ doit être la plus faible pré-condition impliquant P_f et telle que Q_f implique Q. En prenant en compte rigoureusement les expressions e_i, on obtient la définition suivante :

$$\begin{array}{rcl} \mathrm{wp}((f\ e_1\ \ldots\ e_n))(\lambda r.Q) & \stackrel{\text{def}}{=} & \mathrm{wp}(e_1)(\lambda x_1. \\ & & \quad \mathrm{wp}(e_2)(\lambda x_2. \\ & & \qquad \ddots \\ & & \qquad\quad \mathrm{wp}(e_n)(\lambda x_n. \\ & & \qquad\qquad P_f \wedge \forall \vec{y}. \forall v. Q_f(\vec{y}) \Rightarrow Q[r \leftarrow v](\vec{y})\ ) \ldots)) \end{array}$$

On remarque que la quantification universelle sur les variables $\vec{y}$ et v introduit une *abstraction* de la sortie et du résultat de l'appel, de façon similaire aux obligations de preuve obtenues dans ce cas, comme nous l'avons montré précédemment section 2.3.4 page 85.

Cependant, l'appel de fonction ne constitue pas une construction pour laquelle il soit nécessaire d'effectuer en pratique un calcul de plus faible pré-condition. De plus, si la fonction f n'a pas été spécifiée de manière assez générale, il se peut alors que la plus faible-précondition définie ci-dessus ne conduise pas à des obligations de preuve prouvables — dans le cas où l'on suppose la correction du programme. Dès lors, il nous faudrait faire des hypothèses sur la spécification des fonctions pour établir le résultat de complétude, et nous avons préféré ne pas alourdir inutilement les preuves en excluant les appels de fonctions dans ce qui suit.

2.4.2 Résultat de complétude

Nous allons maintenant établir le résultat de complétude. Pour cela, nous commençons par montrer le lemme suivant, qui exprime que la plus faible pré-condition est satisfaite dès lors que le programme établit la post-condition.

Lemme 12 *Soit* Γ *un environnement bien formé et* e *un programme tel que* $\Gamma \vdash_r e : (\tau, \epsilon)$. *Alors*

$$\forall x.\ (\mathsf{let}\ (y,r) = (\overline{e}\ x)\ \mathsf{in}\ Q(y,r)) \Rightarrow \mathrm{wp}(e)(\lambda r.Q)(x)$$

PREUVE. Montrons cette proposition par récurrence sur la structure de e :

$e \equiv E$ **:** Le résultat est immédiat car on a $\mathrm{wp}(e)(\lambda r.Q) = Q[r \leftarrow E[!z \leftarrow z]]$ et $(\overline{e}\ x) = (\{\}, E[!z \leftarrow x.z])$.

$e \equiv e_1\ ;\ e_2$ **:** Soit x_0 et $(y,r) = (\overline{e}\ x_0)$. On suppose $Q(y,r)$. Soit alors $(x_1, \_) = (\overline{e_1}\ x_0)$ et $(x_2, v) = (\overline{e_2}\ (x_0 \oplus x_1))$. Par hypothèse de récurrence appliquée à e_2, à l'état $x_0 \oplus x_1$ et au prédicat $\lambda r.Q$, on a

$$\mathrm{wp}(e_2)(\lambda r.Q)(x_0 \oplus x_1)$$

et par hypothèse de récurrence appliquée à e_1, à l'état x_0 et au prédicat $\lambda\_.\mathrm{wp}(e_2)(\lambda r.Q)$ on a

$$\begin{array}{rl} & \mathrm{wp}(e_2)(\lambda\_.\mathrm{wp}(e_2)(\lambda r.Q))(x_0) \\ = & \mathrm{wp}(e)(\lambda r.Q)(x_0) \end{array}$$

$e \equiv x := e_1$ **:** Soit x_0 et $(y,r) = (\overline{e}\ x_0)$. On suppose $Q(y,r)$. Soit alors $(x_1, v) = (\overline{e_1}\ x_0)$. On a $Q(y,r)$ i.e. $Q(x_1 \oplus \{x = v\})$ i.e. $(\lambda x.Q[r \leftarrow \mathsf{void}])(x_1, v)$. Dès lors par hypothèse de récurrence appliquée à e_1, à l'état x_0 et au prédicat $\lambda x.Q[r \leftarrow \mathsf{void}]$ on a

$$\begin{array}{rl} & \mathrm{wp}(e_1)(\lambda x.Q[r \leftarrow \mathsf{void}])(x_0) \\ = & \mathrm{wp}(e)(\lambda r.Q)(x_0) \end{array}$$

$e \equiv \mathsf{ref}\ e_1$ **:** La récurrence est immédiate car $\overline{e} = \overline{e_1}$ et $\mathrm{wp}(e)(\lambda r.Q) = \mathrm{wp}(e_1)(\lambda r.Q)$.

$e \equiv \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ **:** Soit x_0 et $(y,r) = (\overline{e}\ x_0)$. On suppose $Q(y,r)$. On distingue deux cas suivant que e_1 est ou non une référence :

- si e_1 n'est pas une référence, alors soit $(x_1, x) = (\overline{e_1}\ x_0)$ et $(x_2, v) = (\overline{e_2}\ (x_0 \oplus x_1))$. Par hypothèse de récurrence appliquée à e_2, à l'état $x_0 \oplus x_1$ et au prédicat $\lambda r.Q$ on a

  $$\mathrm{wp}(e_2)(\lambda r.Q)(x_0 \oplus x_1)$$

  Dès lors par hypothèse de récurrence appliquée à e_1, à l'état x_0 et au prédicat $\lambda x.\mathrm{wp}(e_2)(\lambda r.Q)$ on a

  $$\begin{array}{rl} & \mathrm{wp}(e_1)(\lambda x.\mathrm{wp}(e_2)(\lambda r.Q))(x_0) \\ = & \mathrm{wp}(e)(\lambda r.Q)(x_0) \end{array}$$

- si e_1 est une référence, alors soit $(x_1, v_0) = (\overline{e_1}\ x_0)$ et $(x_2, v) = (\overline{e_2}\ (x_0 \oplus x_1 \oplus \{x = v_0\}))$. Par hypothèse de récurrence appliquée à e_2, à l'état $x_0 \oplus x_1 \oplus \{x = v_0\}$ et au prédicat $\lambda r.Q$ on a

  $$\begin{array}{rl} & \mathrm{wp}(e_2)(\lambda r.Q)(x_0 \oplus x_1 \oplus \{x = v_0\}) \\ = & (\lambda x.\mathrm{wp}(e_2)(\lambda r.Q))(x_0 \oplus x_1, v_0) \end{array}$$

  Dès lors par hypothèse de récurrence appliquée à e_1, à l'état x_0 et au prédicat $\lambda x.\mathrm{wp}(e_2)(\lambda r.Q)$ on a

  $$\begin{array}{rl} & \mathrm{wp}(e_1)(\lambda x.\mathrm{wp}(e_2)(\lambda r.Q))(x_0) \\ = & \mathrm{wp}(e)(\lambda r.Q)(x_0) \end{array}$$

$e \equiv \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$ **:** Soit x_0 et $(y, r) = (\overline{e}\ x_0)$. On suppose $Q(y, r)$. Soit $(x_1, b) = (\overline{e_1}\ x_0$. Supposons que $b = \mathsf{true}$. Soit alors $(x_2, v) = (\overline{e_2}\ (x_0 \oplus x_1))$. Par hypothèse de récurrence appliquée à e_2, à l'état $x_0 \oplus x_1$ et au prédicat $\lambda r.Q$ on a

$$\mathrm{wp}(e_2)(\lambda r.Q)(x_0 \oplus x_1)$$

d'où

$$(b = \mathsf{true} \Rightarrow \mathrm{wp}(e_2)(\lambda r.Q) \wedge b = \mathsf{false} \Rightarrow \mathrm{wp}(e_3)(\lambda r.Q))(x_0 \oplus x_1)$$

Dès lors par hypothèse de récurrence appliquée à e_1, à l'état x_0 et au prédicat $\lambda r.r = \mathsf{true} \Rightarrow \mathrm{wp}(e_2)(\lambda r.Q) \wedge r = \mathsf{false} \Rightarrow \mathrm{wp}(e_3)(\lambda r.Q)$ on a

$$\begin{array}{rl} & \mathrm{wp}(e_1)(\lambda r.r = \mathsf{true} \Rightarrow \mathrm{wp}(e_2)(\lambda r.Q) \wedge r = \mathsf{false} \Rightarrow \mathrm{wp}(e_3)(\lambda r.Q))(x_0) \\ = & \mathrm{wp}(e)(\lambda r.Q)(x_0) \end{array}$$

Idem si $b = \mathsf{false}$.

□

De ce lemme on déduit le corollaire suivant, qui constitue la réciproque du corollaire 2 :

Corollaire 3 *Soit Γ un environnement bien formé et e un programme tel que $\Gamma \vdash_r e : (\tau, \epsilon)$. Alors pour toutes pré- et post-conditions P et Q, on a*

$$[\![e]\!]_{(r:\tau,\epsilon,P,Q)} \quad \Rightarrow \quad (P \Rightarrow \mathrm{wp}(e)(\lambda r.Q))$$

Preuve. Par le Lemme 9 page 81, $[\![e]\!]_{(r:\tau,\epsilon,P,Q)}$ implique

$$\overline{e}\ \mathsf{r}_i\ (r : \tau, \epsilon, P, Q)$$

c'est-à-dire

$$\forall x.\ P(x) \Rightarrow \mathsf{let}\ (y, r) = (\overline{e}\ x)\ \mathsf{in}\ Q(y, r) \tag{2.23}$$

Supposons alors que la pré-condition P est vérifiée, et soit $(y, r) = (\overline{e}\ x)$. Alors par (2.23) on a $Q(y, r)$. Dès lors, par le lemme 12 ci-dessus on a $\mathrm{wp}(e)(\lambda r.Q)$. □

On constate que les corollaires 2 et 3 établissent l'équivalence entre notre notion de correction et la notion classique de correction définie en terme de plus faible pré-condition. On aurait donc pu commencer par définir la plus faible pré-condition, puis définir la notion de correction de la manière suivante :

$$[\![e]\!]_{(r:\tau,\epsilon,P,Q)} \quad \stackrel{\text{def}}{=} \quad P \Rightarrow \mathrm{wp}(e)(\lambda r.Q)$$

Théorème 5 *Soit Γ un environnement bien formé et e un programme tel que $\Gamma \vdash_r e : (\tau, \epsilon)$. Alors si e vérifie la spécification $(r : \tau, \epsilon, P, Q)$, c'est-à-dire si on a la proposition $[\![e]\!]_{(r:\tau,\epsilon,P,Q)}$, il existe une annotation de e telle que toutes les obligations de preuve contenues dans $\widehat{e}$ soient prouvables.*

Preuve. L'idée consiste à propager la post-condition Q à travers tout le programme e par des calculs successifs de plus faibles pré-conditions. La preuve s'établit par récurrence sur la structure de e :

$e \equiv E$ **:** On a une unique obligation de preuve dans e qui est $P \Rightarrow Q[r \leftarrow E[!z \leftarrow z]]$ c'est-à-dire $P \Rightarrow \mathrm{wp}(e)(\lambda r.Q)$, prouvable d'après le Corollaire 3.

$e \equiv e_1 \ ; \ e_2$ **:** On prend pour annotations externes de e_1 et e_2 les propositions suivantes :

$$\begin{array}{lcl} P_1 & = & \mathrm{wp}(e_1)(\lambda\_.Q_1) \\ Q_1 = P_2 & = & \mathrm{wp}(e_2)(\lambda r.Q_2) \\ Q_2 & = & Q \end{array}$$

Les programmes e_1 et e_2 vérifient ces spécifications, en vertu du Corollaire 2. Dès lors, on peut appliquer l'hypothèse de récurrence à ces deux programmes, et l'on obtient des programmes annotés dont les obligations de preuve seront toutes prouvables. De plus

$$\begin{array}{ll} \widehat{e} = \lambda x_0.\lambda h : P. & \mathsf{let}\ (x_1, \_, q_1) = (\widehat{e_1}\ x_0\ ?)\ \mathsf{in} \\ & \mathsf{let}\ (x_2, v, q_2) = (\widehat{e_2}\ (x_0 \oplus x_1)\ ?)\ \mathsf{in} \\ & (x_1 \oplus x_2, v, ?) \end{array}$$

La première obligation de preuve est $P \Rightarrow P_1$. Or $P_1 = \mathrm{wp}(e_1)(\lambda\_.\mathrm{wp}(e_2)(\lambda r.Q)) = \mathrm{wp}(e)(Q)$ et donc cette obligation est prouvable d'après le Corollaire 3. Les deux autres obligations sont établies immédiatement car $Q_1 = P_2$ et $Q_2 = Q$.

$e \equiv x := e_1$ **:** On prend pour annotations de e_1 les propositions

$$\begin{array}{lcl} P_1 & = & \mathrm{wp}(e_1)(\lambda x.Q_1) \\ Q_1 & = & Q[r \leftarrow \mathsf{void}] \end{array}$$

Par le même argument que pour le cas précédent, on peut appliquer l'hypothèse de récurrence à e_1. De plus

$$\widehat{e} = \lambda x_0.\lambda h : P.\mathsf{let}\ (x_1, v, q_1) = (\widehat{e_1}\ x_0\ ?)\ \mathsf{in}\ (x_1 \oplus \{x = v\}, \mathsf{void}, ?)$$

La première obligation est $P \Rightarrow P_1$, prouvable comme précédemment. La seconde est immédiate par choix de Q_1.

$e \equiv \mathsf{ref}\ e_1$ **:** On conserve pour e_1 la même spécification que pour e et la récurrence est immédiate.

$e \equiv \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ **:** On prend pour annotations de e_1 et e_2 les propositions suivantes :

$$\begin{array}{lcl} P_1 & = & \mathrm{wp}(e_1)(\lambda x.Q_1) \\ Q_1 = P_2 & = & \mathrm{wp}(e_2)(\lambda r.Q_2) \\ Q_2 & = & Q \end{array}$$

et l'on raisonne alors comme pour le cas de la séquence.

$e \equiv \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$ **:** On prend pour annotations de e_1, e_2 et e_3 les propositions suivantes :

$$\begin{array}{lcl} P_1 & = & \mathrm{wp}(e_1)(\lambda r.Q_1) \\ Q_1 & = & r = \mathsf{true} \Rightarrow P_2 \wedge r = \mathsf{false} \Rightarrow P_3 \\ P_2 & = & \mathrm{wp}(e_2)(\lambda r.Q_2) \\ P_3 & = & \mathrm{wp}(e_3)(\lambda r.Q_3) \\ Q_2 = Q_3 & = & Q \end{array}$$

On annote alors les programmes e_1, e_2 et e_3 par récurrence (la justification est toujours la même). De plus, on a

$$\begin{array}{l} \widehat{e} = \lambda x_0.\lambda h : P.\ \mathsf{let}\ (x_1, b, q_1) = (\widehat{e_1}\ x_0\ ?)\ \mathsf{in} \\ \qquad\qquad\qquad \mathsf{if}\ b\ \mathsf{then} \\ \qquad\qquad\qquad\quad \mathsf{let}\ (x_2, v, q_2) = (\widehat{e_2}\ (x_0 \oplus x_1)\ ?)\ \mathsf{in}\ (x_0 \oplus x_1 \oplus x_2, v, ?) \\ \qquad\qquad\qquad \mathsf{else} \\ \qquad\qquad\qquad\quad \mathsf{let}\ (x_2, v, q_3) = (\widehat{e_3}\ (x_0 \oplus x_1)\ ?)\ \mathsf{in}\ (x_0 \oplus x_1 \oplus x_2, v, ?) \end{array}$$

La première obligation de preuve est $P \Rightarrow P_1$, prouvable en vertu du Corollaire 3, car $P_1 = \mathrm{wp}(e)(\lambda r.Q)$. Supposons que $b = \mathsf{true}$. Alors la première obligation de preuve est $(b = \mathsf{true} \Rightarrow P_2 \wedge b = \mathsf{false} \Rightarrow P_3) \Rightarrow P_2$, trivialement prouvable. Enfin la troisième obligation de preuve est immédiate car on a posé $Q_2 = Q$. Idem si $b = \mathsf{false}$.

□

On remarquera qu'on obtient finalement une unique obligation de preuve non triviale, toutes les autres obligations étant immédiatement déchargées par une hypothèse. L'obligation restante est la suivante :

$$P \Rightarrow \mathrm{wp}(e)(\lambda r.Q)$$

Cela ne nous étonne guère, car nous avons montré à travers les lemmes 11 et 12 que cette proposition est équivalente à la correction du programme. Dans la pratique, cette proposition sera vite assez complexe, et elle nécessitera d'être décomposée en un ensemble de propositions plus simples. En particulier, une construction if donne une proposition de la forme :

$$b = \mathsf{true} \Rightarrow P_1 \wedge b = \mathsf{false} \Rightarrow P_2$$

et il est naturel de voir là deux obligations de preuve correspondant à chacune des branches du if — et ainsi de suite sur les propositions P_1 et P_2.

L'intérêt de l'utilisateur n'est donc pas que la correction soit exprimée par une unique proposition, mais plutôt par un ensemble de propositions correspondant chacune à un point du programme particulier, et chacune associée à un contexte particulier. C'est ce que nous obtenons directement avec notre méthode, et le calcul de plus faible pré-condition apparaît plutôt comme un résultat théorique affirmant qu'il est toujours possible d'annoter convenablement le programme dès lors que celui-ci est correct. En pratique, on pourra appliquer un calcul de plus faible pré-condition de façon automatique « en désespoir de cause », lorsqu'il est manifeste que l'absence d'une annotation donnera lieu à des obligations non prouvables, comme par exemple à l'issue d'une conditionnelle dont les branches comportent des effets de bord.

Conclusion

Dans ce chapitre, nous avons introduit une méthodologie de preuve de programmes impératifs dans le Calcul des Constructions Inductives, basée sur la construction d'une preuve partielle de sa spécification. La construction de ce terme est bâtie sur la traduction fonctionnelle définie au premier chapitre. Nous avons montré la correction de cette méthode (théorèmes 1 et 2), ainsi que sa complétude (théorème 5).

Si nous avons illustré le fonctionnement de cette méthode sur des exemples simples, il nous faut maintenant la valider de manière plus réaliste, sur des programmes non triviaux. Dans les deux prochains chapitres, nous allons présenter, d'une part, l'implantation de notre méthode dans le système Coq et, d'autre part, un certain nombre d'études de cas.

Chapitre 3

Implantation dans le système Coq

« The best theory is inspired by practice
and the best practice is inspired by theory. »
D. E. Knuth

J'ai été doublement incité à réaliser rapidement une implantation de la méthodologie que je viens de présenter : d'une part en tant que théoricien, et d'autre part en tant qu'informaticien. En tant que théoricien, tout d'abord, la confrontation de ses idées à l'effectivité est toujours la source d'un grand nombre de désillusions mais aussi d'un grand nombre d'idées nouvelles. C'est là en particulier que l'on s'aperçoit que la théorie n'est pas adéquate pour traiter des problèmes réels, et que des détails que l'on croyait uniquement techniques doivent être en fait incorporés dans la théorie. Mais la réciproque est également vraie, et c'est là le sens de la citation de D. E. Knuth ci-dessus[1] : des améliorations dans le discours théorique sont généralement bien plus efficaces que des bricolages du côté de l'implantation. C'est ainsi que nous avions commencé par donner un statut particulier aux tests et aux connectives logiques dans notre implantation avant de nous apercevoir que nous pouvions les faire entrer dans le cadre général, ce qui a eu pour effet de simplifier l'implantation et d'augmenter même l'expressivité des programmes. De manière générale plus l'adéquation est grande entre théorie et pratique, et mieux les deux s'en portent. C'est dans cet esprit que nous avons implanté notre méthode dans le système Coq.

D'autre part, j'ai été poussé à mettre mes idées en pratique en tant qu'informaticien, attiré par l'idée de pouvoir prouver la correction de programmes. Car c'est une idée intellectuellement très plaisante mais dont on ne parvient pas à saisir la difficulté sans méthode réellement effective. Quelle sera la taille de la spécification par rapport à la taille du programme ? Combien y aura-t-il d'obligations de preuve ? Seront-elles difficiles à prouver ou au contraire immédiates ? La taille de la preuve croîtra-elle linéairement avec la taille du programme ? etc. Répondre à ces questions « manuellement » est une entreprise laborieuse qui comporte un grand risque d'erreur, et l'utilisation d'un outil s'impose. Enfin l'utilisation du système Coq comme outil de spécification et de preuve apporte un plaisir supplémentaire par le caractère formel de la preuve et la possibilité de la rejouer en quelques minutes.

Ce chapitre a la structure suivante. Nous commencerons dans un premier temps par

[1] Cette citation est tirée d'un exposé intitulé *Theory and Practice*, reproduit dans [35] pages 149–167.

décrire l'implantation proprement dite. Nous donnerons sa présentation générale dans la section 3.1, en terme de commandes dans le système Coq et de syntaxe concrète de programmes et d'annotations, puis nous détaillerons son fonctionnement dans la section 3.2, en montrant en particulier comment il est intimement rattaché au discours théorique. Enfin, les sections 3.3 et 3.4 éclairciront certains détails techniques volontairement laissés dans l'ombre jusqu'ici, et décriront en particulier les différentes bibliothèques Coq qu'il a fallu développer autour de cette implantation.

3.1 Présentation

La méthodologie que nous venons de présenter dans les deux précédents chapitres a été implantée dans l'assistant de preuve Coq [1] durant les premiers mois de l'année 1998. Comme le système Coq, elle est écrite en Objective Caml [2], et son code est d'environ 6000 lignes, auxquelles s'ajoutent un millier de lignes de bibliothèques Coq. Cette implantation, bien que distribuée avec le système Coq, en est complètement indépendante et a été écrite séparément du reste du système. Si j'insiste sur ce point, c'est pour montrer la relative indépendance de la méthodologie présentée vis-à-vis des diverses implantations de théorie des types. La même méthode aurait pu être implantée tout aussi bien dans les systèmes LEGO ou PVS.

L'implantation se présente sous la forme d'une commande qui engendre les obligations de preuve à partir d'un programme annoté, et d'un certain nombre de commandes annexes permettant de manipuler l'environnement de programmes, par exemple pour introduire des variables globales ou encore visualiser l'environnement courant.

Nous allons donner ici des grammaires pour la syntaxe concrète des types et des programmes. Ces grammaires sont présentées au format BNF, où les symboles terminaux sont écrits dans la fonte sans sérif et les symboles non-terminaux en *italique*. Les parenthèses sont utilisées pour grouper, la barre verticale pour désigner l'alternative, les crochets le caractère optionnel, le symbole + une répétition d'au moins une occurrence et le symbole $\star$ une répétition d'un nombre quelconque d'occurrence, y compris aucune. Le symbole non-terminal *constr* désigne un terme quelconque du Calcul des Constructions Inductives.

L'environnement de programmes est une association de type de valeurs (notés τ dans le discours théorique) à des identificateurs. On déclare un nouvel objet dans cet environnement avec la syntaxe suivante :

`Global Variable` *nom* : *type-de-valeur*.

où la syntaxe concrète des types, de valeur et de calcul, est donnée Figure 3.1. Cette commande permet aussi bien d'introduire des variables globales, comme par exemple avec la commande

`Global Variable` x : Z ref.

que d'introduire une fonction et sa spécification, sans pour autant avoir besoin de la coder. Si par exemple on souhaite introduire une fonction *set_x* de profil

$$set\_x \quad : \quad (z : Z) \rightarrow (unit, (\emptyset, \{x\}), True, x = z)$$

c'est-à-dire une fonction qui fixe la valeur d'une référence globale x, alors on déclarera cette fonction par la commande

$$\texttt{Global Variable}\ set\_x\ :\quad \textsf{fun}\ (z : Z)\ \textsf{returns}\ u : unit\ \ \textsf{writes}\ x\ \textsf{post}\ x = z\ \textsf{end}.$$

La commande `Show Programs` permet de visualiser l'environnement de programmes courant.

$$\begin{array}{lcl}
\textit{type-de-valeur} & ::= & \textit{constr} \\
 & | & \textit{type-de-valeur}\ \textsf{ref} \\
 & | & \textsf{array}\ \textit{constr}\ \textsf{of}\ \textit{type-de-valeur} \\
 & | & \textsf{fun}\ \textit{lieurs}\ \textit{type-de-calcul} \\
\textit{type-de-calcul} & ::= & \textsf{returns}\ \textit{ident}\ : \textit{type-de-valeur} \\
 & & [\textsf{reads}\ \textit{liste-ident}]\ [\textsf{writes}\ \textit{liste-ident}] \\
 & & [\textsf{pre}\ \textit{constr}]\ [\textsf{post}\ \textit{constr}]\ \textsf{end} \\
\textit{lieurs} & ::= & ((\textit{liste-ident}\ : \textit{type-de-valeur}))+
\end{array}$$

FIG. 3.1: Syntaxe concrète des types

La commande implantant la méthode de preuve de programmes proprement dite a pour nom `Correctness` et sa syntaxe est la suivante :

`Correctness` *nom programme-annoté*.

où la syntaxe concrète des programmes annotés est donnée Figure 3.2. Cette commande a pour effet d'engendrer un certain nombre de buts, qui apparaissent alors à l'écran. Ce sont en fait des sous-buts d'un but initial unique, construit à partir de la spécification du programme. Le fonctionnement de la commande `Correctness` sera détaillé dans la section suivante. Les buts engendrés sont uniquement logiques, et portent sur des variables qui représentent les valeurs successives des références ou des tableaux manipulés par le programme. L'utilisateur peut alors s'attacher à les prouver, sans jamais avoir à manipuler d'expression de programme.

Une variante de la commande `Correctness` permet d'exécuter une tactique donnée sur tous les buts résultants. Sa syntaxe est la suivante :

`Correctness` *nom programme-annoté*; *tactique*.

3.2 Architecture

L'implantation que nous avons réalisée suit le plus fidèlement possible la théorie que nous avons présentée dans les deux chapitres précédents. Les quelques différences existantes ne sont pas nombreuses et exclusivement techniques ; elles seront présentées dans la section suivante. L'architecture de l'implantation étant un sujet relativement technique, nous renvoyons de lecteur à l'annexe B, page 165.

prog	::=	({ *prédicat* }) ⋆ *instruction* [{ *prédicat* }]
instruction	::=	*expression*
	|	*ident* := *prog*
	|	*ident* [*expression*] := *prog*
	|	let *ident* = ref *prog* in *prog*
	|	if *prog* then *prog* [else *prog*]
	|	while *prog* do *annot_boucle* *bloc* done
	|	begin *bloc* end
	|	let *ident* = *prog* in *prog*
	|	fun *lieurs* → *prog*
	|	let rec *ident* *lieurs* : *value_type*
		{ variant *arg_bf* } = *prog* [in *prog*]
	|	(*prog* *prog*)
expression	::=	*ident*
	|	! *ident*
	|	*ident* [*expression*]
	|	(*expression* +)
bloc	::=	*instr_bloc* [; *bloc*]
instr_bloc	::=	*prog*
	|	label *ident*
	|	assert { *prédicat* }
annot_boucle	::=	{ invariant *prédicat* variant *arg_bf* }
arg_bf	::=	*constr* [for *constr*]
prédicat	::=	*constr* [as *ident*]

FIG. 3.2: Syntaxe concrète des programmes annotés

Complexité. Il convient de faire ici une remarque sur la complexité de cette implantation. Le typage statique des effets, tel que nous l'avons présenté dans le discours théorique page 69, est linéaire par rapport à la taille du programme. D'autre part la construction du terme de preuve incomplet $\widehat{e}$, présentée page 75, est également linéaire par rapport à la taille du programme. Enfin, la tactique Refine a une exécution linéaire par rapport à la taille de son argument. On a donc finalement une tactique engendrant les obligations de preuve dont la complexité est linéaire en temps et espace par rapport à la taille du programme annoté. Dans la pratique, la constante est faible et les obligations de preuve engendrées presque instantanément pour un programme de moins de trente lignes.

Cependant, nous verrons dans la section suivante, page 117, qu'il est intéressant d'offrir un minimum d'inférence des effets dans le typage des fonctions récursives. Dès lors la complexité du typage d'une fonction récursive n'est plus linéaire en la taille de cette fonction, mais peux devenir exponentielle dans le pire des cas. Mais en pratique ce typage reste linéaire et cette amélioration du typage n'affecte pas les performances. C'est parfaitement analogue à l'inférence de type de ML qui est exponentielle dans le pire des cas, mais linéaire en pratique.

3.3 Détails techniques

Dans cette section, nous présentons des détails techniques qui n'ont pas été abordés dans le discours théorique, mais qui ont pour autant leur importance en pratique.

Tableaux

Comme nous l'avons déjà expliqué, les tableaux sont vus comme des références sur des tableaux fonctionnels, et ce à cause des alias potentiels entre éléments d'un même tableau — on ne peut décider statiquement si deux entiers i et j sont égaux ou non, donc on ne peut pas décider si $t[i]$ et $t[j]$ représentent le même élément du tableau t. Ces tableaux fonctionnels sont décrits dans la section suivante, page 120. Il y est défini un type $(\mathtt{array}\ N\ T)$ pour les tableaux de taille N contenant des valeurs de type T, muni d'opérations d'accès et de modification (`access` et `store`). Dès lors, on peut voir l'introduction des tableaux comme un simple sucre syntaxique, qui serait le suivant :

$$\begin{array}{lcl} t[e] & \leadsto & (\mathtt{access}\ !t\ e) \\ t[e_1] := e_2 & \leadsto & t := (\mathtt{store}\ !t\ e_1\ e_2) \end{array}$$

où t désigne donc en fait une référence. Dans ce cas, on peut voir le type des tableaux comme une abréviation du type des références sur des tableaux fonctionnels.

Si l'on préfère voir les constructions portant sur les tableaux comme des constructions primitives, il faut leur associer les règles de typage suivantes :

$$\frac{t : (\mathtt{array}\ n\ \tau) \in \Gamma \qquad \Gamma \vdash_a e : (\mathsf{int}, (\rho, \omega))}{\Gamma \vdash_a t[e] : (\tau, (\{t\} \cup \rho, \omega))}(\mathsf{TABACC}_a)$$

$$\frac{t : (\mathtt{array}\ n\ \tau) \in \Gamma \qquad \Gamma \vdash_a e_1 : (\mathsf{int}, (\rho_1, \omega_1)) \qquad \Gamma \vdash_a e_2 : (\tau, (\rho_2, \omega_2))}{\Gamma \vdash_a t[e_1] := e_2 : (\mathsf{unit}, (\{t\} \cup \rho_1 \cup \rho_2, \{t\} \cup \omega_1 \cup \omega_2))}(\mathsf{TABAFF}_a)$$

Elles sont analogues aux règles (DEREF_a) et (AFF_a) portant sur les références. Les termes de preuves incomplets sont les suivants :

$e \equiv \{P\}\ t[e_1]\ \{Q\}$ **:** Alors

$$\begin{array}{ll}\widehat{e} = \lambda x_0.\ \lambda h : P. & \mathsf{let}\ (x_1, v_1, q_1) = (\widehat{e_1}\ x_0\ ?)\ \mathsf{in} \\ & (x_1, (\mathtt{access}\ x_1.t\ v_1), ?)\end{array}$$

On notera que l'on va chercher la valeur de t dans x_1 et non dans x_0 car l'évaluation de e_1 peut modifier le tableau t, comme dans l'expression suivante[2] :

$$t[t[0]\ :=\ 1\ ;\ 0]$$

$e \equiv \{P\}\ t[e_1]\ :=\ e_2\ \{Q\}$ **:** Alors

$$\begin{array}{ll}\widehat{e} = \lambda x_0.\ \lambda h : P. & \mathsf{let}\ (x_1, v_1, q_1) = (\widehat{e_1}\ x_0\ ?)\ \mathsf{in} \\ & \mathsf{let}\ (x_2, v_2, q_2) = (\widehat{e_2}\ (x_0 \oplus x_1)\ ?)\ \mathsf{in} \\ & (x_1 \oplus x_2 \oplus \{t = (\mathtt{store}\ x_2.t\ v_1\ v_2)\}, \mathsf{void}, ?)\end{array}$$

Obligations de preuve portant sur les indexes. Lorsque l'on utilise la structure de tableau dans un programme, et que l'on cherche à prouver la correction de ce programme, se pose en particulier le problème de prouver que tous les indexes utilisés sont bien compris entre les bornes des tableaux. En effet, si l'on omet cette vérification, on peut parfaitement montrer la correction d'un programme dont l'exécution échoue à cause d'un accès dans un tableau en dehors de ses bornes. (Le comportement peut alors être très divers, et dépend du système ; cela peut être un « plantage » mais le programme peut tout aussi bien se poursuivre, des données ayant été modifiées de façon inconsistante.)

Pour introduire les obligations de preuve relatives aux indexes dans notre méthodologie, deux options au moins s'offrent à nous. La première consiste à définir les opérations fonctionnelles `access` et `store` de manière à ce qu'elles prennent en argument un index *et* une preuve que cet index se situe bien dans les bornes du tableau. Mais l'inconvénient est que l'on manipulerait alors dans les preuves des termes de la forme ($\mathtt{access}\ t\ i\ \pi$) ou π est une preuve que i est bien dans les bornes de t, et ce terme n'est pas nécessairement égal à ($\mathtt{access}\ t\ i\ \pi'$) si π' est une autre telle preuve.

Nous avons donc préféré une autre solution, qui consiste à introduire directement les obligations de preuve relatives aux indexes dans la construction des termes de preuve $\widehat{e}$. Nous les redéfinissons de la manière suivante :

$e \equiv \{P\}\ t[e_1]\ \{Q\}$ **:** Alors

$$\begin{array}{ll}\widehat{e} = \lambda x_0.\ \lambda h : P. & \mathsf{let}\ (x_1, v_1, q_1) = (\widehat{e_1}\ x_0\ ?)\ \mathsf{in} \\ & \mathsf{let}\ \pi_1 = ?: 0 \leq v_1 < (\mathsf{size}\ x_1.t)\ \mathsf{in} \\ & (x_1, (\mathtt{access}\ x_1.t\ v_1), ?)\end{array}$$

[2] À déconseiller !

$e \equiv \{P\}\ t[e_1] := e_2\ \{Q\}$ **:** Alors

$$\begin{array}{ll} \widehat{e} = \lambda x_0.\ \lambda h : P. & \mathsf{let}\ (x_1, v_1, q_1) = (\widehat{e_1}\ x_0\ ?)\ \mathsf{in} \\ & \mathsf{let}\ (x_2, v_2, q_2) = (\widehat{e_2}\ (x_0 \oplus x_1)\ ?)\ \mathsf{in} \\ & \mathsf{let}\ \pi_1 = ? : 0 \leq v_1 < (\mathsf{size}\ x_2.t)\ \mathsf{in} \\ & (x_1 \oplus x_2 \oplus \{t = (\mathtt{store}\ x_2.t\ v_1\ v_2)\}, \mathsf{void}, ?) \end{array}$$

Enfin les lemmes portant sur les opérations `access` et `store` ne sont définis que pour des indexes situés dans les bornes des tableaux considérés (voir page 120).

Annotations supplémentaires : label et assert

La syntaxe concrète donnée Figure 3.2 page 110 se distingue de la syntaxe abstraite du discours théorique, notamment dans la syntaxe des séquences. En effet, celles-ci ne sont plus binaires, i.e. de la forme $e_1\ ;\ e_2$, mais n-aires, i.e. de la forme begin $e_1;\ e_2;\ \ldots;\ e_n$ end, et elles peuvent contenir de plus, outre des expressions de programme, deux nouvelles constructions, label et assert.

Le caractère n-aire des séquences a pour but de simplifier la construction du terme de preuve pour lui donner un caractère homogène de la forme :

$$\begin{array}{ll} \lambda x_0.\ \lambda h : P. & \mathsf{let}\ (x_1, v_1, q_1) = \ldots\ \mathsf{in} \\ & \mathsf{let}\ (x_2, v_2, q_2) = \ldots\ \mathsf{in} \\ & \vdots \\ & \mathsf{let}\ (x_n, v_n, q_n) = \ldots\ \mathsf{in} \\ & \ldots \end{array}$$

et non en faire un « peigne » de la forme :

$$\begin{array}{lll} \lambda x_0.\ \lambda h : P. & \mathsf{let}\ (x_1, v_1, q_1) = \ldots\ \mathsf{in} & \\ & \mathsf{let}\ (x'_1, v'_1, q'_1) = & \mathsf{let}\ (x_2, v_2, q_2) = \ldots\ \mathsf{in} \\ & & \mathsf{let}\ (x'_2, v'_2, q'_2) = \ \ddots\ \ \mathsf{in} \\ & & \ldots \\ & \mathsf{in} & \\ & \ldots & \end{array}$$

La construction label permet d'introduire une étiquette dans une séquence, à la manière des étiquettes utilisées pour les instructions goto dans les vieux langages de programmation. Ces étiquettes serviront à désigner des points de programmes auxquels on pourra se référer dans les annotations lorsque l'on souhaitera parler de la valeur qu'avait une référence à un certain point de programmes. Nous détaillerons cette possibilité et ses avantages dans la section suivante.

La construction assert permet quant à elle d'introduire une annotation à l'intérieur d'une séquence, sans pour autant qu'elle soit la pré-condition ou la post-condition d'une expression de programme. Ainsi on peut écrire le morceau de programme suivant :

begin
  . . .
  $aux := !x;\ x := min(!x, !y);\ y := max(!aux, !y);$
  assert $\{\ x \leq y\ \}$;
  . . .
end

Cela s'avère très pratique lorsque la programmation est très impérative, et lorsqu'une propriété vraie à un instant est susceptible de servir plusieurs fois par la suite. On trouvera deux utilisations de cette annotation dans le Programme 4 page 127.

On remarquera que la construction assert correspond à la construction du même nom que l'on trouve dans la plupart des langages de programmation, dont le but est de vérifier une condition booléenne et d'arrêter le programme si cette dernière vaut false, à la différence près qu'ici la condition est une proposition quelconque et qu'une obligation de preuve est engendrée. On trouve assez souvent dans la littérature consacrée à l'algorithmique l'utilisation d'assertions (booléennes presque toujours) pour justifier certains algorithmes, et notamment dans l'excellent livre de J. L. Bentley *Programming Pearls* [7].

De l'inutilité des variables auxiliaires

Nous avons déjà abordé la question des variables auxiliaires dans la section 2.2.3, et nous avions expliqué à ce moment-là qu'elles n'étaient utilisées que pour relier entre elles dans les spécifications les valeurs successives des références. En effet, des post-conditions ne reliant que les valeurs des variables avant et après l'évaluation, telles que celles que nous avons introduites, ne sont pas toujours suffisantes, tel que le montre l'exemple suivant tiré de [10], page 940 (partiellement spécifié seulement) :

1. procedure F;
2. begin
3. $\{\ n = N \land s = S\ \}$
4.   if $n > 0$ then begin
5. $n := !n - 1;$
6. $F;\ \{\ s = S + 2^{N-1} - 1\ \}$
7. $s := !s + 1;$
8. $F;\ \{\ s = S + 2^{N} - 1\ \}$
9. $n := !n + 1;$
10.   end
11. $\{\ n = N \land s = S + 2^{N} - 1\ \}$
12. end

Ici la variable auxiliaire S dénote la valeur initiale de s. Dans la post-condition de la fonction F (ligne 11), il s'agit de la valeur avant l'évaluation du corps de F, et nous la dénoterions donc par $\overleftarrow{s}$. Mais à l'intérieur du corps de la fonction, la variable auxiliaire S est également utilisée, lignes 6 et 8, et comme elle dénote toujours la valeur initiale de s, elle ne dénote plus à ces deux endroits la valeur $\overleftarrow{s}$, qui désignerait à chaque fois la valeur de s avant l'appel récursif à F, c'est-à-dire au début de la ligne. Les prédicats avant-après ne sont donc pas suffisants.

Mais comme nous l'avons laissé entrevoir dans la section 2.2.3, les variables auxiliaires n'en sont pas rendues nécessaires pour autant. En effet, seule est nécessaire la possibilité de parler dans les spécifications des valeurs successives des références, et cette possibilité nous est offerte de façon quasi immédiate par la méthodologie que nous avons choisie. Nous avions déjà donné la possibilité de désigner deux points de programme particuliers : le point de programme courant et pour chaque post-condition le point de programme situé exactement avant l'évaluation de l'expression sur laquelle porte la post-condition. Nous généralisons cette possibilité en offrant la possibilité à l'utilisateur de désigner la valeur d'une référence à n'importe quel point de programme, ces points de programme pouvant être nommés grâce à des étiquettes introduites par la construction label, décrite dans la section précédente. La notation que nous avons choisie pour représenter la valeur de la référence r au point de programme i est la suivante :

$$r_{@i}$$

Une étiquette '0' est implicitement insérée au début de chaque programme, si bien que $r_{@0}$ représente toujours la valeur initiale de r, sans que l'utilisateur ait besoin d'introduire explicitement une telle étiquette. Enfin, $r_@$ sera la notation concrète pour $\overleftarrow{r}$. (On peut remarquer qu'en insérant automatiquement une étiquette devant chaque expression, la notation $r_@$ pourrait d'ailleurs s'exprimer comme une instance d'un $r_{@i}$.)

En pratique, cela nécessite tout de même quelques modifications par rapport au discours théorique. Tout d'abord, il faut typer les annotations dans les bons environnements, qui ne sont plus les environnements $Pre(\Gamma)$ et $Post(\Gamma, r : \tau)$ définis précédemment, mais des environnements qui dépendent maintenant également d'un ensemble d'étiquettes (représentant l'ensemble des points de programme traversés) et définis ainsi :

$$\begin{array}{lcl} Pre(\Gamma, \{l_1, \dots, l_m\}) & \stackrel{\text{def}}{=} & \widehat{\Gamma}, x^1, x^1_{@l_1}, \dots, x^1_{@l_m} : \widehat{\tau_1}, \dots, x^n, x^n_{@l_1}, \dots, x^n_{@l_m} : \widehat{\tau_n} \\ Post(\Gamma, r : \tau, L) & \stackrel{\text{def}}{=} & Pre(\Gamma, L), x^1_@ : \widehat{\tau_1}, \dots, x^n_@ : \widehat{\tau_n} \end{array}$$

où x^1, ..., x^n sont les références de Γ. De plus, il faut modifier dans la construction de la preuve incomplète $\widehat{e}$ le calcul des types des « trous » : il s'agit en effet de substituer dans l'annotation $x_{@l}$ par la variable représentant la valeur de x au point de programme l. Pour cela, on maintient lors de la traduction une structure de donnée conservant les points de programmes traversés ainsi que noms de variables introduits pour représenter les valeurs successives des références, et cette structure nous donne la substitution désirée.

On peut maintenant reprendre l'exemple de la procédure F ci-dessus et en donner la version annotée sans recours à des variables auxiliaires :

procedure F;
begin
    if $n > 0$ then begin
        label L;
        $n := !n - 1$;
        F; $\{\ s = s_{@L} + 2^{n_{@0}-1} - 1\ \}$
        $s := !s + 1$;
        F; $\{\ s = s_{@L} + 2^{n_{@0}} - 1\ \}$
        $n := !n + 1$;
    end
    $\{\ n = n_{@} \land s = s_{@} + 2^{n_{@}} - 1\ \}$
end

Les études de cas qui seront présentées dans la section 4 utilisent abondamment cette possibilité de spécification, et nous laisserons le lecteur en apprécier les avantages par rapport à une utilisation de variables auxiliaires, en terme de clarté. De plus, le recours à des variables auxiliaires entraîne des étapes de réécriture dans la preuve des obligations, que l'on évite ici.

Restrictions

Par rapport au discours théorique, l'implantation est restreinte en ce qui concerne les deux points suivants :

1. Dans le discours théorique nous avons utilisé une notion primitive d'enregistrements dans le langage cible pour définir la traduction fonctionnelle, puis le terme de preuve incomplet. Une telle notion n'existe pas dans le Calcul des Constructions Inductives, et pour y remédier nous avons représenté les ensembles de valeurs d'entrée et de sortie non pas par des enregistrements mais par des n-uplets de valeurs. Ces n-uplets sont définis comme des types inductifs à un constructeur et la bibliothèque `Tuples`, décrite page 119, regroupe ces définitions. Les valeurs sont rangées dans les n-uplets par ordre alphabétique des références qu'elles représentent.

2. Le cas des applications partielles à des références est aisément traité dans le cadre théorique par l'utilisation des variables de champs. Puisque, comme nous venons de le dire, nous ne disposions pas d'une telle notion dans le Calcul des Constructions Inductives mais que nous avons recouru à des n-uplets, l'application partielle nécessiterait d'introduire un opérateur effectuant explicitement la substitution qui résulte de l'application d'un terme abstrait par rapport à une variable de champ à un champ donné. Bien qu'il soit possible de définir un tel opérateur, sa définition et son utilisation s'avèrent très lourdes, et nous avons donc renoncé à son utilisation en pratique. Notre implantation n'accepte donc pas d'application partielle de fonction — du moins pas pour l'instant.

Fonctions récursives

Dans la pratique, il s'avère plus utile de manipuler directement des fonctions à n arguments, et c'est d'autant plus vrai pour les fonctions récursives. C'est pour cela que nous avons introduit directement la syntaxe suivante pour ces dernières :

$$\mathsf{let\ rec}\ f\ (x_1 : \tau_1) \cdots (x_n : \tau_n) : \tau\ \{\mathsf{variant}\ \nu\} = e\ [\ \mathsf{in}\ e\ ]$$

On remarque que, contrairement à la syntaxe abstraite donnée Figure 2.1 page 67, on ne déclare pas pour le résultat de f un type de calcul mais seulement un type de valeur τ. En effet, ainsi que nous l'avions déjà fait remarquer page 29, l'utilisateur ne souhaite pas devoir donner les effets et la spécification du corps de la fonction récursive, mais attend plutôt que ceux-ci soient inférés — seul le type du résultat doit être donné, car nous ne faisons pas d'inférence de type. La règle de typage modifiée est alors la suivante :

$$\frac{\begin{array}{c}\Gamma \vdash_a \vec{\tau}\ \mathsf{bf} \qquad \kappa = (\tau, \epsilon, P, Q) \qquad \Gamma, \vec{x} : \vec{\tau} \vdash_a \kappa\ \mathsf{bf} \\ \Gamma, f : (\vec{x} : \vec{\tau}) \rightarrow \kappa, \vec{x} : \vec{\tau} \vdash_a e : \kappa \qquad \mathit{Pre}(\Gamma, \vec{x} : \vec{\tau}) \vdash_{\mathrm{CIC}} \nu : \mathit{Variant}(A)\end{array}}{\Gamma \vdash_a \mathsf{rec}\ f\ (\vec{x} : \vec{\tau}) : \tau\ \{\mathsf{variant}\ \nu\} = e : ((\vec{x} : \vec{\tau}) \rightarrow \kappa, \emptyset)}\ (\mathsf{REC'}_a)$$

Il reste à justifier que nous pouvons bien inférer le type κ dans la règle ci-dessus. Pour cela, nous partons de la valeur initiale $\kappa_0 = (\tau, \bot, P_e, Q_e)$, où P_e et Q_e désignent les annotations du corps e de la fonction f, et nous itérons en prenant pour κ_{n+1} le type de e lorsque f est supposé retourner un calcul de type κ_n, c'est-à-dire :

$$\Gamma, f : (\vec{x} : \vec{\tau}) \rightarrow \kappa_n, \vec{x} : \vec{\tau} \vdash_a e : \kappa_{n+1}$$

On calcule ainsi un *plus petit point fixe*, dont l'existence est justifiée par le fait que l'effet est la seule composante de κ_n qui évolue, qu'il croit (pour l'inclusion ensembliste sur chacune de ces propres composantes), et qu'il est borné par l'union de l'ensemble des variables de Γ et de $\vec{x}$.

Ce calcul de plus petit point fixe aboutit généralement dès la valeur de κ_1, mais l'exemple suivant illustre un cas où un plus grand nombre d'étapes est nécessaire :

$$\begin{array}{l}\mathsf{let\ rec}\ f\ (x : \mathsf{int\ ref})\ (y : \mathsf{int\ ref}) : \mathsf{unit}\ \{\mathsf{variant}\ \nu\}\ = \\ \quad \mathsf{if}\ !x = 0\ \mathsf{then}\ x := 0\ \mathsf{else}\ (f\ y\ x)\end{array}$$

Dans cet exemple, les types inférés successivement sont au nombre de trois, et sont les suivants (le corps de f n'a pas d'annotation) :

$$\begin{array}{rcl}\kappa_0 & = & (\mathsf{unit}, \bot) \\ \kappa_1 & = & (\mathsf{unit}, (\{x\}, \{x\})) \\ \kappa_2 & = & (\mathsf{unit}, (\{x, y\}, \{x, y\}))\end{array}$$

On peut bien sûr obtenir un nombre d'étapes arbitraire en généralisant cet exemple de la façon suivante :

$$\begin{array}{l}\mathsf{let\ rec}\ f\ (x_1 : \mathsf{int\ ref})\ \cdots\ (x_n : \mathsf{int\ ref}) : \mathsf{unit}\ \{\mathsf{variant}\ \nu\}\ = \\ \quad \mathsf{if}\ !x_1 = 0\ \mathsf{then}\ x_1 := 0\ \mathsf{else}\ (f\ x_2\ x_3\ \ldots\ x_n\ x_1)\end{array}$$

où les types inférés successivement sont alors au nombre de $n+1$ et sont les suivants :

$$\begin{array}{lcl}
\kappa_0 & = & (\mathsf{unit}, \bot) \\
\kappa_1 & = & (\mathsf{unit}, (\{x_1\}, \{x_1\})) \\
\kappa_2 & = & (\mathsf{unit}, (\{x_1, x_2\}, \{x_1, x_2\})) \\
\vdots & & \\
\kappa_n & = & (\mathsf{unit}, (\{x_1, x_2, \dots, x_n\}, \{x_1, x_2, \dots, x_n\}))
\end{array}$$

Sucre syntaxique

Enfin, la syntaxe concrète de notre implantation est agrémentée d'un peu de sucre syntaxique, afin de rendre les programmes plus lisibles.

Expressions arithmétiques et connectives logiques. Nous pouvons déjà manipuler ces objets en tant qu'objets purement fonctionnels définis dans le Calcul des Constructions Inductives, mais dans la pratique on souhaite écrire

$$\mathsf{if}\ n + i \times p > m\ \mathsf{then}\ \dots$$

et non pas

$$\mathsf{if}\ (\texttt{Z\_gt\_le\_bool}\ (\texttt{Zplus}\ n\ (\texttt{Zmult}\ i\ p))\ m)\ \mathsf{then}\ \dots$$

La syntaxe concrète pour les expressions arithmétique concerne le type `Z` des entiers relatifs de la bibliothèque `ZArith` de `Coq`. Cette syntaxe est la suivante :

$$\begin{array}{lcl}
\textit{expression} & ::= & \textit{expression}\ \times\ \textit{expression} \\
 & | & \textit{expression}\ +\ \textit{expression} \\
 & | & \textit{expression}\ -\ \textit{expression} \\
 & | & -\ \textit{expression} \\
 & | & \textit{constante-entière}
\end{array}$$

La syntaxe concrète pour les expressions booléennes et les connectives logiques, quant à elle, est la suivante :

$$\begin{array}{lcl}
\textit{instruction} & ::= & \textit{prog}\ \mathsf{and}\ \textit{prog} \\
 & | & \textit{prog}\ \mathsf{or}\ \textit{prog} \\
 & | & \mathsf{not}\ \textit{prog} \\
 & | & \textit{expression}\ \textit{order_relation}\ \textit{expression} \\
 & & \\
\textit{order_relation} & ::= & > \mid >= \mid < \mid <= \mid = \mid <>
\end{array}$$

L'interprétation des relations d'ordre et des connectives logiques repose sur une bibliothèque, `ProgBool`, que l'on détaille dans la section suivante (page 120).

Boucle for. En dernier lieu, une construction for est ajoutée, avec la syntaxe suivante :

$$\text{for } i = v_1 \text{ to } v_2 \text{ do } \{ \text{ invariant } I \ \} \ e \text{ done}$$

où la variable i n'est *pas* une référence — comme c'est le cas en Caml, mais pas dans la plupart des vieux langages impératifs où l'index d'une boucle for peut être modifié à l'intérieur même de cette boucle. La construction for n'est que du sucre syntaxique pour la construction suivante :

$$\begin{array}{l}
(\text{let rec } f \ i \ \{ \text{ variant } v_2 + 1 - i \ \} \ = \\
\quad \{ \ i \leq v_2 + 1 \wedge I(i) \ \} \\
\quad (\text{if } i > v_2 \text{ then void else begin } e; \ (f \ (i+1)) \text{ end}) \\
\quad \{ \ I(v_2 + 1) \ \} \\
\ \text{in } (f \ v_1)) \\
\{ \ I(v_2 + 1) \ \}
\end{array}$$

Dès lors, si l'on écarte les obligations de preuve triviales, l'utilisateur aura à montrer d'une part l'initialisation de la boucle, à savoir

$$v_1 \leq v_2 + 1 \wedge I(v_1)$$

et d'autre part la conservation de l'invariant I par le corps de la boucle, soit

$$\{i \leq v_2 \wedge I(i)\} \ e \ \{I(i+1)\}$$

3.4 Bibliothèques

La tactique de preuve de correction de programmes s'accompagne d'un certain nombre de bibliothèques. On peut distinguer d'une part les bibliothèques nécessaires au fonctionnement de la tactique proprement dite et les bibliothèques d'usage plus général destinées à la spécification des programmes.

3.4.1 Bibliothèques de base de la tactique

Les bibliothèques de base de la tactique introduisent les définitions utilisées dans la construction du terme de preuve $\hat{e}$. On trouve les modules suivants :

`Tuples` Comme nous l'avons expliqué dans la section précédente, le Calcul des Constructions Inductives ne possède pas de notion primitive d'enregistrement et nous avons du y remédier en utilisant des n-uplets, définis pour l'occasion dans ce module comme des types inductifs. Ces n-uplets existent sous une version dépendante et une version non dépendante, suivant la présence ou l'absence de post-condition. Ils se nomment respectivement `tuple_`n pour les n-uplets non dépendants et `sig_`n pour les n-uplets dépendants (le prédicat étant dans Prop). Ils y sont définis jusqu'à $n = 10$, c'est-à-dire que les programmes ne peuvent contenir dans leurs effets plus de dix valeurs distinctes[3].

[3] Cela peut paraître une limitation excessive et il serait plus judicieux d'engendrer ces types inductifs « à la demande » — mais cela demanderait un peu plus de travail.

`ProgInt` Ce module se contente de charger le module `ZArith` de la bibliothèque standard de Coq, dans lequel est défini le type `Z` des entiers binaires de Coq, qui est le type des indexes de tableaux dans nos programmes.

`Arrays` Cette bibliothèque axiomatise des tableaux fonctionnels, que l'on peut donc voir plutôt comme des tables d'associations. Le type `array` des tableaux est polymorphe et contient l'information de taille du tableau. On a donc :

$$\texttt{array} : \texttt{Z} \to \mathsf{Set} \to \mathsf{Set}$$

Les trois opérations sur les tableaux sont la création, l'accès et la mise à jour, qui ont pour types respectifs

$$\begin{array}{lcl} \texttt{new} & : & \forall n : \texttt{Z}.\forall T : \mathsf{Set}.T \to (\texttt{array}\ n\ T) \\ \texttt{access} & : & \forall n : \texttt{Z}.\forall T : \mathsf{Set}.(\texttt{array}\ n\ T) \to \texttt{Z} \to T \\ \texttt{store} & : & \forall n : \texttt{Z}.\forall T : \mathsf{Set}.(\texttt{array}\ n\ T) \to \texttt{Z} \to T \to (\texttt{array}\ n\ T) \end{array}$$

On peut remarquer que les indexes ne sont pas ici restreints à l'intervalle $[0, n[$. Si l'on devait implanter un tel type de tableaux fonctionnels, on pourrait par exemple décider que la valeur renvoyée lors d'un accès en dehors des bornes est la valeur d'initialisation du tableau, et que la mise à jour en dehors des bornes est sans effet. Quoiqu'il en soit, ce sont les axiomes que l'on se donne sur cette structure de tableaux qui importent. En l'occurrence, ce sont les suivants :

$$\begin{array}{lcl} \texttt{new\_def} & : & \forall n : \texttt{Z}.\forall T : \mathsf{Set}.\forall v_0 : T.\forall i : \texttt{Z}. \\ & & 0 \leq i < n \Rightarrow (\texttt{access}\ (\texttt{new}\ n\ v_0)\ i) = v_0 \\ \\ \texttt{store\_def\_1} & : & \forall n : \texttt{Z}.\forall T : \mathsf{Set}.\forall t : (\texttt{array}\ n\ T).\forall v : T.\forall i : \texttt{Z}. \\ & & 0 \leq i < n \Rightarrow (\texttt{access}\ (\texttt{store}\ t\ i\ v)\ i) = v \\ \\ \texttt{store\_def\_2} & : & \forall n : \texttt{Z}.\forall T : \mathsf{Set}.\forall t : (\texttt{array}\ n\ T).\forall v : T.\forall i : \texttt{Z}.\forall j : \texttt{Z}. \\ & & 0 \leq i < n \Rightarrow 0 \leq j < n \Rightarrow \\ & & (\texttt{access}\ (\texttt{store}\ t\ i\ v)\ j) = (\texttt{access}\ t\ j) \end{array}$$

Enfin ce module étend la syntaxe de Coq par la construction t`#[`i`]`, pour (`access` t i), car de telles expressions apparaissent souvent dans les spécifications. (Mais nous continuerons d'utiliser par la suite la notation plus agréable $t[i]$ dans les spécifications.)

`ProgBool` Ce module introduit les définitions utilisées dans la traduction des expressions booléennes et des connectives logiques. Lors que l'on raisonne sur une conditionnelle, par exemple de la forme

$$\mathsf{if}\ x < y\ \mathsf{then}\ \ldots\ \mathsf{else}\ \ldots$$

alors on souhaite disposer directement dans chaque branche des propriétés $x < y$ et $x \geq y$, plutôt que de savoir que le résultat de la fonction booléenne appelée par ce test vaut respectivement true et false. Lorsque l'on fait des raisonnements dans le système Coq, on ne passe d'ailleurs pas par une fonction booléenne, mais on commence par démontrer le théorème

$$\texttt{Z\_lt\_ge\_dec} \quad : \quad \forall x, y : Z.\ \{x < y\} + \{x \geq y\}$$

où $\{A\}+\{B\}$ dénote la disjonction informative, définie dans le prélude du système Coq. On peut alors éliminer dans une preuve tout terme de la forme (`Z_lt_ge_dec` e_1 e_2) pour obtenir deux buts, l'un avec pour hypothèse $e_1 < e_2$ et l'autre avec $e_1 \geq e_2$. Pour obtenir le même confort dans les preuve de programmes, on montre le résultat suivant, transformant toute disjonction informative en une expression booléenne convenablement annotée :

$$\texttt{bool\_of\_sumbool} \quad : \quad \forall A, B : \mathsf{Prop}.\{A\} + \{B\} \Rightarrow \{b : \mathsf{bool} \mid \mathsf{if}\ b\ \mathsf{then}\ A\ \mathsf{else}\ B\}$$

où il s'agit ici de la construction if then else de Coq. On peut alors convertir tous les théorèmes de décidabilité de la bibliothèque standard de Coq en des fonctions booléennes correctement spécifiées. Ces fonctions se nomment `Z_lt_ge_bool`, `Z_eq_bool`, etc. À titre d'illustration, on a

$$\begin{array}{lcl} \texttt{Z\_lt\_ge\_bool} & = & \lambda x, y : Z.\ (\texttt{bool\_of\_sumbool}\ (\texttt{Z\_lt\_ge\_dec}\ x\ y)) \\ & : & \forall x, y : Z.\ \{b : \mathsf{bool} \mid \mathsf{if}\ b\ \mathsf{then}\ x < y\ \mathsf{else}\ x \geq y\} \end{array}$$

Il peut tout de même arriver qu'une expression booléenne se réduise à un booléen b non annoté. Il est alors automatiquement remplacé par l'expression (`annot_bool` b), où `annot_bool` est le théorème trivial suivant :

$$\texttt{annot\_bool} \quad : \quad \forall b : \mathsf{bool}.\{b' : \mathsf{bool} \mid \mathsf{if}\ b'\ \mathsf{then}\ b = \mathsf{true}\ \mathsf{else}\ b = \mathsf{false}\}$$

et l'on se retrouve alors avec des hypothèses $b = \mathsf{true}$ et $b = \mathsf{false}$ dans chaque branche, de manière classique.

Enfin, le module `ProgBool` définit les opérations sur les expressions booléennes correspondant aux connectives logiques, à savoir :

$$\begin{array}{lcl} \texttt{prog\_bool\_and} & : & \forall Q_1, Q_2 : \mathsf{bool} \to \mathsf{Prop}. \\ & & \{b : \mathsf{bool} \mid (Q_1\ b)\} \Rightarrow \{b : \mathsf{bool} \mid (Q_2\ b)\} \Rightarrow \\ & & \{b : \mathsf{bool} \mid \mathsf{if}\ b\ \ \mathsf{then}\ (Q_1\ \mathsf{true}) \wedge (Q_2\ \mathsf{true}) \\ & & \qquad\qquad\qquad \mathsf{else}\ (Q_1\ \mathsf{false}) \vee (Q_2\ \mathsf{false})\} \\[1ex] \texttt{prog\_bool\_or} & : & \forall Q_1, Q_2 : \mathsf{bool} \to \mathsf{Prop}. \\ & & \{b : \mathsf{bool} \mid (Q_1\ b)\} \Rightarrow \{b : \mathsf{bool} \mid (Q_2\ b)\} \Rightarrow \\ & & \{b : \mathsf{bool} \mid \mathsf{if}\ b\ \ \mathsf{then}\ (Q_1\ \mathsf{true}) \vee (Q_2\ \mathsf{true}) \\ & & \qquad\qquad\qquad \mathsf{else}\ (Q_1\ \mathsf{false}) \wedge (Q_2\ \mathsf{false})\} \\[1ex] \texttt{prog\_bool\_not} & : & \forall Q : \mathsf{bool} \to \mathsf{Prop}.\{b : \mathsf{bool} \mid (Q\ b)\} \Rightarrow \\ & & \{b : \mathsf{bool} \mid \mathsf{if}\ b\ \mathsf{then}\ (Q\ \mathsf{false})\ \mathsf{else}\ (Q\ \mathsf{true})\} \end{array}$$

`ProgWf` Ce module défini la famille de relation sur `Z` $\times$ `Z` définie par

$$(\texttt{Zwf}\ c\ x\ y) \quad \stackrel{\text{def}}{=} \quad c \leq x < y$$

et établit que (`Zwf` c) est bien fondée pour tout c. La relation (`Zwf` 0), qui correspond à la relation $<$ sur les entiers naturels, est la relation par défaut pour les variants dans le type Z lorsque celle-ci n'est pas donnée.

Ce module définit également la famille de relations « montantes » suivante :

$$(\texttt{Zwf\_up}\ c\ x\ y) \quad \stackrel{\text{def}}{=} \quad y < x \leq c$$

et établit que $(\texttt{Zwf\_up}\ c)$ est bien fondée pour tout c.

3.4.2 Bibliothèques d'usage général

Les bibliothèques d'usage général se limitent pour l'instant aux notions de tableaux triés et de permutations de tableaux, accompagnées des principaux résultats sur ces notions. Les trois modules concernés sont les suivants :

`Exchange` Cette bibliothèque définit la propriété pour deux tableaux t et t' de même taille n d'être identiques à l'exception de l'échange de deux de leurs éléments. Si i et j désignent les positions de cet échange, cette propriété est définie formellement de la manière suivante :

$$\begin{array}{lcl} \texttt{exchange}\ t\ t'\ i\ j & \stackrel{\text{def}}{=} & 0 \leq i,j < n \wedge t[i] = t'[j] \wedge t[j] = t'[i] \wedge \\ & & \forall k : \texttt{Z}.(0 \leq k < n \wedge k \neq i \wedge k \neq j) \Rightarrow t[k] = t'[k] \end{array}$$

Lorsque l'on échange deux éléments dans un tableau, on utilise généralement un code ressemblant à

$$\ldots\ \mathsf{let}\ a = t[i]\ \mathsf{in\ begin}\ t[i] := t[j];\ \ t[j] := a\ \mathsf{end}\ \ldots$$

et on obtient alors une traduction du programme de la forme

$$\begin{array}{ll} \ldots & \mathsf{let}\ a = t[i]\ \mathsf{in} \\ & \mathsf{let}\ t_1 = (\texttt{store}\ t_0\ i\ t[j])\ \mathsf{in} \\ & \mathsf{let}\ t_2 = (\texttt{store}\ t_1\ j\ a)\ \mathsf{in} \\ & \ldots \end{array}$$

qui donne, après substitution, un terme de la forme

$$\ldots\ \mathsf{let}\ t_2 = (\texttt{store}\ (\texttt{store}\ t_0\ i\ t_0[j])\ j\ t_0[i])\ \mathsf{in}\ \ldots$$

Le module `Exchange` établit donc en particulier le résultat suivant :

$$\begin{array}{lcl} \texttt{exchange\_proof} & : & \forall n : \texttt{Z}.\forall T : \mathsf{Set}.\forall t : (\texttt{array}\ n\ T).\forall i,j : \texttt{Z}. \\ & & 0 \leq i,j < n \Rightarrow \\ & & (\texttt{exchange}\ (\texttt{store}\ (\texttt{store}\ t\ i\ t[j])\ j\ t[i])\ t\ i\ j) \end{array}$$

Cela permet d'annoter tout échange dans un tableau par une post-condition de la forme $(\texttt{exchange}\ t\ t@\ i\ j)$, et de prouver alors l'obligation correspondante avec le lemme ci-dessus, sans avoir à déplier tout de suite la définition du prédicat `exchange`. Ce module montre également la symétrie de la propriété `exchange` sous la forme du lemme suivant :

$$\begin{array}{lcl} \texttt{echange\_sym} & : & \forall n : \texttt{Z}.\forall T : \mathsf{Set}.\forall t,t' : (\texttt{array}\ n\ T).\forall i,j : \texttt{Z}. \\ & & (\texttt{exchange}\ t\ t'\ i\ j) \Rightarrow (\texttt{exchange}\ t'\ t\ i\ j) \end{array}$$

`Permut` Ce module définit la notion de permutation sur les tableaux, utile lorsque l'on souhaite montrer qu'un tableau n'a été modifié que par permutations de ses éléments — en particulier lorsque l'on spécifie des programmes de tri. Il y a de multiples façons de définir un tel prédicat. On pourrait par exemple commencer par définir la notion de bijection, et définir la propriété pour deux tableau t et t' d'être des permutations l'un de l'autre par l'existence d'une bijection de t vers t'. Mais il est en fait plus simple de revenir à la définition même de permutation et de définir le prédicat de permutation comme la plus petite relation d'équivalence contenant les transpositions. Or, nous avons déjà défini dans le module précédent la notion d'échange, qui correspond à la spécification des transpositions. Dès lors, la propriété de permutation peut être définie inductivement de la manière suivante :

$$\begin{array}{l} \texttt{permut} : (\texttt{array}\ n\ T) \to (\texttt{array}\ n\ T) \to \mathsf{Prop} \quad \stackrel{\text{def}}{=} \\ \quad \begin{array}{lll} & \texttt{permut\_transp} & : \ \forall i,j : \texttt{Z}.(\texttt{exchange}\ t\ t'\ i\ j) \Rightarrow (\texttt{permut}\ t\ t') \\ \mid & \texttt{permut\_refl} & : \ (\texttt{permut}\ t\ t) \\ \mid & \texttt{permut\_sym} & : \ (\texttt{permut}\ t\ t') \Rightarrow (\texttt{permut}\ t'\ t) \\ \mid & \texttt{permut\_trans} & : \ \forall t''.(\texttt{permut}\ t\ t') \Rightarrow (\texttt{permut}\ t'\ t'') \Rightarrow (\texttt{permut}\ t\ t'') \end{array} \end{array}$$

Ce module définit également un raffinement de cette définition, où l'on précise un segment $[g,d]$ des tableaux sur lequel la permutation intervient, les éléments situées en dehors de ce segment étant inchangés. Ce prédicat est

$$(\texttt{sub\_permut}\ g\ d\ t\ t')$$

et sa définition est semblable à celle du prédicat `permut`. On montre en particulier le résultat suivant :

$$\texttt{sub\_permut\_is\_permut} \quad : \quad \forall g,d : \texttt{Z}.(\texttt{sub\_permut}\ g\ d\ t\ t') \Rightarrow (\texttt{permut}\ t\ t')$$

Enfin ce module définit le prédicat `array_id` qui exprime que deux tableaux t et t' sont identiques sur un segment $[g,d]$ et en montre quelques propriétés. Cette propriété sera notamment utilisée dans la preuve de l'algorithme *quicksort* page 129.

`Sorted` Ce module est une contribution de Nicolas Magaud, développée pendant sa preuve du tri par insertion (voir section 4.5). Il y a défini la notion de tableau trié pour les tableaux à valeurs dans `Z` et en a prouvé les propriétés essentielles. La propriété pour un tableau d'être trié sur le segment $[g,d]$ est définie de la manière suivante :

$$(\texttt{sorted\_array}\ n\ t\ g\ d) \quad \stackrel{\text{def}}{=} \quad \forall i : \texttt{Z}.\ g \le i < d \Rightarrow t[i] \le t[i+1]$$

Chapitre 4

Études de cas

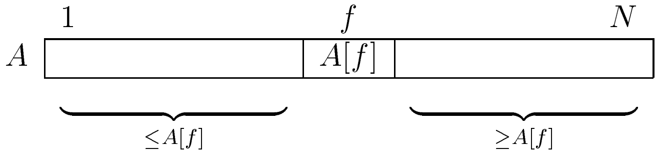
Nous allons maintenant présenter quelques études de cas réalisées avec la méthode que nous avons présentée au chapitre 2 et dont l'implantation dans le système `Coq` a été décrite dans le chapitre précédent. Ces études de cas sont au nombre de quatre, et concernent respectivement les algorithmes *Find* (section 4.1), *Quicksort* (section 4.2), *Knuth-Morris-Pratt* (section 4.3) et *Heapsort* (section 4.4). Nous détaillerons surtout la partie spécification et donnerons dans chaque cas les programmes annotés auxquels nous avons abouti. Nous donnerons également parfois le détail de certaines obligations de preuve. Ces preuves de correction ont été entièrement validées par l'implantation que nous avons réalisée dans le système `Coq` et présentée au chapitre précédent. Les sources de ces développements sont disponibles à l'URL `http://coq.inria.fr/contrib/`. Un bilan de ces études de cas est donné plus loin, section 4.6.1.

4.1 Find

Le programme que nous allons étudier ici a été l'objet d'un article de C. A. R. Hoare en 1971 intitulé *Proof of a program : Find* [25]. L'auteur y donne la preuve de correction et de terminaison d'un programme non trivial, *Find*, publié par lui-même dix ans auparavant [22]. Cette preuve formelle est faite « à la main », sans référence à l'axiomatique récemment développée par l'auteur dans l'article introductif *An axiomatic basis for computer programming* [24] dont nous avons déjà parlé. La preuve est menée par raffinements successifs, en partant de la spécification globale du problème. On ne peut qu'être impressionné de voir des idées si résolument modernes dans un article de 1971 ! À chaque étape les obligations de preuve sont énoncées et prouvées. Il y en aura au total 18. La terminaison est prouvée séparément, quoique formellement.

Le programme *find* est en fait plus connu dans la littérature sous le nom de *select*. Partant d'un tableau A de N entiers, indexés de 1 à N, et d'un entier f compris entre 1 et N, il consiste à réarranger les éléments du tableau de telle sorte que l'on ait, à l'issue de l'évaluation, la configuration suivante[1] :

[1] Bien que nos tableaux soient indexés de 0 à $N-1$, nous avons choisi de conserver exactement la spécification de l'article de Hoare, qui suppose des tableaux indexés de 1 à N. (En pratique, nous utilisons des tableaux de taille $N+1$, dont le premier élément est inutilisé.)



Le nom de *select* vient du fait qu'un tel algorithme est souvent utilisé pour extraire le f-ième ou les f plus petits éléments d'un tableau. Mais il est également utilisé dans des variantes de *quicksort*. (On pourra consulter [54] page 129 ou encore [34] pour en savoir plus sur le programme *select* et ses applications.)

Le programme *find* fût le premier exemple conséquent de preuve de correction réalisée avec la méthode que nous venons de présenter. Bien qu'ayant « essuyé les plâtres » d'une méthode en cours de développement, il s'est révélé être de loin l'exemple le plus intéressant, car il a permis une comparaison avec la logique de Hoare, qui s'est révélée plus que satisfaisante.

4.1.1 Spécification

La spécification complète du programme *find* peut être exprimée à travers quatre propriétés :

1. la « correction » de l'algorithme, à savoir la propriété qu'ont les éléments du tableau à l'issue de l'évaluation d'être dans la configuration souhaitée ;
2. la propriété pour les éléments du tableau de constituer à l'issue de l'évaluation une permutation des éléments initiaux ;
3. la vérification que les accès dans les tableaux (en lecture ou en écriture) se font toujours sur des indexes compris entre 1 et N ;
4. la terminaison de l'algorithme.

Correction. Le tableau A, sa taille N et l'hypothèse $1 \leq f \leq N$ sont déclarés de façon globale, comme dans l'article de Hoare [25]. La propriété que l'on souhaite obtenir à l'issue de l'algorithme, à savoir sa post-condition, est donnée par le prédicat *Found* portant sur le tableau A défini dans [25] de la manière suivante :

$$(Found\ A) \quad \stackrel{\text{def}}{=} \quad \forall p,q.1 \leq p \leq f \leq q \leq N \Rightarrow A[p] \leq A[f] \leq A[q] \tag{4.1}$$

Pour établir cette post-condition, l'auteur introduit des prédicats sur les variables m et n, *m_invariant* et *n_invariant*, que nous conservons et qui sont définis de la manière suivante :

$$\begin{aligned}
(n_invariant\ n\ A) &\stackrel{\text{def}}{=} f \leq n \land \forall p,q.1 \leq p \leq n < q \leq N \Rightarrow A[p] \leq A[q] \\
(m_invariant\ m\ A) &\stackrel{\text{def}}{=} m \leq f \land \forall p,q.1 \leq p < m \leq q \leq N \Rightarrow A[p] \leq A[q]
\end{aligned}$$

De même, il introduit des prédicats *i_invariant* et *j_invariant* sur les variables i et j :

$$\begin{aligned}
(i_invariant\ m\ n\ i\ r\ A) &\stackrel{\text{def}}{=} m \leq i \land \forall p.1 \leq p < i \Rightarrow A[p] \leq r \\
(j_invariant\ m\ n\ j\ r\ A) &\stackrel{\text{def}}{=} j \leq n \land \forall q.j < q \leq N \Rightarrow r \leq A[q]
\end{aligned}$$

Ces définitions sont regroupées dans le fichier `find_spec.v`.

Programme 4 Le programme *Find*

```
begin
  m := 1; n := N;
  while !m < !n do
    { invariant (m_invariant m A) ∧ (n_invariant n A)
      ∧ (permut A A@0) ∧ 1 ≤ m ∧ n ≤ N
      variant n − m }
    label B;
    r := A[f]; i := !m; j := !n;
    label C;
    while !i ≤ !j do
      { invariant (i_invariant m n i r A) ∧ (j_invariant m n j r A)
            ∧ (m_invariant m A) ∧ (n_invariant n A) ∧ 0 ≤ j ∧ i ≤ N + 1
            ∧ (termination i j i@C j@C r A) ∧ (permut A A@0)
        variant j − i for (Zwf (−N − 2)) }
      label L;
      while A[!i] < !r do
        { invariant (i_invariant m n i r A) ∧ i@L ≤ i ∧ i ≤ n
              ∧ (termination i j i@C j@C r A)
          variant N + 1 − i }
        i := !i + 1
      done;
      while !r < A[!j] do
        { invariant (j_invariant m n j r A) ∧ j ≤ j@L ∧ m ≤ j
              ∧ (termination i j i@C j@C r A)
          variant j }
        j := !j − 1
      done;
      assert { A[j] ≤ r ≤ A[i] };
      if !i ≤ !j then begin
        w := A[!i]; A[!i] := A[!j]; A[!j] := !w;
        assert { (exchange A A@L i j) ∧ A[i] ≤ r ≤ A[j] };
        i := !i + 1;
        j := !j − 1
      end
    done;
    assert { m < i ∧ j < n };
    if f ≤ !j then
      n := !j
    else if !i ≤ f then
      m := !i
    else
      begin n := f; m := f end
  done
end
{ (Found A) ∧ (permut A A@0) }
```

Permutation. Dans sa preuve de correction [25], C. A. R. Hoare n'établit pas la préservation des éléments du tableau en même temps que la correction de l'algorithme. Il consacre à cette propriété un paragraphe particulier (5.5 page 72), a posteriori, tout en en concédant l'importance puisque, comme il le fait remarquer lui-même, le programme

$$\textsf{for}\ i = 1\ \textsf{to}\ N\ \textsf{do}\ A[i] := i\ \textsf{done}$$

établit trivialement la propriété *Found*. Pour établir la permutation, C. A. R. Hoare fait remarquer que les seules modifications du tableau A sont des échanges d'éléments, que les échanges sont des permutations et que les compositions de permutations sont des permutations.

C'est en fait exactement comme cela que nous allons établir cette propriété, puisque notre prédicat *permut* sur les tableaux est justement défini comme la plus petite relation d'équivalence contenant les échanges (C.f. la bibliothèque *Permut* décrite dans la section 3.4). On adjoint donc à la post-condition la propriété $(permut\ A\ A_{@0})$.

Indexes. Les obligations de preuve relatives aux indexes des tableaux sont engendrées automatiquement, comme expliqué dans la section 3.3.

Terminaison. La terminaison fait dans [25] l'objet d'un paragraphe particulier (5.4 page 70). Pour chaque boucle while, l'argument de terminaison consiste en un variant dont C. A. R. Hoare montre qu'il décroit strictement. Pour établir cette décroissance pour les deux boucles portant respectivement sur i et j, l'auteur établit de nouveaux invariants de boucle sur ces deux variables. Nous les intégrons donc aux prédicats $i\_invariant$ et $j\_invariant$, qui sont ainsi redéfinis :

$$\begin{array}{lcl}(i\_invariant\ m\ n\ i\ r\ A) & \stackrel{\text{def}}{=} & \quad m \leq i \wedge \forall p.1 \leq p < i \Rightarrow A[p] \leq r \\ & & \wedge \quad i \leq n \Rightarrow \exists p.i \leq p \leq n \wedge r \leq A[p] \\ (j\_invariant\ m\ n\ j\ r\ A) & \stackrel{\text{def}}{=} & \quad j \leq n \wedge \forall q.j < q \leq N \Rightarrow r \leq A[q] \\ & & \wedge \quad m \leq j \Rightarrow \exists q.m \leq q \leq j \wedge A[q] \leq r\end{array}$$

La preuve de terminaison de la boucle portant sur i et j ne nécessite pas d'invariant supplémentaire, comme le fait remarquer l'auteur.

En revanche, la terminaison de la boucle la plus externe, portant sur m et n, est établie dans [25] d'une manière très particulière : afin de montrer que la distance entre m et n décroit strictement, l'auteur fait remarquer qu'il suffit de montrer qu'à l'issue de la boucle intermédiaire on a $m < i \wedge j < n$ (car on a ensuite l'une ou l'autre des affectations $m := !j$ ou $n := !i$). Et pour montrer cette propriété, l'auteur affirme que la condition du test if $!i \leq !j \ldots$ est satisfaite *au moins une fois*. Il montre plus exactement qu'elle est satisfaite la première fois. Nous ne pouvons utiliser un tel argument, qui fait intervenir le temps explicitement dans la spécification, mais nous pouvons le traduire en un invariant de boucle. Pour cela, nous introduisons le prédicat *termination* qui exprime que soit i et j ont été respectivement incrémenté et décrémenté, soit ils sont toujours de part et d'autre de f avec $A[f] = r$. Si i_0 et j_0 désignent les valeurs initiales de i et j, la définition de ce prédicat est la suivante :

$$(termination\ i\ j\ i_0\ j_0\ r\ A) \stackrel{\text{def}}{=} (i > i_0 \wedge j < j_0) \vee (i \leq f \leq j \wedge A[f] = r)$$

Cette définition est introduite dans le fichier `find_spec.v`.

4.1.2 Codage

Nous avons tenu à conserver autant que possible le programme de l'article de C. A. R. Hoare. Il y avait cependant dans le programme initial une construction goto pour sortir de la boucle principal. Plus précisément, la configuration était la suivante :

$$\begin{array}{l} \mathsf{while}\ \ m < n\ \mathsf{do} \\ \quad \vdots \\ \quad \ldots \mathsf{else}\ \ \mathsf{goto}\ L; \\ \mathsf{done}\ ; \\ L : \end{array}$$

Nous avons adapté cette partie du programme afin de ne pas recourir à la construction goto de la manière suivante[2] :

$$\begin{array}{l} \mathsf{while}\ \ m < n\ \mathsf{do} \\ \quad \vdots \\ \quad \ldots \mathsf{else}\ \ \mathsf{begin}\ \ m := f; n := f\ \mathsf{end} \\ \mathsf{done} \end{array}$$

Le reste du programme est inchangé. Le programme complètement spécifié est donné programme 4. On le trouve dans le fichier `find.v`. On peut noter l'utilisation des étiquettes pour indiquer la valeur d'une variable au début d'une boucle (respectivement C et L pour les boucles sur m, n et i, j), ainsi que l'utilisation de l'étiquette implicite 0 pour indiquer la valeur initiale du tableau A (par $A_{@0}$).

4.1.3 Preuve

Passé à la tactique Correctness, ce programme engendre 22 obligations de preuve. Parmi celles-ci on retrouve *exactement* les 18 lemmes que C. A. R. Hoare avait énoncés et prouvés dans [25]. Leurs preuves en sont faites dans le fichier `find_lemmas.v`. Des lemmes supplémentaires, nécessaires à la clarté de la preuve, sont établis dans le fichier `find_proof.v`.

4.2 Quicksort

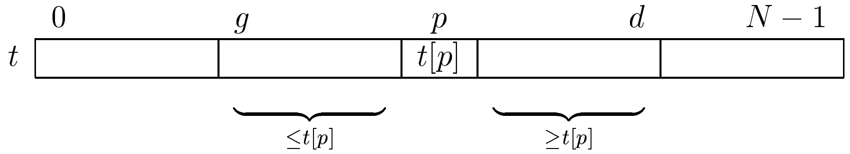
Il n'est plus utile de présenter *quicksort*, mais on oublie souvent que cet algorithme est dû à C. A. R. Hoare [23]. On pourra en trouver une présentation dans le livre de R. Sedgewick *Algorithms* [54], pages 115–131, qui y avait auparavant consacré sa thèse [53], et bien sûr une étude détaillée dans le volume 3 de *The art of computer programming* [34], pages 114–123.

[2] Les extrémistes de l'efficacité crieront que cela introduit un test inutile, mais je leur rétorquerai — comme je l'ai déjà fait — qu'une preuve de correction est bien plus importante que le prix d'un test arithmétique, par ailleurs ridicule.

Nous avons choisi sa présentation traditionnelle en une partie impérative, dite opération de « pivot », et une partie récursive, paradigme de la méthode « diviser pour régner ». En ce qui concerne sa partie impérative, la preuve de correction n'est a priori pas plus difficile que celle de *find*, qui réalise une opération similaire et même légèrement plus complexe. Mais la partie récursive, si elle n'est pas difficile à spécifier, s'avère être assez délicate à prouver. C'est une bonne occasion pour nous d'introduire un peu de *modularité* en découpant *quicksort* en plusieurs fonctions.

4.2.1 Partie impérative : pivot

La partie impérative de *quicksort* consiste, étant donné un segment $[g, d]$ d'un tableau t, à ré-arranger les éléments de ce segment de manière à se retrouver dans la situation suivante :



où p peut être quelconque dans l'intervalle $[g, d]$, et où les segments $[1, g[$ et $]d, N]$ de t restent inchangés. L'index p est alors appelé un *pivot*. On va spécifier et coder une fonction *pivot* réalisant cette opération et retournant la valeur du pivot. On commence donc par définir un prédicat *pivot_p* exprimant la propriété ci-dessus. C'est la conjonction de trois propriétés, la première exprimant que le pivot p se trouve dans l'intervalle $[g, d]$ et deux autres exprimant que les éléments à gauche et à droite du pivot sont respectivement plus petits et plus grands que $t[p]$. D'où les définitions suivantes :

$$
\begin{array}{lcl}
(\mathit{array\_le}\ t\ g\ d\ v) & \stackrel{\text{def}}{=} & \forall i.g \leq i \leq d \Rightarrow t[i] \leq v \\
(\mathit{array\_ge}\ t\ g\ d\ v) & \stackrel{\text{def}}{=} & \forall i.g \leq i \leq d \Rightarrow v \leq t[i] \\
(\mathit{pivot\_p}\ t\ g\ d\ p) & \stackrel{\text{def}}{=} & g \leq p \leq d \wedge (\mathit{array\_le}\ t\ g\ (p-1)\ t[p]) \\
 & & \qquad\qquad \wedge\ (\mathit{array\_ge}\ t\ (p+1)\ d\ t[p])
\end{array}
$$

Ces définitions sont contenues dans le fichier `Pivot.v`. La spécification de la fonction *pivot* est alors simple : sa pré-condition suppose que les bornes g et d sont telles que $0 \leq g < d < N$ et sa post-condition exprime que le tableau a été réarrangé de telle sorte que le résultat de la fonction (noté *result*) est un pivot pour le segment $[g, d]$ i.e.

$$(\mathit{pivot\_p}\ t\ g\ d\ \mathit{result}) \wedge (\mathit{sub\_permut}\ g\ d\ t\ t_{@})$$

Il est en fait plus pratique pour la suite de rajouter explicitement dans la post-condition la propriété $g \leq \mathit{result} \leq d$, même si celle-ci est déjà contenue dans le prédicat *pivot_p*.

Pour coder la fonction *pivot*, on commence par définir une procédure *swap* réalisant l'échange de deux éléments dans un tableau. Cette procédure est donnée programme 5. La spécification et la preuve en sont immédiates. On peut alors coder la fonction *pivot*, qui est donnée programme 6. Ces deux fonctions et leur preuve de correction sont contenues dans le fichier `Pivot_prog.v`.

Programme 5 La procédure *swap*

```
fun (t : array N of Z)(i, j : Z) →
  { 0 ≤ i < N ∧ 0 ≤ j < N }
    (let v = t[i] in
     begin
       t[i] := t[j];
       t[j] := v
     end)
  { (exchange t t@ i j) }
```

4.2.2 Partie récursive : diviser pour régner

La partie récursive de *quicksort* applique la célèbre technique « diviser pour régner ». Pour trier un segment $[g, d]$ du tableau t on appelle la fonction *pivot* qui, après avoir réarrangé le segment $[g, d]$, retourne la position p du pivot correspondant. On appelle alors récursivement la même méthode sur les segments $[g, p-1]$ et $[p+1, d]$, jusqu'à ce que les segments ne contiennent plus qu'au plus un élément. La fonction *quick_rec* qui implémente cet algorithme récursif a donc pour pré-condition la propriété $0 \leq g \wedge d < N$ et pour post-condition la propriété suivante :

$$(sorted\_array\ t\ g\ d) \wedge (sub\_permut\ g\ d\ t\ t_@)$$

qui exprime d'une part que le segment $[g, d]$ est trié et d'autre part que seuls les éléments de ce segment ont été permutés. Le code de la fonction *quick_rec* est donné programme 7, et est contenu dans le fichier `Quicksort_prog.v`. Sa preuve comporte une étape non triviale, lorsqu'il faut montrer la post-condition une fois l'appel à la fonction *pivot* et les deux appels récursifs effectués. Elle correspond à la situation suivante :

	0	g	p		d	$N-1$
t_0						
t_1	$= t_0$	$\leq v$	v	$\geq v$	$= t_0$	
t_2	$= t_1$	trié	v	$= t_1$	$= t_1$	
t_3	$= t_2$	$= t_2$	v	trié	$= t_2$	

L'établissement de cette post-condition fait l'objet d'un lemme, `quicksort_lemma`, démontré dans le fichier `Quicksort.v`, et dont l'énoncé exact est le suivant :

Programme 6 La fonction *pivot*

```
fun (t : array N of Z)(g, d : Z) →
 { 0 ≤ g < d ∧ d < N }
 (let pv = t[g] in
  let i = ref (g + 1) in
  let j = ref d in
  begin
    while !i < !j do
      { invariant g + 1 ≤ i ≤ d ∧ j ≤ d
              ∧ (array_le t (g + 1) (i − 1) pv) ∧ (array_ge t (j + 1) d pv)
              ∧ (sub_permut g d t t@0) ∧ t[g] = t@0[g]
        variant j − i for (Zwf (−N − 2)) }
      label L;
      while t[!i] ≤ pv and !i < !j do
        { invariant i@L ≤ i ≤ d ∧ (array_le t (g + 1) (i − 1) pv)
          variant d − i }
        i := !i + 1
      done;
      while t[!j] ≥ pv and !i < !j do
        { invariant g ≤ j ≤ j@L ∧ (array_ge t (j + 1) d pv)
          variant j }
        j := !j − 1
      done;
      if !i < !j then begin
        (swap t !i !j);
        i := !i + 1;
        j := !j − 1
      end
    done;
    if t[!i] < pv then begin
      (swap t g !i);
      !i
    end else begin
      (swap t g (!i − 1));
      !i − 1
    end
  end)
  { g ≤ result ≤ d ∧ (pivot_p t g d result) ∧ (sub_permut g d t t@) }
```

Programme 7 La fonction récursive *quick_rec*

$$
\begin{array}{l}
\mathsf{let\ rec}\ quick\_rec\ (t : \mathsf{array}\ N\ \mathsf{of}\ Z)(g, d : Z)\ :\ unit \\
\{\ \mathsf{variant}\ d - g\ \mathsf{for}\ (Zwf\ (-1))\ \}\ = \\
\quad \{\ 0 \leq g\ \wedge\ d < N\ \ \} \\
\quad (\mathsf{if}\ g < d\ \mathsf{then} \\
\qquad \mathsf{let}\ p\ =\ (pivot\ t\ g\ d)\ \mathsf{in} \\
\qquad \mathsf{begin} \\
\qquad\quad (quick\_rec\ t\ g\ (p-1)); \\
\qquad\quad (quick\_rec\ t\ (p+1)\ d) \\
\qquad \mathsf{end}) \\
\quad \{\ (sorted\_array\ t\ g\ d)\ \wedge\ (sub\_permut\ g\ d\ t\ t_{@0})\ \}.
\end{array}
$$

$$
\begin{array}{l}
\forall t_0, t_1, t_2, t_3.\ \forall g, d, p. \\
0 \leq g < d < N \\
(pivot\_p\ t_1\ g\ d\ p) \wedge (sub\_permut\ g\ d\ t_1\ t_0) \\
(sorted\_array\ t_2\ g\ (p-1)) \wedge (sub\_permut\ g\ (p-1)\ t_2\ t_1) \\
(sorted\_array\ t_3\ (p+1)\ d) \wedge (sub\_permut\ (p+1)\ d\ t_3\ t_2) \\
\hline
(sorted\_array\ t_3\ g\ d) \wedge (sub\_permut\ g\ d\ t_3\ t_0)
\end{array}
$$

Le programme *quicksort* lui-même n'est qu'une instance de *quick_rec* sur le segment constitué par le tableau tout entier, c'est-à-dire $[0, N-1]$. Sa spécification et sa preuve sont immédiates. Le programme *quicksort* est donné programme 8 et est contenu dans le fichier `Quicksort_prog.v`.

Programme 8 Le programme *quicksort*

$$
\begin{array}{l}
\mathsf{fun}\ (t : \mathsf{array}\ N\ \mathsf{of}\ Z)\ \rightarrow \\
\quad (quick\_rec\ t\ 0\ (N-1)) \\
\quad \{\ (sorted\_array\ t\ 0\ (N-1))\ \wedge\ (permut\ t\ t_{@})\ \}.
\end{array}
$$

4.3 Knuth-Morris-Pratt

Le troisième algorithme dont nous allons établir la correction et la terminaison est l'algorithme de Knuth-Morris-Pratt de recherche d'une chaîne de caractères dans une autre. Cet algorithme, très efficace, a été publié pour la première fois en 1970, dans un rapport technique de J. H. Morris et V. R. Pratt [42]. On pourra en trouver une présentation sommaire dans *Algorithms* [54], pages 281–285. J'ai repris ce dernier code, adapté pour des tableaux indexés à partir de 0 et non de 1.

La chaîne de caractères dont on recherche une occurrence s'appelle le *motif* et l'on suppose qu'il s'agit d'un tableau p de taille M. La chaîne dans laquelle on recherche l'occurrence s'appelle le *texte* et l'on suppose qu'il s'agit d'un tableau a de taille N. Le type des éléments des tableaux p et a est un type A quelconque, sur lequel on suppose seulement disposer d'une égalité décidable, à travers la fonction de comparaison $A\_eq\_bool : A \to A \to \mathsf{bool}$.

L'idée principale de l'algorithme de Knuth-Morris-Pratt est la suivante : lorsque l'on a comparé avec succès j éléments du motif et du texte mais que les caractères suivants du texte et du motif sont différents, on peut exploiter la connaissance que l'on a du motif pour décaler celui-ci *le plus possible* vers la droite, et non pas d'une unité seulement comme on le ferait dans la méthode brutale qui essaye toutes les occurrences possible de p dans a. On suppose donc que l'on est dans la situation illustrée figure 4.1 et que l'on cherche la plus grande valeur de n telle que $[p_{j-n} \dots p_{j-1}] = [p_0\, p_1 \dots p_{n-1}]$. Cette valeur de n dépend bien

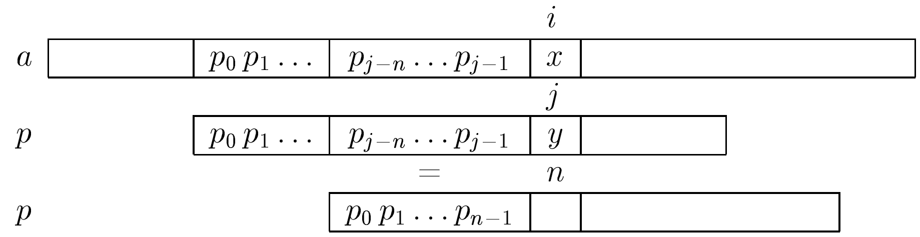


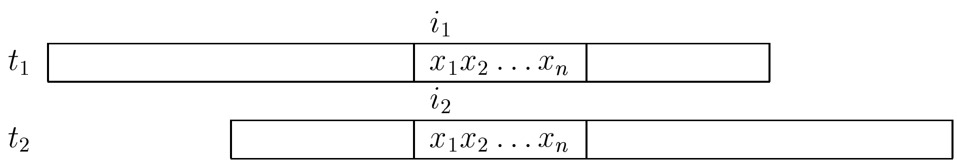
FIG. 4.1: Décalage du motif

sûr de j, mais ne dépend que du motif p. On peut donc calculer à l'avance la valeur de n pour chaque j, que l'on conservera dans une table *next* indexée par j.

Une fois cette table construite, la recherche est assez facile. On va donc commencer par prouver l'algorithme de recherche en supposant que l'on dispose d'une procédure *initnext* construisant la table *next*, puis on prouvera dans un second temps une implantation de cette procédure.

4.3.1 Spécification de la table *next*

Afin de spécifier l'algorithme de recherche, et en particulier la table *next*, on commence par définir un prédicat $(match\ t_1\ i_1\ t_2\ i_2\ n)$ exprimant que les tableaux $t_1[0 \dots N_1 - 1]$ et $t_2[0 \dots N_2 - 1]$ ont en commun les segments respectifs $[i_1, i_1 + n - 1]$ et $[i_2, i_2 + n - 1]$, c'est-à-dire que l'on est dans la situation suivante :



Ce prédicat se définit formellement de la façon suivante :

$$(match\ t_1\ i_1\ t_2\ i_2\ n) \stackrel{\text{def}}{=} \begin{array}{ll} & 0 \le i_1 \le N_1 - n \land 0 \le i_2 \le N_2 - n \\ \land & (\forall i : Z.\ 0 \le i < n \Rightarrow t_1[i_1 + i] = t_2[i_2 + i]) \end{array} \tag{4.2}$$

On montre alors quelques propriétés de ce prédicat telles que la symétrie, la transitivité, l'affaiblissement à gauche ou à droite (i.e. diminution de n), etc. La définition du prédicat *match* et ses propriétés sont contenues dans le fichier `Match.v`.

On définit ensuite un prédicat *Next* qui va exprimer ce que l'on a illustré par la figure 4.1, à savoir que le décalage $j \to n$ est tel que les deux segments $[j-n, j-1]$ et $[0, n-1]$ du motif se superposent et que n est maximal pour cette propriété dans l'intervalle $[0, j[$. On définit donc

$$\begin{array}{rcl}(Next\ p\ j\ n) & \stackrel{\text{def}}{=} & 0 \leq n < j \\ & \wedge & (match\ p\ (j-n)\ p\ 0\ n) \\ & \wedge & (\forall z : Z.\ n < z < j \Rightarrow \neg(match\ p\ (j-z)\ p\ 0\ z))\end{array} \tag{4.3}$$

Le prédicat *Next* et quelque unes de ces propriétés sont contenues dans le fichier `Next.v`. La seule supposition que l'on fait alors sur la procédure *initnext* est qu'elle établit la post-condition suivante :

$$\forall j : Z.\ 0 < j < M \Rightarrow (Next\ p\ j\ next[j])$$

4.3.2 Algorithme de recherche

L'algorithme de recherche est donné programme 9. Il se présente sous la forme d'une fonction *kmp* retournant l'indice de la première occurrence de p dans a. Étant donné que la recherche peut échouer, on décide que *kmp* retourne une valeur supérieure ou égale à N en cas d'échec. Ainsi, spécifier qu'il n'y a pas d'occurrence de p avant la valeur retournée par la fonction *kmp* exprime aussi bien qu'il s'agit de la première occurrence en cas de succès que le fait qu'il n'y ait pas d'occurrence en cas d'échec. On introduit donc le prédicat $(first\_occur\ r)$ qui exprime que r est l'indice de la première occurrence de p dans a lorsque que $r < N$ et qu'il n'y a pas de telle occurrence pour un indice plus petit que r :

$$\begin{array}{rcl}(first\_occur\ p\ a\ r) & \stackrel{\text{def}}{=} & (0 \leq r < N \Rightarrow (match\ a\ r\ p\ 0\ M)) \\ & \wedge & (\forall k : Z.\ 0 \leq k < r \Rightarrow \neg(match\ a\ k\ p\ 0\ M))\end{array} \tag{4.4}$$

et la post-condition de la fonction *kmp* est directement $(first\_occur\ p\ a\ result)$.

La spécification de la fonction *kmp* ne pose pas de problème — son invariant exprime que l'on a à présent une occurrence de longueur j et qu'aucune occurrence de longueur M n'a été encore trouvée — mais sa terminaison, en revanche, est plus délicate. On constate en effet que la variable i est incrémentée dans deux cas sur trois, mais que dans le troisième on se contente de réaliser l'affectation $j := next[!j]$. Dans ce cas, j décroit car le prédicat $Next$ sur j implique que $next[j] < j$ (voir définition 4.3 ci-dessus), et i reste inchangée. Dès lors, on va choisir comme variant la paire $(N-i, j)$ pour l'ordre lexicographique sur les couples d'entiers naturels, ordre que l'on appelle *lexZ*. Cet ordre et la preuve de son caractère bien-fondé sont contenues dans le fichier `Lex.v`. Celui-ci utilise essentiellement des définitions et des résultats du chapitre `RELATIONS` de la bibliothèque standard de Coq.

Le code spécifié et la preuve de correction de la fonction *kmp* sont contenues dans le fichier `KMP.v`.

Programme 9 Le programme *kmp*

$$
\begin{array}{l}
(\mathsf{let}\ i\ =\ \mathsf{ref}\ 0\ \mathsf{in} \\
\ \mathsf{let}\ j\ =\ \mathsf{ref}\ 0\ \mathsf{in} \\
\ \mathsf{begin} \\
\quad (\mathit{initnext}\ \mathit{tt}); \\
\quad \mathsf{while}\ !j < M\ \mathsf{and}\ !i < N\ \mathsf{do} \\
\qquad \{\ \mathsf{invariant}\ 0 \le j \le M\ \wedge\ j \le i \le N\ \wedge\ (\mathit{match}\ a\ (i-j)\ p\ 0\ j) \\
\qquad\qquad\quad \wedge\ \forall k : Z.\ 0 \le k < i - j\ \Rightarrow\ \neg(\mathit{match}\ a\ k\ p\ 0\ M) \\
\qquad\quad \mathsf{variant}\ (\mathit{pairZ}\ (N-i)\ j)\ \mathsf{for}\ \mathit{lexZ}\ \ \} \\
\qquad \mathsf{if}\ (\mathit{A\_eq\_bool}\ a[!i]\ p[!j])\ \mathsf{then\ begin} \\
\qquad\quad i\ :=\ !i+1;\ j\ :=\ !j+1 \\
\qquad \mathsf{end\ else} \\
\qquad\quad \mathsf{if}\ !j\ =\ 0\ \mathsf{then}\ i\ :=\ !i+1\ \mathsf{else}\ j\ :=\ \mathit{next}[!j] \\
\quad \mathsf{done}; \\
\quad \mathsf{if}\ !j\ =\ M\ \mathsf{then}\ !i - M\ \mathsf{else}\ !i \\
\ \mathsf{end}) \\
\ \{\ (\mathit{first\_occur}\ p\ a\ \mathit{result})\ \ \}
\end{array}
$$

4.3.3 Construction de la table *next*

Reste à construire la table *next*. Comme on le constate rapidement en jetant un coup d'oeil au code (programme 10), l'algorithme de construction de la table *next* est très similaire à l'algorithme de la recherche elle-même. Cela n'est guère surprenant puisque l'on cherche justement des superpositions entre le motif et lui-même.

L'invariant exprime le fait que l'on a déjà calculé $next[k]$ pour $0 < k \le i$ et contient les propriétés nécessaires pour établir le prédicat *Next* pour la valeur suivante, à savoir $i + 1$. En effet, i et j sont incrémentés *avant* que l'affectation $next[!i] := !j$ ne soit faite, et donc la deuxième ligne de l'invariant peut surprendre parce qu'elle exprime justement la propriété attendue pour $i+1$ et $j+1$. La terminaison se prouve tout comme pour la fonction *kmp*, en prenant la paire $(M - i, j)$ comme variant pour l'ordre lexicographique *lexZ*.

Le code spécifié et la preuve de la procédure *initnext* sont contenues dans le fichier `KMP.v`.

4.4 Heapsort

La dernière étude de cas que nous présenterons est une preuve de correction de l'algorithme de *tri par tas* (en anglais *heapsort*). L'algorithme est à l'origine dû à R. W. Floyd [12], et sa version en place est dûe à J. W. J. Williams [58]. Comme pour *quicksort*, on en trouvera une description dans le livre de R. Sedgewick *Algorithms* [54], pages 153–158 et encore une fois une étude détaillée dans le volume 3 de *The art of computer programming* [34], pages 149–153.

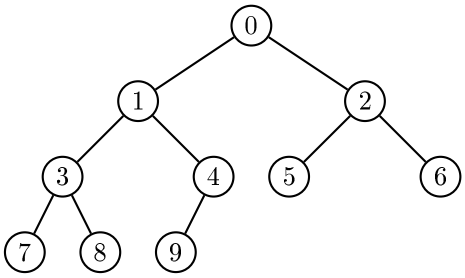
Cet algorithme de tri a la particularité d'avoir une complexité en $N \log(N)$ dans *le pire*

Programme 10 La procédure *initnext*

$$
\begin{array}{l}
\mathsf{fun}\ (u : unit)\ \rightarrow \\
\ (\mathsf{let}\ i\ =\ \mathsf{ref}\ 1\ \mathsf{in} \\
\ \ \mathsf{let}\ j\ =\ \mathsf{ref}\ 0\ \mathsf{in} \\
\ \ \mathsf{if}\ 1 < M\ \mathsf{then\ begin} \\
\quad next[1]\ :=\ 0; \\
\quad \mathsf{while}\ !i < M - 1\ \mathsf{do} \\
\qquad \{\ \mathsf{invariant}\ 0 \leq j \leq M\ \wedge\ j < i \leq M\ \wedge\ (match\ p\ (i-j)\ p\ 0\ j) \\
\qquad\qquad \wedge\ (\forall z : Z.\ j+1 < z < i+1\ \Rightarrow\ \neg(match\ p\ (i+1-z)\ p\ 0\ z)) \\
\qquad\qquad \wedge\ (\forall k : Z.\ 0 < k \leq i\ \Rightarrow\ (Next\ p\ k\ next[k])) \\
\qquad\ \ \mathsf{variant}\ (pairZ\ (M-i)\ j)\ \mathsf{for}\ lexZ\ \ \} \\
\quad\ \ \mathsf{if}\ (A\_eq\_bool\ p[!i]\ p[!j])\ \mathsf{then\ begin} \\
\qquad i\ :=\ !i+1;\ j\ :=\ !j+1;\ next[!i]\ :=\ !j \\
\quad\ \ \mathsf{end\ else} \\
\qquad \mathsf{if}\ !j\ =\ 0\ \mathsf{then\ begin}\ i\ :=\ !i+1;\ next[!i]\ :=\ 0\ \mathsf{end\ else}\ j\ :=\ next[!j] \\
\quad \mathsf{done} \\
\ \ \mathsf{end}) \\
\ \{\ \forall j : Z.\ 0 < j < M\ \Rightarrow\ (Next\ p\ j\ next[j])\ \ \}
\end{array}
$$

des cas, ce qui est la complexité optimale de tout algorithme de tri par échanges. L'idée consiste à construire à partir des éléments du tableau de départ une structure de *tas*. Un tas est un arbre binaire complet dans lequel tout nœud est supérieur ou égal à chacun de ses fils. (La structure de tas est généralement utilisée pour coder des *files de priorité* ; voir par exemple [54], chapitre 11.) La structure de tas permet de donner en temps constant le plus grand des éléments qu'elle contient (il s'agit de la racine du tas). D'autre part, les opérations d'ajout et de suppression peuvent être réalisées en temps $\log(N)$, si N est le nombre d'éléments présents dans le tas. Dès lors, on comprend comment en déduire un programme de tri efficace : on insère un par un tous les éléments du tableau dans un tas initialement vide, ce qui prend un temps $N \log(N)$, puis on retire successivement tous les éléments du tas en prenant à chaque fois le plus grand, ce qui permet de les réarranger en ordre décroissant, et ce qui prend également un temps $N \log(N)$.

L'un des attraits de l'algorithme *heapsort* est qu'il peut être réalisé *en place*. En effet, on peut représenter un tas dans un tableau en mettant les deux fils d'un nœud i aux positions respectives $2i+1$ et $2i+2$ (on considère ici des tableaux indexés à partir de 0). Pour $N = 10$, cela donne le codage suivant :



En partant d'un tableau t quelconque, il est possible d'en permuter les éléments pour en faire un tas (nous verrons plus loin comment). Une fois le tas construit dans t, on peut également le trier en place en maintenant dans sa partie gauche le tas formé par les éléments restant à trier et dans sa partie droite des éléments déjà triés (et à leur place définitive), de la manière suivante :

	0	$k+1$ $\quad$ $N-1$
t	tas formé d'éléments $\leq t[k+1]$	partie déjà triée

Nous avons repris le code de R. Sedgewick ([54], pages 152 et 156). Ce code, incroyablement concis, n'utilise qu'une seule fonction de manipulation du tas, appelée *downheap*, dont le rôle est de faire descendre à sa place dans le tas la valeur située à la racine (en supposant que toutes les autres valeurs forment déjà un tas). Cette seule fonction permet d'une part de construire le tas (de bas en haut) et d'autre part de maintenir la structure de tas à chaque fois que l'on en retire le plus grand élément. Il en résulte un code d'une dizaine de lignes seulement !

4.4.1 Définition de la structure de tas

Pour spécifier le programme de tri proprement dit, nous disposons déjà des définitions adéquates, introduites avec la preuve de correction de *quicksort*. Mais pour annoter la fonction intermédiaire et les invariants de boucles de *heapsort*, il nous faut définir la notion de tas. Plus précisément, nous allons définir un prédicat ($heap\ t\ n\ k$), dont la signification précise est la suivante : « Dans le tableau t, l'arbre de racine k formé d'éléments d'indices inférieurs ou égaux à n est un tas ». Le prédicat *heap* est défini inductivement de la manière suivante :

$$\frac{\begin{array}{c}2k+1 \leq n \Rightarrow t[k] \geq t[2k+1] \wedge (heap\ t\ n\ (2k+1)) \\ 2k+2 \leq n \Rightarrow t[k] \geq t[2k+2] \wedge (heap\ t\ n\ (2k+2))\end{array}}{(heap\ t\ n\ k)}$$

Le prédicat *heap* et ses propriétés sont contenus dans le fichier `Heap.v`. À la structure de tas correspond un principe de récurrence assez inhabituel, mais qui se révèle pratique pour montrer certains lemmes. Ce principe, établi dans le module `Heap`, est le suivant :

$$\begin{array}{l}\forall P : Z \rightarrow \mathsf{Prop}. \\ \quad (P\ 0) \Rightarrow \\ \quad (\forall x : Z.\ 0 \leq x \Rightarrow (P\ x) \Rightarrow (P\ (2x+1)) \wedge (P\ (2x+2))) \Rightarrow \\ \quad \forall x : Z.\ 0 \leq x \Rightarrow (P\ x)\end{array}$$

Un autre prédicat nous sera utile dans la spécification du code de *heapsort*. Il exprime que tous les éléments d'un arbre donné — toujours représenté au sein d'un tableau par le codage précédemment introduit — ne contient que des valeurs inférieurs ou égales à une valeur v donnée. Pour un tableau t, une racine k et des indices inférieurs ou égaux à n, ce prédicat, noté $(\mathit{inftree}\ t\ n\ v\ k)$, est défini inductivement de la manière suivante :

$$\frac{t[k] \leq v \quad 2k+1 \leq n \Rightarrow (\mathit{inftree}\ t\ n\ v\ (2k+1)) \quad 2k+2 \leq n \Rightarrow (\mathit{inftree}\ t\ n\ v\ (2k+2))}{(\mathit{inftree}\ t\ n\ v\ k)}$$

Le prédicat *inftree* et ses propriétés sont contenus dans le fichier `Inftree.v`.

4.4.2 La procédure *downheap*

L'unique fonction auxiliaire nécessaire à l'écriture de *heapsort* est une procédure appelée *downheap* dont la spécification est la suivante. Elle prend en entrée un tableau t et deux entiers k et n, et suppose que l'arbre de racine k et d'éléments d'indices inférieurs ou égaux à n forme un tas, à l'exception peut-être de sa racine $t[k]$. Dès lors, elle rétablit une structure de tas sur l'arbre de racine k en faisant « descendre $t[k]$ à sa place ». Pour cela, on sélectionne le plus grand des deux fils de la racine et s'il se trouve être plus grand que la racine elle-même on les échange tous les deux et on recommence la même opération sur le fils sélectionné. Sinon, on a terminé et l'arbre obtenu est bien un tas.

Cette procédure est généralement écrite avec une boucle while et une construction break de la façon suivante :

```
downheap t k n ≝
  while 2×!k + 1 ≤ n do
    let j = < sélectionner le plus grand des deux fils > in
    if t[j] ≤ t[!k] then break;
    (swap t !k j);
    k := j
  done
```

À cette écriture utilisant la construction break — le code de R. Sedgewick utilise une construction goto de façon similaire — nous avons préféré une écriture récursive, tout aussi simple, sinon plus :

```
downheap t k n ≝
  if 2 × k + 1 ≤ n then
    let j = < sélectionner le plus grand des deux fils > in
    if t[k] < t[j] then begin
      (swap t k j);
      (downheap t j n)
    end
```

Pour spécifier cette procédure récursive il semble naturel d'imposer comme pré-condition que les deux arbres situés aux indices $2k+1$ et $2k+2$ soient des tas (lorsqu'ils existent), et comme post-condition que l'arbre situé à l'indice k soit un tas. Malheureusement, une telle spécification n'est pas assez forte pour établir la correction du code ci-dessus. En effet, à l'issue de l'appel récursif $(\mathit{downheap}\ t\ j\ n)$, on est dans l'incapacité d'établir la post-condition car on ne dispose pas d'assez d'informations sur ce que « la procédure ne fait pas », en particulier sur les invariants qu'elle préserve au sein du tableau t.

Une première idée consiste à affaiblir la spécification de *downheap* en supposant comme pré-condition que l'on a $(\mathit{heap}\ t\ n\ i)$ pour tout i tel que $k+1 \leq i \leq n$, et en imposant comme post-condition la même propriété pour tout i tel que $k \leq i \leq n$. Cela ne suffit toujours pas et il faut ajouter à la post-condition le fait que la procédure *downheap* ne modifie pas certains éléments de t (à savoir ceux d'indices i tels que $i < k$, $k < i < 2k+1$ ou $n < i$), et également le fait qu'elle n'augmente pas la valeur maximale du tas situé à l'indice k, ce que l'on exprime de la manière suivante :

$$\forall v.\ (\mathit{inftree}\ t_@\ n\ v\ k) \Rightarrow (\mathit{inftree}\ t\ n\ v\ k)$$

On ajoute enfin à la post-condition de la procédure *downheap* le fait qu'elle n'agit sur le tableau t que par permutations d'éléments. On obtient finalement le code annoté donné Programme 11. La prédicat *select_son* qui apparaît comme annotation du calcul de j' n'est

Programme 11 La procédure récursive *downheap*

$$
\begin{array}{l}
\mathsf{let\ rec}\ \mathit{downheap}\ (N : Z)(t : \mathsf{array}\ N\ \mathsf{of}\ Z)(k, n : Z)\ :\ \mathit{unit}\ \{\ \mathsf{variant}\ n-k\ \}\ = \\
\quad \{\ 0 \leq k \leq n\ \wedge\ n < N \\
\quad\ \ \wedge\ \forall i : Z.\ k+1 \leq i \leq n\ \Rightarrow\ (\mathit{heap}\ t\ n\ i)\ \ \} \\
\quad (\mathsf{let}\ j\ =\ 2 \times k + 1\ \mathsf{in} \\
\quad\ \ \mathsf{if}\ j \leq n\ \mathsf{then} \\
\quad\quad \mathsf{let}\ j'\ =\ (\mathsf{if}\ j+1 \leq n\ \mathsf{then\ if}\ t[j] < t[j+1]\ \mathsf{then}\ j+1\ \mathsf{else}\ j\ \mathsf{else}\ j) \\
\quad\quad\quad\quad\quad \{\ (\mathit{select\_son}\ t\ k\ n\ \mathit{result})\ \}\ \mathsf{in} \\
\quad\quad \mathsf{if}\ t[k] < t[j']\ \mathsf{then\ begin}\ (\mathit{swap}\ N\ t\ k\ j');\ (\mathit{downheap}\ N\ t\ j'\ n)\ \mathsf{end}) \\
\quad \{\ (\mathit{permut}\ t\ t_@) \\
\quad\ \ \wedge\ (\forall i : Z.\ k \leq i \leq n\ \Rightarrow\ (\mathit{heap}\ t\ n\ i)) \\
\quad\ \ \wedge\ (\forall i : Z.\ (0 \leq i < k\ \vee\ k < i < 2 \times k + 1\ \vee\ n < i < N)\ \Rightarrow\ t[i] = t_@[i]) \\
\quad\ \ \wedge\ (\forall v : Z.\ (\mathit{inftree}\ t_@\ n\ v\ k)\ \Rightarrow\ (\mathit{inftree}\ t\ n\ v\ k))\ \ \}
\end{array}
$$

utlisé qu'à des fins de simplification. Il exprime que j' est le plus grand des deux fils de k, en prenant en compte la possibilité d'un unique fils (lorsque $n = 2k+1$). Ce prédicat est défini comme suit :

$$
\begin{array}{rcl}
(\mathit{select\_son}\ t\ k\ n\ j) & \stackrel{\text{def}}{=} & (j = 2k+1 \wedge (2k+2 \leq n \Rightarrow t[j] \geq t[2k+2])) \\
& \vee & (j = 2k+2 \wedge j \leq n \wedge t[j] \geq t[2k+1])
\end{array}
$$

La preuve de correction de la procédure *downheap* est contenue dans le fichier `Downheap.v`.

4.4.3 Le programme de tri *heapsort*

Le programme *heapsort* proprement dit est plus aisé à spécifier et à prouver que la procédure *downheap*, quoique plus long en apparence. Il est constitué de deux parties, la première construisant le tas à l'intérieur du tableau, et la seconde le détruisant progressivement en en retirant à chaque fois le plus grand élément.

La première partie s'écrit de façon très élégante et concise en utilisant la procédure *downheap*. Elle consiste à construire le tas de bas en haut, en remarquant que les éléments d'indices compris entre $(N-2)/2+1$ et $N-1$ forment déjà des tas réduits à un unique élément. On obtient alors le code suivant :

$$\mathsf{for}\ k = (N-2)/2\ \mathsf{downto}\ 0\ \mathsf{do}\ (\mathit{downheap}\ t\ k\ (N-1))\ \mathsf{done}$$

Ce code s'annote facilement en exprimant que l'on a $(\mathit{heap}\ t\ n\ i)$ pour tout i tel que $k+1 \leq i \leq N-1$. À l'issue de cette boucle, on a en particulier $(\mathit{heap}\ t\ 0\ (N-1))$, qui est la seule propriété qui nous intéresse — elle signifie que le tableau t tout entier forme un tas. Bien entendu, on aura ajouté également la propriété que le tableau reste une permutation du tableau initial.

La deuxième partie consiste alors constuire le tableau trié en commençant par les éléments les plus grands, selon le principe illustré page 138. Pour k variant de $N-2$ à 1, on échange $t[0]$, qui est le plus grand élément du tas $t[0..k]$, avec $t[k]$, qui est la place définitive de cet élément. Puis on appele $(\mathit{downheap}\ t\ 0\ (k-1))$ pour rétablir une structure de tas sur le segment $t[0..k-1]$.

L'invariant exprime que l'on a une structure de tas sur $t[0..k]$, que le segment $t[k+1..N-1]$ est trié et enfin que les éléments du tas $t[0..k]$ sont tous plus petits que les éléments du segment $t[k+1..N-1]$, ce qui s'exprime simplement par $t[0] \leq t[k+1]$. Comme toujours, on exprime également dans cet invariant le fait que t reste une permutation du tableau initial. Lorsque l'on sort de la boucle — avec donc $k=0$ — cet invariant implique trivialement la post-condition, à savoir que le tableau t est trié $(\mathit{sorted\_array}\ t\ 0\ (N-1))$ et qu'il est une permutation du tableau initial $(\mathit{permut}\ t\ t_{@})$.

On obtient finalement le code donné Programme 12. Sa preuve de correction est contenue dans le fichier `Heapsort.v`.

4.5 Autres études de cas

Aux études de cas que je viens de présenter, pour les avoir réalisées moi-même, il faut ajouter les développements suivants.

Tri par insertion. Une preuve d'un programme de tri par insertion a été réalisée en juin-juillet 1998 par Nicolas Magaud, élève de première année de magistère à l'École Normale Supérieure de Lyon. Cette preuve a été menée en six semaines, sans connaissance préalable du système Coq, et avec une version de mon implantation laissant encore à désirer.

Programme 12 Le programme *heapsort*

```
fun (N : Z)(t : array N of Z) →
  { 1 ≤ N }
  begin
    let k = ref (Zdiv2 (N − 2)) in
    while !k ≥ 0 do
      { invariant − 1 ≤ k ≤ N − 1
              ∧ (∀i : Z. k + 1 ≤ i ≤ N − 1 ⇒ (heap t (N − 1) i))
              ∧ (permut t t@0)
        variant k + 1 }
      (downheap N t !k (N − 1));
      k := !k − 1
    done;
    assert { (heap t (N − 1) 0) ∧ (permut t t@0) };
    let k = ref (N − 1) in
    while !k ≥ 1 do
      { invariant 0 ≤ k ≤ N − 1
              ∧ (∀i : Z. 0 ≤ i ≤ k ⇒ (heap t k i))
              ∧ (k + 1 ≤ N − 1 ⇒ (access t 0) ≤ (access t (k + 1)))
              ∧ (k + 1 ≤ N − 1 ⇒ (sorted_array t (k + 1) (N − 1)))
              ∧ (permut t t@0)
        variant k }
      (swap N t 0 !k);
      (downheap N t 0 (!k − 1));
      k := !k − 1
    done
  end
  { (sorted_array t 0 (N − 1)) ∧ (permut t t@) }
```

Algorithme de Floyd. Christine Paulin a réalisé une preuve de l'algorithme de Floyd (recherche de tous les chemins minimaux dans un graphe orienté pondéré). On pourra trouver une description de cet algorithme dans de nombreux livres, dont celui de R. Sedgewick [54], pages 476–478.

Protocole du bit alterné. Eduardo Giménez a réalisé une preuve d'un algorithme d'exclusion mutuelle (protocole du bit alterné). Cette preuve est décrite dans un article à paraître dans les actes de TPHOL'99 [15].

Ces développements sont également librement disponibles à l'adresse `http://coq.inria.fr/contrib/`.

4.6 Bilan et comparaisons

4.6.1 Bilan

Quelques données numériques concernant les études de cas que je viens de présenter sont rassemblées dans le tableau donné Figure 4.2. On y trouve les chiffres significatifs de chaque développement, et en particulier les proportions de programme, de spécification et de preuve. Les temps de compilation indiqués dans la dernière ligne sont bien sûr très relatifs[3] mais ils montrent tout de même que la vérification d'une preuve de programme est une opération rapide.

	Find	**Quicksort**	**KMP**	**Heapsort**
Spécification				
lignes de spécification	29	13	17	23
lignes de code (nb fonctions)	27 (1)	41 (4)	24 (2)	19 (2)
nb d'annotations	9	18	8	7
Preuve				
obligations de preuve	22	26	23	22
lemmes (introduits manuellement)	39	11	11	28
lignes de preuves	619	468	377	626
temps total	—	2 jours	1,5 jours	3 jours
temps de compilation	9 mn 02 s	8 mn 06 s	5 mn 45 s	12 mn 07 s

FIG. 4.2: Données numériques sur les études de cas

Il convient de s'attarder un instant sur ces chiffres. Tout d'abord, on s'aperçoit que spécifier des algorithmes même complexes n'est pas une opération si difficile : quelques lignes de définitions suffisent à chaque fois et les annotations à insérer dans les programmes ne sont pas très nombreuses. Mis à part dans le cas de *heapsort*, il y a moins de lignes d'annotations que de lignes de programmes, chose qui n'était pas évidente *a priori*.

[3]Les compilations ont été effectuées sur un Pentium 200 Mhz, 64 Mo de RAM, tournant sous Linux.

Concernant la preuve elle-même, on s'aperçoit que le nombre d'obligations de preuve est lui aussi tout à fait raisonnable. Il ne dépasse pas non plus le nombre de lignes de programme. On ne peut bien sûr statuer sur leur difficulté, qui dépend du programme *et de sa spécification*, mais le nombre de lignes de preuve dans Coq en donnera une idée à l'initié. Le temps total de travail (spécification, codage et preuve) est peut-être la meilleure indication sur l'efficacité de la méthode : deux jours pour établir la correction totale d'un algorithme comme *Quicksort* ou *Knuth-Morris-Pratt* nous parait tout à fait raisonnable.

Ces quelques exemples, pourtant assez académiques, ont confirmé la nécessité de considérer un langage réaliste lorsque l'on cherche à faire de la preuve de programme. En effet, au delà des aspects impératifs traditionnellement traités — affectation, conditionnelle, séquence et boucle — nous avons utilisé toutes les autres constructions introduites :

- variables et références locales à travers la construction let in (*swap, pivot, kmp, initnext, downheap* et *heapsort*) ;
- procédures et fonctions, avec appel par valeur et par variable (dans toutes les fonctions de *Quicksort*, de *KMP* et de *Heapsort*) ;
- fonction récursive (*quick_rec* et *downheap*).

D'autre part, nous avons utilisé toutes les possibilités offertes en matière de spécification :

- utilisation des étiquettes pour désigner des valeurs antérieures des références ou des tableaux (*find* et *pivot*) ;
- utilisation d'assertions intermédiaires à travers la commande assert (*find* et *heapsort*) ;
- possibilité de faire porter la post-condition sur le résultat de l'évaluation (*quick_rec*, *pivot* et *kmp*) ;
- utilisation d'un variant autre qu'un entier (*kmp* et *initnext*).

De manière générale, notre méthode et son implantation ont prouvé leur *viabilité*, sur le terrain même de la logique de Floyd-Hoare — les exemples traités jusqu'à présent sont des manipulations essentiellement impératives de tableaux. Mais ces exemples restent très académiques, et il conviendrait de s'attaquer aussi à des problèmes plus proches des réalités industrielles. D'autre part, les exemple traités ne font pas usage d'ordre supérieur, et notre méthode n'a toujours pas subi de mise à l'épreuve dans ce domaine.

D'autre part, il faut relativiser ce bilan en admettant que les études de cas ont mis en avant la *difficulté d'annotation* des programmes. Par là, nous entendons non pas la difficulté de trouver les pré- et post-conditions et invariants de boucle, indépendante d'une méthodologie particulière, mais plutôt la difficulté à atteindre la *complétude* dans les annotations. En effet, ainsi que nous l'avons détaillé dans la section 2.4, une annotation manquant, par exemple sur une conditionnelle, peut amener à des obligations de preuve non prouvables alors qu'*a priori* toute la spécification nécessaire avait été donnée dans le programme. En fait, c'est la difficulté à saisir la dichotomie entre *substitution* et *abstraction* qui apparaît comme la plus gênante. Même une fois que l'on a saisi cette notion, on reste parfois surpris par certaines obligations de preuve. Les nombreuses propagations automatiques d'annotations dont nous avons parlé section 2.4 ont pour but d'éliminer cette difficulté.

4.6.2 Comparaisons

Nous avions brièvement présenté quelques méthodes formelles existantes dans la section 2.1. Après les études de cas que nous venons d'exposer, nous sommes en mesure de comparer notre méthode aux plus représentatifs de ces systèmes. En l'occurrence, nous avons choisi la méthode VDM, la méthode B et le système KIV, pour leur diversité mais également pour leur maturité, du moins en ce qui concerne les deux premiers. La méthode VDM, pour *Vienna Development Method* [27, 28][4], est la plus ancienne, directement inspirée de la logique de Hoare. La méthode B de J. R. Abrial [4] est plus récente mais a déjà fait ses preuves dans les milieux industriels, notamment en certifiant la partie logicielle de la ligne de métro automatique parisienne METEOR. Enfin, le système KIV, pour *Karlsruhe Interactive Verifier* [51], est le plus récent mais il possède déjà une sérieuse collection d'études de cas [14] et affiche nettement l'ambition d'être utilisé à l'échelle industrielle.

Nous allons comparer ces trois méthodes à celle que nous avons proposée — et non entre elles, ce qui n'est pas notre propos — sur les trois plans suivants : la manière de spécifier les programmes, les constructions offertes pour écrire et annoter ces programmes, et enfin les moyens mis à disposition pour en établir la correction.

Spécifications. La manière de spécifier les programmes est certainement le point sur lequel les méthodes que nous comparons diffèrent le plus. On peut résumer très schématiquement les différentes approches par le tableau suivant :

VDM	méthode B	KIV	notre méthode
théorie des ensembles pré/post conditions	théorie des ensembles machines abstraites	spécifications algébriques	théorie des types pré/post conditions

Les méthodes VDM et B reposent toutes deux sur une axiomatisation de la théorie des ensembles du premier ordre. Ce sont des formalisations très lourdes, avec un grand nombre d'axiomes et/ou de très nombreuses définitions, dont l'expressivité est difficile à cerner. Au delà de ces aspects théoriques, le choix de la théorie des ensembles a également des impacts pratiques, tels que la difficulté de définir de nouveaux types de données — ceux-ci doivent l'être par des codages ensemblistes — ou encore le fait d'impliquer des obligations de preuve de la forme $n \in NAT$. Dans notre cas, le choix du Calcul des Constructions Inductives comme langage de spécification apporte une solution satisfaisante autant du point de vue théorique que pratique à ces divers problèmes. En particulier, nous n'obtenons pas d'obligations de preuve de la forme $n \in NAT$, ces propriétés étant des conséquences immédiates du typage.

Du point de vue de la spécification des programmes, vue indépendamment du formalisme logique sous-jacent, notre approche est très semblable à celle de VDM, et toutes deux sont directement inspirées par la logique de Floyd-Hoare. Les fonctions sont spécifiées à l'aide d'une pré- et d'une post-condition portant sur les arguments et les variables du programme. Dans la déclaration d'une opération implicite en VDM — c'est-à-dire dont on donne uniquement la spécification mais pas le code — les effets doivent être explicitement déclarés en termes de variables lues ou modifiées, exactement de la même manière qu'ils apparaissent

[4] Le livre de C. B. Jones [28] a été traduit en français, sous le titre *VDM : une méthode rigoureuse pour le développement du logiciel* [29].

dans notre type de calcul κ. Ainsi un programme de tri est spécifié dans VDM de la manière suivante :

$$\begin{array}{l} SORT \\ \quad \mathsf{wr}\ t : seq(T) \\ \mathsf{pre\ true} \\ \mathsf{post}\ sorted(t) \land permut(t, \overleftarrow{t}) \end{array}$$

et dans notre cas par la déclaration suivante :

$$sort \quad : \quad \Big(\mathsf{unit}, (\{t\}, \{t\}), \mathsf{True}, sorted(t) \land permut(t, \overleftarrow{t})\Big)$$

On peut d'ailleurs constater que notre syntaxe concrète pour les spécifications, donnée section 3.1, est très proche de la syntaxe de VDM.

Dans la méthode B, en revanche, les opérations d'une machine abstraite sont spécifiées à l'aide d'une substitution généralisée, que l'on peut voir comme une extension de la notion de commande gardée de Dijkstra [11]. Une telle substitution est généralement non déterministe. Ainsi, un programme de tri pourra être spécifié de la manière suivante :

$$\begin{array}{l} sort = \\ \textsc{any}\ t'\ \textsc{where} \\ \quad sorted(t) \land permut(t, t') \\ \textsc{then} \\ \quad t\ :=\ t' \\ \textsc{end} \end{array}$$

J. R. Abrial a montré que l'utilisation des substitutions généralisées est équivalente à l'utilisation de pré- et post-conditions (voir [4] pages 292–295). Il argumente alors en faveur des substitutions généralisées en expliquant que le recours à des prédicats avant-après oblige à déclarer explicitement qu'une variable n'est pas modifiée, alors que l'invariance d'une variable est implicite dans une substitution généralisée. C'est un argument valable pour la logique de Floyd-Hoare traditionnelle, ou le programme est vu comme un transformateur d'états globaux, mais qui ne s'applique pas vraiment à VDM, et encore moins à notre méthode. En effet, la déclaration implicite d'une fonction nécessite que l'on en déclare les effets, et l'on ne peut clairement pas faire moins. Une substitution généralisée le fait d'ailleurs tout autant qu'une déclaration dans VDM ou dans notre cas, par l'intermédiaire des variables qu'elle manipule. De plus, la déclaration explicite d'une fonction s'accompagne chez nous d'une inférence statique des effets, qui dispense donc l'utilisateur de leurs déclarations.

Dans le système KIV, une spécification est donnée sous la forme d'une *spécification algébrique*, c'est-à-dire d'une signature, composée de déclarations de sortes et d'arités de fonctions, et d'un ensemble d'axiomes sous forme équationnelle. Les objets y sont considérés comme purement fonctionnels. Une implantation, quant à elle, est la donnée d'un ensemble de déclarations de types et de procédures, dans une syntaxe proche du Pascal. Un module est alors la donnée d'une spécification, d'une implantation et d'une correspondance explicite entre les objets de l'implantation et ceux de la spécification. La définition d'un module engendre un certain nombre d'obligations de preuve, visant à établir la correction du module, obtenus par traduction des programmes à l'aide de la *logique dynamique* — voir les références données dans [51]. D'un certain point de vue, il y a une similitude entre l'interface purement fonctionnelle que constitue une spécification dans le système KIV, et l'interprétation fonctionnelle que nous faisons des programmes. Cependant, la correspondance s'arrête là, notamment parce que les spécifications algébriques induisent une manière de spécifier les programmes complètement différente de l'utilisation des pré- et post-conditions. Les spécifications algébriques sont d'ailleurs réputées complexes à utiliser. On pourra lire l'intéressante discussion de C. B. Jones à ce sujet dans [29], pages 194–199.

Du simple point de vue de la spécification, sans parler de système logique ni de preuve, mon expérience personnelle de programmeur m'incite à penser que les pré- et post-conditions constituent la manière la plus naturelle de spécifier, que ce soit des programmes fonctionnels ou impératifs. Le choix du Calcul des Constructions Inductives s'explique quant à lui par son expressivité et sa simplicité mathématique, qui l'emportent de ces points de vue-là sur la théorie des ensembles. On pourra noter, par exemple, avec quelle facilité nous avons pu définir le prédicat *heap* dans la preuve de *heapsort* décrite page 136, et raisonner par induction sur ce prédicat pour établir la plupart des lemmes impliqués dans cette preuve de correction.

Programmes. Dans les méthodes VDM, B ou KIV, on trouve les constructions de base de la programmation impérative, à savoir l'introduction de variable locale, l'affectation, la séquence, la conditionnelle et la boucle, et rien de plus. Les procédures, fonctions et fonctions récursives sont des constructions particulières, qui ne peuvent apparaître qu'au niveau le plus haut. Cette relative pauvreté du langage de programmation est justifiée par des critères de sécurité et de non-ambiguïté sémantique lors de la production de code.

Notre culture de programmeur ML nous a amené à considérer naturellement un ensemble plus vaste de constructions de programmes. Les variables locales, par exemple, sont introduites par la construction let in, et elles peuvent être aussi bien fonctionnelles, comme dans let $x = 2 \times y$ in e, qu'impératives, comme dans let $x =$ ref $(2 \times y)$ in e. On remarquera qu'elles sont nécessairement initialisées, ce qui impose une rigueur de développement supplémentaire — ce qui n'est pas le cas des variables locales dans la méthode B. Les références sont réellement des objets de première classe, qui peuvent être créées dynamiquement, et non pas seulement déclarées statiquement. En particulier, une nouvelle référence peut être retournée comme résultat d'une fonction.

Nous considérons également les fonctions comme des objets de première classe, et c'est là un point qui nous est propre, les méthodes VDM, B et KIV étant définitivement des formalismes du premier ordre. En particulier, nous avons introduit un système de spécification d'ordre supérieur, montrant par là comment il est possible de raisonner à propos des fonctions en tant qu'arguments ou résultats d'autres fonctions. Disposer des fonctions comme objets de première classe permet en particulier de définir des fonctions locales, une construction permettant une meilleure encapsulation dans l'écriture du code. On pourra remarquer par exemple avec quelle facilité nous avons pu introduire une construction for en utilisant une fonction récursive locale (voir page 119). Notre méthode inclut également une forme de polymorphisme, certes restreinte mais que l'on ne retrouve pas dans les autres systèmes.

Enfin, et peut-être surtout, nous nous distinguons nettement des systèmes existants en ce qui concerne les types de données, en permettant à l'utilisateur de manipuler n'importe quel type de données défini dans le Calcul des Constructions Inductives[5]. La conséquence

[5] À l'extraction, ces types seront soit des types récursifs primitifs si le langage cible en dispose, soit codés

la plus importante est que nous avons su définir une méthode de preuve de programmes indépendamment des objets de base du langage, sans avoir à introduire ni axiomatiser de nombreuses constructions primitives — constructions ensemblistes dans le cas de la méthode B, objets composites dans le cas de VDM, etc.

Preuves. Le prouveur qui soutient la méthode est un élément déterminant de son succès. En ce qui concerne VDM, les tentatives pour écrire un assistant de preuve, MURAL, se sont globalement soldées par un échec, et l'on doit plutôt considérer VDM comme une méthode que comme un réel outil de validation formelle. Cela s'explique peut-être par la relative complexité d'une formalisation de la théorie des ensembles, qui comporte nécessairement un grand nombre d'axiomes et/ou de définitions. La méthode B, qui repose sur un formalisme similaire, s'en est pourtant bien tirée, et les résultats de l'atelier B sont mêmes impressionnants, dépassant les 80% d'obligations de preuve automatiquement déchargées. Il n'en reste pas moins que la correction d'un tel prouveur est difficile à garantir[6]. Le système KIV ne décrit absolument pas son prouveur, mais se contente d'en donner des résultats statistiques, d'ailleurs assez impressionnants également [14]. Ce prouveur repose sur un puissant noyau de réécriture, les spécifications étant données sous forme équationnelle. On constate que les méthodes B et KIV s'efforcent toutes deux de fournir des prouveurs automatiques puissants, avec le souci de ne laisser à la charge de l'utilisateur qu'un nombre minimum d'obligations de preuve. Elles fournissent néanmoins chacune un prouveur interactif, permettant à l'utilisateur d'établir les obligations restantes ou de prouver la validité de lemmes supplémentaires.

En ce qui nous concerne, notre méthode est bâtie sur l'assistant de preuve Coq [1]. Nous n'avons développé aucune tactique particulière pour aider l'utilisateur dans la résolution des obligations de preuve, et celui-ci ne dispose donc que des tactiques fournies en standard par le système. Celles-ci comportent peu de tactiques automatiques, et ces dernières sont peu puissantes, exception faite toutefois de la tactique Omega résolvant les égalités et inégalités dans l'arithmétique de Presburger. Il en résulte que les preuves sont essentiellement interactives. Mais à la différence des outils B et KIV, le système Coq dispose d'un excellent prouveur interactif, et les obligations de preuve engendrées étant concises et naturelles, les temps de développement sont tout à fait raisonnables, ainsi que l'illustrent les chiffres donnés page 143. D'autre part, le système Coq repose sur un très petit nombre de règles logiques, et offre par conséquent un système formel sûr.

Ainsi que nous l'avons déjà expliqué, l'un des avantages majeurs de notre approche est de se reposer sur un formalisme logique et un prouveur existants, en dissociant par là des problèmes de natures différentes et en laissant à d'autres, plus compétents que nous, le soin d'étendre la logique et d'améliorer le prouveur.

Nous n'avons comparé ici que les aspects pour lesquels notre méthode propose d'ores et déjà une solution. Il faut bien entendu reconnaître que les méthodes décrites ci-dessus offrent également d'autres fonctionnalités, dont nous n'avons pas parlé. En particulier, les méthodes B et KIV apportent toutes deux des mécanismes de développement modulaire, permettant

à l'aide de pointeurs et manipulés avec l'assistance d'une bibliothèque de GC dans les autres cas.

[6] Il a d'ailleurs été montré de plusieurs façons que le prouveur de l'atelier B permettait facilement d'établir une preuve de l'absurde, et il a été montré par ailleurs que dans une version précédente de cet outil plus d'une centaine de règles utilisées par le prouveur n'étaient pas valides [3].

d'écrire des spécifications de modules et de prouver la correction de modules de manière séparée. La méthode B offre de surcroît une notion de raffinement — d'une implantation ou d'une spécification en une implantation. Ces deux aspects ne font pour l'instant partie que des perspectives de notre travail, que nous allons exposer dans le chapitre de conclusion qui va suivre.

Conclusion et perspectives

Nous voilà parvenus au terme de ce travail, et il est temps d'en tirer les conclusions. D'un point de vue général, nous pouvons présenter notre contribution comme une méthode formelle de preuve de programmes impératifs reposant sur le Calcul des Constructions Inductives comme langage de spécification et le système Coq comme assistant de preuve. Notre travail est dès à présent distribué avec le système Coq, et nous l'avons mis en pratique sur quelques algorithmes non triviaux, avec d'excellents résultats. Ce fut d'un point de vue personnel la plus belle des récompenses.

Notre méthodologie repose en premier lieu sur une interprétation fonctionnelle des programmes impératifs. Pour définir cette traduction, nous avons combiné une analyse statique des effets et l'utilisation de monades, généralisées pour l'occasion. Si ces deux aspects ont été séparément largement étudiés, leur combinaison semble être originale. Notre traduction a pour principal avantage d'être proche de ce que l'on ferait « à la main », ce qui conduira à des obligations de preuve très naturelles dans la deuxième partie de notre travail. En contre-partie, elle souffre de certaines limitations en ce qui concerne les traits de programmation fonctionnels, comme nous l'avons montré à la fin du chapitre 1, pages 54–58, mais cet aspect n'est pas primordial dans le cadre des utilisations actuelles des méthodes formelles.

Dans un deuxième temps, nous avons proposé une méthode pour établir la correction totale de programmes, dans le cadre général d'un λ-calcul typé d'ordre supérieur. Les programmes sont spécifiés à l'aide de pré- et post-conditions, dans la tradition de la logique de Floyd-Hoare. La formule de correction $\{P\}\ e\ \{Q\}$ est alors interprétée comme la proposition « la traduction fonctionnelle de e réalise la proposition $\forall x.\ P \Rightarrow \exists y.\ Q$ », au sens de la réalisabilité fonctionnelle. Pour établir une telle proposition, on construit un terme de preuve partiel de la proposition $\forall x.\ P \Rightarrow \exists y.\ Q$, bâti sur la traduction fonctionnelle de e. Ce terme de preuve est incomplet en ce qui concerne ses parties logiques, et les parts manquantes constituent les obligations de preuve. Nous avons montré la correction de cette méthode, qui repose sur le théorème de réalisabilité, et également sa complétude relative. Si une telle approche permet de retrouver les obligations de preuve de la logique de Floyd-Hoare dans les cas les plus simples, elle a surtout l'avantage de traiter sans difficulté des constructions souvent bien plus délicates, telles que les effets de bords dans les expressions ou les fonctions récursives.

Enfin, nous avons implanté notre méthode dans le système Coq, et l'avons testée sur des programmes non triviaux. Cet aspect pratique était essentiel pour valider notre méthode, et il a permis en outre de dégager un certain nombre de problèmes techniques qui ne ressortaient pas de la simple étude théorique, tel que la nécessité d'insérer automatiquement certaines annotations pour n'impliquer l'utilisateur que dans une spécification minimale. Nous avons dégagé un grand nombre d'avantages de notre méthode par rapport aux méthodes formelles existantes, tant du point de vue de la spécification que de celui des programmes et des preuves, et cela est notamment dû au fait d'avoir impliqué dans notre démarche des outils de spécification et de preuve existants, à savoir le Calcul des Constructions Inductives et le système Coq. Nous espérons maintenant que notre travail sera validé par une large communauté d'utilisateurs.

Perspectives

Il reste du chemin à parcourir pour parvenir à un environnement complet de preuve de programmes. À côté de certaines améliorations essentiellement techniques (étendre le langage de programmation, par exemple par du filtrage ; fournir une bibliothèque standard de fonctions spécifiées, en particulier autour de la manipulation des nombres à virgule flottante ; tactiques automatiques pour aider à la résolution des obligations de preuve ; etc.), nous entrevoyons des extensions plus profondes. Nous les décrivons ici en quelques lignes.

Exceptions. Les exceptions constituent une structure de programmation très puissante, qui apporte une solution pour l'instant sans égale à la gestion des erreurs. On trouve des exceptions aussi bien dans les langages impératifs, fonctionnels et orientés objets. C'est pour cela qu'il nous semble nécessaire d'intégrer à terme des exceptions dans notre langage. Nous avons montré dans les sections 1.2.2 et 1.3.1 comment les exceptions pouvaient être interprétées dans notre cadre de travail, en terme d'analyse statique des effets et de traduction fonctionnelle à l'aide de monades. Il reste cependant à définir comment *spécifier* un programme pouvant lever une exception. Une possibilité consiste à étendre la syntaxe d'annotation sous la forme

$$\{P\}\ e\ \{Q + R\}$$

avec la signification usuelle de $\{P\}\ e\ \{Q\}$ lorsque e retourne une valeur normalement, et l'idée que la proposition R est vérifiée dès lors que l'évaluation de e lève une exception — on se place ici dans le cas très simple d'une unique exception sans argument. On est alors en mesure d'associer une propriété à la levée d'une exception et d'utiliser plus tard cette information dans le cas où l'exception est effectivement levée. Ainsi, une fonction *map* de recherche d'une clé k dans une table t pourrait être spécifiée de la manière suivante :

$$\{\}\ (\mathit{map}\ t\ k)\ \{\lambda r.(\mathit{In}\ t\ k\ r) + \forall v.\neg(\mathit{In}\ t\ k\ v)\}$$

Un cas particulier d'usage des exceptions est celui où le programmeur se retrouve dans une situation qu'il sait par ailleurs inaccessible, généralement par contradiction avec une pré-condition, comme dans l'exemple suivant :

$$\begin{array}{l} \{n \neq 0\} \\ \quad \mathsf{match}\ n\ \mathsf{with} \\ \quad\quad 0 \Rightarrow [\ \textit{code inaccessible}\ ] \\ \quad \mid (S\ p) \Rightarrow p \\ \{r \mid r < n\} \end{array}$$

L'usage est en effet de déclencher une erreur à cet endroit, donc généralement de lever une exception, dont on sait qu'elle ne sera en fait jamais levée. Dès lors, un tel programme pourrait se voir attribuer une post-condition de la forme $\{Q + \mathsf{False}\}$ exprimant que la levée de l'exception est impossible. Mais il n'en reste pas moins que tout programme au dessus de celui-là déclarera dans ses effets pouvoir potentiellement lever cette exception, qu'il faudra nécessairement prendre en compte dans les obligations de preuve, même s'il s'agira à chaque fois d'une obligation de preuve triviale. Une bien meilleure idée consisterait à introduire un mot clé never que l'on interpréterait comme du code inaccessible et qui serait traduit dans le terme de preuve en $(\mathit{False\_rec}\ \overline{\tau}\ ?)$ où τ est le type attendu et l'obligation de preuve celle d'une preuve de False. On aurait bien alors à démontrer l'absurdité de cette situation, à travers cette preuve de l'absurde, mais le programme conserverait son type initial, contrairement à l'utilisation d'une exception. Rien n'empêche ensuite de traduire cette construction en une exception lors de la production de code pour un langage particulier.

Modularité. La modularité est un aspect essentiel de la programmation lorsque les développements deviennent importants. Elle permet entre autres à plusieurs développeurs de collaborer aisément, avec pour chacun une connaissance minimale du travail des autres, et de modifier aisément des parties du code avec des répercutions minimales sur le reste du code. La plupart des langages de programmation modernes fournissent une possibilité de programmation modulaire, que se soit à travers un vrai système de modules ou à travers des classes. Une méthode de preuve de programmes se doit donc d'offrir la même possibilité de développement modulaire. C'est d'ailleurs le cas pour la méthode B et le système KIV.

En ce qui nous concerne, nous disposons des éléments nécessaires à un développement modulaire de preuve de programmes. En effet, ainsi que nous l'avons expliqué dans la section 1.4 page 1.4, la modularité repose sur l'unique propriété suivante : si la preuve de correction d'un programme e est faite dans un certain contexte Γ contenant $x : \tau$, alors cette preuve ne dépend que de τ. Cette propriété est immédiate dans notre cas. Cependant, cela ne nous donne pas pour autant un système de modules pour notre méthodologie. Pour cela, il faut adopter un système de module particulier pour notre langage de programmation et surtout comprendre comment l'étendre pour y ajouter des informations logiques. Il faut tout d'abord donner pour chaque fonction son type *annoté*, c'est-à-dire non seulement son type au sens usuel, mais également ses effets et sa spécification. Mais cela ne suffit toujours pas. En effet, spécifier les fonctions d'un module nécessite également d'exporter avec ces fonctions les définitions permettant de les spécifier. Imaginons par exemple que l'on spécifie un module définissant une table d'association impérative, sous la forme d'un type abstrait t. Alors il faut exporter également un prédicat indiquant si une association appartient ou non à une table donnée, car dans le cas contraire nous serions dans l'impossibilité de spécifier les fonctions opérant sur cette table. Une fois un tel prédicat exporté — appelons-le *In* — alors il est possible de spécifier une fonction telle que celle qui crée une table vide, par exemple de la manière suivante :

$$(\mathit{empty}\ x)\ \{r \mid \forall k.\forall v.\neg(\mathit{In}\ r\ k\ v)\}$$

On remarquera qu'il n'est nul besoin de savoir comment le type t est construit, ni comment le prédicat *In* est défini. L'encapsulation est respectée et permet à tout moment de changer l'implantation de la table d'association. Enfin, il faut pouvoir déclarer également des axiomes dans une interface, tel que par exemple un axiome affirmant qu'une table ne contient au plus qu'une association pour une même clé.

En fin de compte, une interface de module devra comprendre des déclarations au sens usuel, agrémentées d'informations d'effets et de spécification, ainsi que des déclarations d'objets logiques, qui peuvent être des définitions, des axiomes, etc. Voici par exemple à quoi pourrait ressembler l'interface d'un module implantant une table d'association impérative :

```
interface Map
  type t
  predicate In : t → key → info → Prop
        < axiomes sur In >
  val empty : unit → t
        < effets et spécification de empty >
  val add : t → key → info → unit
        < effets et spécification de add >
  val map : t → key → info
        < effets et spécification de map >
end
```

où la fonction *empty* pourrait être spécifié de la manière suivante :

Raffinement. Les méthodes B et KIV proposent la fonctionnalité de *raffinement*, qui permet de raffiner une spécification en une implantation — éventuellement partielle et non-déterministe — ou une implantation en une autre implantation. C'est une très vieille idée, que l'on trouve déjà dans la preuve de *find* de C. A. R. Hoare [25], et qui consiste à construire un programme « de haut en bas » en partant de sa spécification [5, 41]. M. Gordon montre dans son cours d'introduction à la preuve de programmes en logique de Floyd-Hoare [18] comment le raffinement peut être vu de façon très syntaxique comme un ensemble de règles de réécriture sur des formules de la forme $[P, Q]$ dont l'interprétation est « l'ensemble des programmes e tels que $\{P\}\ e\ \{Q\}$ ». Dans notre cas, l'information à donner est le type de calcul annoté κ, interprété comme « n'importe quel programme e de type κ ». Avec un tel formalisme, il est en particulier possible de décrire la structure la plus externe d'un programme sans en donner les parties internes, et d'obtenir tout de suite des obligations de preuve. Ainsi je peux écrire

```
{P}
  while b do
    {invariant I}
    [c : κ]
  done
{Q}
```

où P, b, I et Q sont donnés, mais pas le corps de la boucle c, dont on donne uniquement le type de calcul κ, de la forme $(r : \tau, \epsilon, P_c, Q_c)$. On peut imaginer dans le cas présent que P_c et Q_c prennent par défaut la valeur de I, exprimant ainsi la préservation de l'invariant par le corps de la boucle, mais ce n'est en rien une obligation. On obtient alors en particulier les obligations de preuve $P \Rightarrow I$ et $I \wedge b = \mathsf{false} \Rightarrow Q$, que l'on peut s'attacher à prouver sans plus tarder. On obtient également des obligations de la forme $I \wedge b = \mathsf{true} \Rightarrow P_c$ et $Q_c \Rightarrow I$ dans le cas où la spécification κ de c n'exprime pas directement la préservation de l'invariant. Ainsi on vérifie immédiatement la pertinence de l'invariant I, avant de chercher à en prouver la préservation. C'est là l'intérêt du raffinement, car il permet ainsi de trouver rapidement les erreurs dans la spécification et dans le code.

On remarquera que l'on peut déjà procéder par raffinement dans notre système, de manière « artisanale ». En effet, on peut déclarer un programme c de type κ et écrire la portion de code ci-dessus avec c comme corps de la boucle. Dès lors, on obtient bien les obligations de preuve recherchées, et une fois ces obligations de preuve établies, on peut remplacer c par un morceau de programme, qui peut contenir à son tour des variables de programme, et ainsi de suite. Bien entendu, un véritable mécanisme de raffinement doit proposer d'autres fonctionnalités, telles que la possibilité d'affaiblir une post-condition ou de renforcer une pré-condition. Mais on peut dès à présent constater que le raffinement est envisageable dans notre système, et il est intéressant de noter qu'il l'est *pour la même raison que la modularité est elle-même possible* : en effet, c'est le fait qu'une preuve de correction ne dépende que du *type* des sous-programmes qui rend possible le raffinement, tout comme la modularité, comme nous l'avons expliqué ci-dessus.

Avant d'être des aspects théoriques, la modularité et le raffinement sont surtout des aspects *méthodologiques*. La puissance de la méthode B ne réside pas dans celle des programmes que l'on peut y décrire, ou dans ses aspects méta-théoriques, mais dans la méthodologie de certification qu'elle propose, s'adaptant ainsi à un large public d'horizons divers. C'est dans cette optique qu'il nous faudra concevoir les développements futurs de notre méthode, car négliger les aspects méthodologiques, c'est vouer une recherche scientifique à ne jamais dépasser le stade de la recherche elle-même.

Rapprochement de la tactique Program. L'un des aspects les plus séduisants de la tactique Program de C. Parent [45, 46] est son exploitation complète de l'extraction. Cela permet en particulier de déclarer des types de données mélangeant parties informatives et logiques, de les manipuler dans les programmes uniquement à travers leurs parties informatives et d'obtenir à l'issue de la tactique des obligations de preuve portant sur les parties logiques. Ainsi, on peut définir le type *chiffre* des entiers compris entre 0 et 9, par exemple par la définition

$$\mathsf{Inductive}\ \mathit{chiffre} : \mathsf{Set} := \mathit{ch} : \forall z : Z.\ 0 \leq z \leq 9 \Rightarrow \mathit{chiffre}.$$

et le manipuler à l'intérieur des programmes sous la forme $(ch\ z)$, c'est-à-dire en ne manipulant que l'entier z. Si à un certain moment on écrit $(ch\ (z_1 + z_2))$ alors on obtiendra l'obligation de preuve $0 \leq z_1 + z_2 \leq 9$. De manière générale, cela permet de définir de types de données auxquels sont associées des propriétés, que l'on pourrait appeler des invariants, qui bien entendu disparaissent à l'extraction. Notre méthode n'exploite pas du tout l'extraction dans ce sens-là, mais uniquement sous une forme très pauvre, à savoir sur des propositions de la forme

$$\forall x : I.\ P(x) \Rightarrow \exists y : J.\ Q(x, y)$$

où I et J sont purement informatifs, et P et Q purement logiques. Un développement intéressant de notre méthode consisterait à voir les constantes dans un programme e comme appartenant au monde extrait et l'interprétation $\widehat{e}$ comme appartenant au monde logique, des obligations de preuve étant insérées maintenant également à l'intérieur des objets du premier ordre à la manière de ce que réalise la tactique Program.

Annexes

A Preuve des théorèmes 1 et 2

Nous donnons ici les preuves détaillées des théorèmes 1 (page 53) et 2 (page 54).

Théorème 1 (correction) *Soit* Γ *un environnement bien formé dont les références sont* $x_1 : \tau_1$ ref$, \ldots, x_n : \tau_n$ ref *et* e *un programme tel que* $\Gamma \vdash_r e : (\tau, (\rho, \omega))$*. Soient* v_i *des valeurs de types* τ_i *et* θ *l'état associant les* v_i *aux* x_i*. Alors*

$$\forall \theta', v. \quad \theta.e \longmapsto^\star \theta'.v \quad \Longrightarrow \quad (\overline{e}\ \overline{\theta(\rho)}) = (\overline{\theta'(\omega)}, \overline{val(\theta', v)})$$

Preuve. La preuve se fait par récurrence sur la longueur de la dérivation $\longmapsto^\star$ et par cas sur la structure de e.

- $e \equiv c \mid x \mid \mathsf{fun}\ x \to e_1 \mid \mathsf{Y}$: Dans ce cas, $\rho = \omega = \emptyset$, $v = e$ et $val(\theta', v) = v$ car e ne peut être une référence — y compris lorsque e est une variable, en vertu de la règle $(\mathsf{VAR'}_r)$. Dès lors la conclusion est immédiate.
- $e \equiv \mathsf{ref}\ e_1$: Avec E désignant le contexte d'évaluation ref [], l'évaluation de e est de la forme :

$$\begin{array}{rcll} \theta.E[e_1] & \longmapsto^\star & \theta_1.E[v_1] & \text{avec } \theta.e_1 \longmapsto^\star \theta_1.v_1 \\ & \longmapsto & \theta_1.\{(x, v_1)\}.x & (\mathit{ref}) \\ & \longmapsto & \underbrace{\theta_1 \uplus \{(x, v_1)\}}_{=\theta'}.\underbrace{x}_{=v} & (\theta_{merge}) \end{array}$$

  Comme $\theta.e_1 \longmapsto^\star \theta_1.v_1$, et que e_1 a pour effet (ρ, ω) par la règle (REF_r), on a par hypothèse de récurrence

$$(\overline{e_1}\ \overline{\theta(\rho)}) = (\overline{\theta_1(\omega)}, \overline{val(\theta_1, v_1)})$$

  et $val(\theta_1, v_1) = v_1$ car la règle (REF_r) précise que le type de v_1 est pur. De plus, $val(\theta', v) = val(\theta_1 \uplus \{(x, v_1)\}, x) = v_1$ donc

$$\begin{array}{rcl} (\overline{e}\ \overline{\theta(\rho)}) & = & (\overline{e_1}\ \overline{\theta(\rho)}) \\ & = & (\overline{\theta_1(\omega)}, \overline{v_1}) \\ & = & (\overline{\theta'(\omega)}, \overline{val(\theta', v)}) \end{array}$$

– $e \equiv\, !x$: alors $\overline{e} = \lambda u.\, u.x$ et $\epsilon = (\{x\}, \emptyset)$. L'évaluation de e se réduit à $\theta.!x \longmapsto^\star \theta.\theta(x)$ par (*deref*), donc $\theta' = \theta$ et $val(\theta', \theta(x)) = \theta(x)$ car x est une référence sur un type pur. Par conséquent

$$\begin{array}{rcl} (\overline{e}\ \overline{\theta(\rho)}) & = & (\overline{e}\ \overline{\theta(\{x\})}) \\ & = & (\lambda u.\, u.x\ \{x = \overline{\theta(x)}\}) \\ & = & \overline{\theta(x)} \end{array}$$

qui est identifié avec $(\overline{\{\}}, \overline{\theta(x)})$.

– $e \equiv x := e_1$: Avec E désignant le contexte d'évaluation $E = x := []$, l'évaluation de e est la suivante :

$$\begin{array}{rcll} \theta.E[e_1] & \longmapsto^\star & \theta_1.E[v_1] & \text{avec } \theta.e_1 \longmapsto^\star \theta_1.v_1 \\ & \longmapsto & \underbrace{\theta_1 \uplus \{(x, v_1)\}}_{=\theta'} . \underbrace{\mathsf{void}}_{=v} & (\textit{assign}) \end{array}$$

Comme $\theta.e_1 \longmapsto^\star \theta_1.v_1$, et que e_1 a pour effet $(\rho_1, \omega_1) = (\{x\} \cup \rho, \{x\} \cup \omega)$, on a par hypothèse de récurrence

$$(\overline{e_1}\ \overline{\theta(\rho)}) = (\overline{\theta_1(\omega_1)}, \overline{val(\theta_1, v_1)})$$

et $val(\theta_1, v_1) = v_1$ car x est une référence sur un type pur. Par conséquent

$$\begin{array}{rcll} (\overline{e}\ \overline{\theta(\rho)}) & = & (\overline{\theta_1(\omega_1)} \oplus \{x = \overline{v_1}\}, \mathsf{void}) & \\ & = & (\overline{\theta'(\omega)}, \mathsf{void}) & \text{car } \omega = \{x\} \cup \omega_1 \\ & = & (\overline{\theta'(\omega)}, \overline{val(\theta', v)}) & \end{array}$$

– $e \equiv \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$: Avec E désignant le contexte d'évaluation $E = \mathsf{if}\ []\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$, l'évaluation de e est la suivante :

$$\begin{array}{rclll} \theta.E[e_1] & \longmapsto^\star & \theta_1.E[v_1] & & \text{avec } \theta.e_1 \longmapsto^\star \theta_1.v_1 \\ & \longmapsto & \theta_1.e_2 & \text{si } v_1 = \mathsf{true} & (\textit{if-true}) \\ & & \theta_1.e_3 & \text{si } v_1 = \mathsf{false} & (\textit{if-false}) \\ & \longmapsto^\star & \theta'.v & & \end{array}$$

Comme $\theta.e_1 \longmapsto^\star \theta_1.v_1$ on a par hypothèse de récurrence, en notant (ρ_1, ω_1) l'effet de e_1,

$$(\overline{e_1}\ \overline{\theta(\rho_1)}) = (\overline{\theta_1(\omega_1)}, v_1)$$

car v_1 est de type bool. Supposons que $v_1 = \mathsf{true}$. Alors

$$\begin{array}{rcl} (\overline{e}\ \overline{\theta(\rho)}) & = & \mathsf{let}\ (x_1, b) = (\overline{e_1}\ \overline{\theta(\rho_1)})\ \mathsf{in} \\ & & \mathsf{if}\ b\ \mathsf{then} \\ & & \quad \mathsf{let}\ (x_2, v) = (\overline{e_2}\ (\overline{\theta(\rho)} \oplus x_1))\ \mathsf{in}\ (\overline{\theta(\rho)} \oplus x_1 \oplus x_2, v) \\ & & \mathsf{else} \\ & & \quad \mathsf{let}\ (x_2, v) = (\overline{e_3}\ (\overline{\theta(\rho)} \oplus x_1))\ \mathsf{in}\ (\overline{\theta(\rho)} \oplus x_1 \oplus x_2, v) \\ & = & \mathsf{let}\ (x_2, v) = (\overline{e_2}\ (\overline{\theta(\rho)} \oplus \overline{\theta_1(\omega_1)}))\ \mathsf{in}\ (\overline{\theta(\rho)} \oplus \overline{\theta_1(\omega_1)} \oplus x_2, v) \\ & = & \mathsf{let}\ (x_2, v) = (\overline{e_2}\ \overline{\theta_1(\rho_2)})\ \mathsf{in}\ (\overline{\theta(\rho)} \oplus \overline{\theta_1(\omega_1)} \oplus x_2, v) \end{array}$$

De plus on a $\theta_1.e_2 \longmapsto^\star \theta'.v$ donc par hypothèse de récurrence, en notant (ρ_2, ω_2) l'effet de e_2,

$$(\overline{e_2}\ \overline{\theta_1(\rho_2)}) = (\overline{\theta'(\omega_2)}, \overline{val(\theta', v)})$$

et donc finalement

$$\begin{array}{rcl} (\overline{e}\ \overline{\theta(\rho)}) & = & (\overline{\theta(\rho)} \oplus \overline{\theta_1(\omega_1)} \oplus \overline{\theta'(\omega_2)}, \overline{val(\theta', v)}) \\ & = & (\overline{\theta'(\omega)}, \overline{val(\theta', v)}) \end{array}$$

car $\omega_1 \cup \omega_2 \subseteq \omega \subseteq \rho$ et par application du Lemme 4 page 33. Idem si $v_1 = \mathsf{false}$.

- $e \equiv \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$: Soit E le contexte d'évaluation $E = \mathsf{let}\ x = []\ \mathsf{in}\ e_2$. On distingue deux cas suivant que e_1 est ou non une référence.
  - $e_1 : \tau_1 \neq \_\ \mathsf{ref}$: Alors l'évaluation de e est la suivante :

$$\begin{array}{rcll} \theta.E[e_1] & \longmapsto^\star & \theta_1.E[v_1] & \text{avec } \theta.e_1 \longmapsto^\star \theta_1.v_1 \\ & \longmapsto & \theta_1.e_2[x \leftarrow v_1] & (let) \\ & \longmapsto^\star & \theta'.v & \end{array}$$

Comme $\theta.e_1 \longmapsto^\star \theta_1.v_1$ on a par hypothèse de récurrence

$$(\overline{e_1}\ \overline{\theta(\rho_1)}) = (\overline{\theta_1(\omega_1)}, \overline{val(\theta_1, v_1)})$$

avec $val(\theta_1, v_1) = v_1$ car v_1 n'est pas une référence. Donc

$$\begin{array}{rcl} (\overline{e}\ \overline{\theta(\rho)}) & = & \mathsf{let}\ (x_1, x) = (\overline{e_1}\ \overline{\theta(\rho_1)})\ \mathsf{in} \\ & & \mathsf{let}\ (x_2, v) = (\overline{e_2}\ (\overline{\theta(\rho_2)} \oplus x_1))\ \mathsf{in} \\ & & (x_1 \oplus x_2, v) \\ & = & \mathsf{let}\ (x_2, v) = (\overline{e_2}[x \leftarrow \overline{v_1}]\ \overline{\theta_1(\rho_2)})\ \mathsf{in} \\ & & (\overline{\theta_1(\omega_1)} \oplus x_2, v) \end{array}$$

De plus, on a $\theta_1.e_2[x \leftarrow v_1] \longmapsto^\star \theta'.v$ donc par hypothèse de récurrence

$$(\overline{e_2[x \leftarrow v_1]}\ \overline{\theta_1(\rho_2)}) = (\overline{\theta'(\omega_2)}, \overline{val(\theta', v)})$$

Or, de par le lemme de substitution (Lemme 7 page 52), on a

$$\overline{e_2[x \leftarrow v_1]} = \overline{e_2}[x \leftarrow \overline{v_1}]$$

donc

$$\begin{array}{rcl} (\overline{e}\ \overline{\theta(\rho)}) & = & (\overline{\theta_1(\omega_1)} \oplus \overline{\theta'(\omega_2)}, \overline{val(\theta', v)}) \\ & = & (\overline{\theta'(\omega)}, \overline{val(\theta', v)}) \end{array}$$

car $\forall x \in \omega_1 \backslash \omega_2.\ \theta'(x) = \theta_1(x)$ (Lemme 4 page 33).
  - $e_1 : \tau_1\ \mathsf{ref}$: Alors l'évaluation de e est la suivante :

$$\begin{array}{rcll} \theta.E[e_1] & \longmapsto^\star & \theta_1.E[x_1] & \text{avec } \theta.e_1 \longmapsto^\star \theta_1.v_1 \text{ et } x_1 \in dom(\theta_1) \\ & \longmapsto & \theta_1.e_2[x \leftarrow x_1] & (let) \\ & \longmapsto^\star & \theta'.v & \end{array}$$

Comme $\theta.e_1 \longmapsto^\star \theta_1.x_1$ on a par hypothèse de récurrence

$$(\overline{e_1}\ \overline{\theta(\rho)}) = (\overline{\theta_1(\omega_1)}, \overline{val(\theta_1, x_1)})$$

avec $val(\theta_1, x_1) = \theta_1(x_1)$ car x_1 est une référence. Donc

$$\begin{array}{rcl}(\overline{e}\ \overline{\theta(\rho)}) & = & \mathsf{let}\ (x_2, v) = (\overline{e_2}\ \overline{\theta(\rho)} \oplus \overline{\theta_1(\rho_1)} \oplus \{x = \overline{\theta_1(x_1)}\})\ \mathsf{in} \\ & & (\overline{\theta_1(\rho_1)} \oplus x_2 \backslash x, v)\end{array}$$

De plus, on a $\theta_1.e_2[x \leftarrow x_1] \longmapsto^\star \theta'.v$ donc par hypothèse de récurrence

$$(\overline{e_2[x \leftarrow x_1]}\ \overline{\theta_1(\rho_2[x \leftarrow x_1])}) = (\overline{\theta'(\omega_2[x \leftarrow x_1])}, \overline{val(\theta', v)})$$

On a $\Gamma, x : \tau_1\ \mathsf{ref} \vdash_r e_2 : (\tau, \epsilon)$ avec $x \notin \tau$, donc on a $\Gamma, x : \tau_1\ \mathsf{ref} \vdash_r e_2[x \leftarrow x_1] : (\tau, \epsilon[x \leftarrow x_1])$. Or on a trivialement l'identité suivante (il s'agit d'un renommage de variables) :

$$(\overline{e_2[x \leftarrow x_1]}\ x[x \leftarrow x_1]) = (\overline{e_2}\ x)[x \leftarrow x_1] \tag{4.5}$$

donc

$$(\overline{e_2}\ \overline{\theta(\rho)} \oplus \overline{\theta_1(\rho_1)} \oplus \{x = \overline{\theta_1(x_1)}\}) = (\overline{\theta'(\omega_2)}, \overline{val(\theta', v)})$$

car $x \notin \tau \Rightarrow x \notin val(\theta', v)$. Et donc finalement

$$\begin{array}{rcl}(\overline{e}\ \overline{\theta(\rho)}) & = & (\overline{\theta_1(\omega_1)} \oplus \overline{\theta'(\omega_2)} \backslash x, \overline{val(\theta', v)}) \\ & = & (\overline{\theta'(\omega)}, \overline{val(\theta', v)})\end{array}$$

car $\omega = \omega_1 \cup \omega_2 \backslash x$.

- $e \equiv (e_2\ e_1)$: On distingue deux cas suivant que e_1 est ou non une référence.
  - $e_1 : \tau_1 \neq \_\ \mathsf{ref}$: Soit E le contexte d'évaluation $E = (e_2\ [])$. Alors il y a différentes évaluations possibles de e suivant que e_2 s'évalue en une constante, en l'opérateur Y ou en une abstraction :

    1.
    $$\begin{array}{rcll}\theta.E[e_1] & \longmapsto^\star & \theta_1.E[v_1] & \text{avec } \theta.e_1 \longmapsto^\star \theta_1.v_1 \\ & \longmapsto^\star & \theta'.(c\ v_1) & \text{avec } \theta_1.e_2 \longmapsto^\star \theta_2.c \\ & \longmapsto & \theta'.\delta(c, v_1) & (\delta)\end{array}$$

    On a $\theta.e_1 \longmapsto^\star \theta_1.v_1$ donc par hypothèse de récurrence

    $$(\overline{e_1}\ \overline{\theta(\rho)}) = (\overline{\theta_1(\omega_1)}, \overline{val(\theta_1, v_1)}) = (\overline{\theta_1(\omega_1)}, \overline{v_1}) = (\overline{\theta_1(\omega_1)}, v_1)$$

    car v_1 est d'un type pur. De même $\theta_1.e_2 \longmapsto^\star \theta_2.c$ donc par hypothèse de récurrence

    $$(\overline{e_2}\ \overline{\theta_1(\rho_2)}) = (\overline{\theta'(\omega_2)}, \overline{val(\theta', c)}) = (\overline{\theta'(\omega_2)}, c)$$

    donc

    $$\begin{array}{rcl}(\overline{e}\ \overline{\theta(\rho)}) & = & \mathsf{let}\ v = (c\ v_1)\ \mathsf{in}\ (\overline{\theta_1(\omega_1)} \oplus \overline{\theta'(\omega_2)}, v) \\ & = & (\overline{\theta'(\omega)}, \overline{\delta(c, v_1)}) \\ & = & (\overline{\theta'(\omega)}, \overline{val(\theta', v)})\end{array}$$

    car $\overline{val(\theta', v)} = \overline{\delta(c, v_1)} = \delta(c, v_1)$.

2.

$$
\begin{array}{rcll}
\theta.E[e_1] & \longmapsto^\star & \theta_1.E[v_1] & \text{avec } \theta.e_1 \longmapsto^\star \theta_1.v_1 \\
 & \longmapsto^\star & \theta_2.(\mathsf{Y}\ v_1) & \text{avec } \theta_1.e_2 \longmapsto^\star \theta_2.\mathsf{Y} \\
 & \longmapsto & \theta_2.(v_1\ \lambda x.((\mathsf{Y}\ v_1)\ x)) & (Y) \\
 & \longmapsto^\star & \theta'.v &
\end{array}
$$

On a $\theta.e_1 \longmapsto^\star \theta_1.v_1$ donc par hypothèse de récurrence

$$(\overline{e_1}\ \overline{\theta(\rho)}) = (\overline{\theta_1(\omega_1)}, \overline{val(\theta_1, v_1)}) = (\overline{\theta_1(\omega_1)}, \overline{v_1})$$

car v_1 est une fonctionnelle. De même $\theta_1.e_2 \longmapsto^\star \theta_2.\mathsf{Y}$ donc par hypothèse de récurrence

$$(\overline{e_2}\ \overline{\theta_1(\rho_2)}) = (\overline{\theta'(\omega_2)}, \overline{val(\theta', \mathsf{Y})}) = (\overline{\theta'(\omega_2)}, \mathsf{Y})$$

donc, étant donné que l'application de Y est sans effets, on a

$$
\begin{array}{rcl}
(\overline{e}\ \overline{\theta(\rho)}) & = & \mathsf{let}\ v = (\mathsf{Y}\ \overline{v_1})\ \mathsf{in}\ (\overline{\theta_1(\omega_1)} \oplus \overline{\theta_2(\omega_2)}, v) \\
 & = & \mathsf{let}\ v = (\overline{v_1}\ \lambda x.(\mathsf{Y}\ \overline{v_1}\ x))\ \mathsf{in}\ (\overline{\theta_1(\omega_1)} \oplus \overline{\theta_2(\omega_2)}, v) \\
 & = & \mathsf{let}\ v = (\overline{v_1}\ \lambda x.(\mathsf{Y}\ \overline{v_1}\ x))\ \mathsf{in}\ (\overline{\theta'(\omega)}, v)
\end{array}
$$

Or $\theta_2.(v_1\ \lambda x.((\mathsf{Y}\ v_1)\ x)) \longmapsto^\star \theta'.v$ donc par hypothèse de récurrence on a

$$\overline{(v_1\ \lambda x.((\mathsf{Y}\ v_1)\ x))} = \overline{val(\theta', v)}$$

d'où la conclusion car

$$\overline{(v_1\ \lambda x.((\mathsf{Y}\ v_1)\ x))} = (\overline{v_1}\ \lambda x.(\mathsf{Y}\ \overline{v_1}\ x))$$

3.

$$
\begin{array}{rcll}
\theta.E[e_1] & \longmapsto^\star & \theta_1.E[v_1] & \text{avec } \theta.e_1 \longmapsto^\star \theta_1.v_1 \\
 & \longmapsto^\star & \theta_2.(\lambda x.e_3\ v_1) & \text{avec } \theta_1.e_2 \longmapsto^\star \theta_2.\lambda x.e_3 \\
 & \longmapsto^\star & \theta_2.e_3[x \leftarrow v_1] & (\beta) \\
 & \longmapsto^\star & \theta'.v &
\end{array}
$$

Ce cas se traite de la même manière qu'un let.

– $e_1 = r : \tau_1\ \mathsf{ref}$: on a $\Gamma \vdash_r e_2 : ((x : \tau_1\ \mathsf{ref}) \to^{\epsilon_3} \tau, \epsilon_2)$.

$$
\begin{array}{rcll}
\theta.(e_2\ r) & \longmapsto^\star & \theta_1.(\lambda x.e_3\ r) & \\
 & \longmapsto & \theta_1.e_3[x \leftarrow r] & (\beta) \\
 & \longmapsto^\star & \theta'.v &
\end{array}
$$

On a $\theta.e_2 \longmapsto^\star \theta_1.\lambda x.e_3$ donc par hypothèse de récurrence

$$(\overline{e_2}\ \overline{\theta_1(\rho_2)}) = (\overline{\theta_1(\omega_2)}, \overline{val(\theta_1, \lambda x.e_3)})$$

avec $\overline{val(\theta_1, \lambda x.e_3)} = \overline{\lambda x.e_3} = \lambda x.\overline{e_3}$ car x est une référence. Donc

$$
\begin{array}{rcl}
(\overline{e}\ \overline{\theta(\rho)}) & = & \mathsf{let}\ (x_2, v) = (\lambda x.\overline{e_3}\ r\ (\overline{\theta(\omega)} \oplus \overline{\theta_1(\omega_2)}))\ \mathsf{in} \\
 & & (\overline{\theta_1(\omega_2)} \oplus x_2, v) \\
 & = & \mathsf{let}\ (x_2, v) = (\overline{e_3}[x \leftarrow r]\ (\overline{\theta(\omega)} \oplus \overline{\theta_1(\omega_2)}))\ \mathsf{in} \\
 & & (\overline{\theta_1(\omega_2)} \oplus x_2, v)
\end{array}
$$

Or $\theta_1.e_3[x \leftarrow r] \longmapsto^\star \theta'.v$ donc par hypothèse de récurrence

$$(\overline{e_3[x \leftarrow r]}\ \overline{\theta_1(\rho_3)}) = (\overline{\theta'(\omega_3)}, \overline{val(\theta', v)})$$

et on conclut de même que pour le let, en utilisant l'identité (4.5).

$\square$

Théorème 2 (réciproque) *Soit* Γ *un environnement bien formé dont les références sont* $x_1 : \tau_1\ \mathsf{ref}, \dots, x_n : \tau_n\ \mathsf{ref}$ *et* e *un programme tel que* $\Gamma \vdash_r e : (\tau, (\rho, \omega))$. *Alors pour tout état* θ *associant des valeurs aux variables* x_i *on a*

$$\forall y, v_0.\quad (\overline{e}\ \overline{\theta(\rho)}) = (y, v_0) \quad\Longrightarrow\quad \exists \theta', v.\ \theta.e \longmapsto^\star \theta'.v$$

Preuve. La preuve se fait par l'absurde : notons x_0 l'enregistrement $\overline{\theta(\rho)}$ et supposons qu'il existe θ, y et v_0 tels que $(\overline{e}\ x_0) = (y, v_0)$ et tels que l'évaluation de $\theta.e$ ne termine pas ; alors on peut montrer que l'évaluation de $(\overline{e}\ x_0)$ ne devrait pas terminer non plus, ce qui est contradictoire. En effet, nous allons montrer que l'évaluation de $(\overline{e}\ x_0)$ donne lieu à au moins autant de réductions élémentaires que l'évaluation de $\theta.e$. Dès lors, $(\overline{e}\ x_0)$ ne saurait terminer sans qu'il en soit de même pour $\theta.e$.

Afin de rendre la preuve de ce résultat plus aisée, nous allons η-expanser la définition de la traduction fonctionnelle sur la construction ref : nous avions $\overline{\mathsf{ref}\ e_1} \stackrel{\text{def}}{=} \overline{e_1}$ et nous posons maintenant

$$\overline{\mathsf{ref}\ e_1} \quad\stackrel{\text{def}}{=}\quad \lambda x_0.\mathsf{let}\ (x_1, v) = (\overline{e_1}\ x_0)\ \mathsf{in}\ (x_1, v)$$

Cela ne change en rien, bien sûr, la définition de la traduction fonctionnelle. Nous pouvons maintenant énoncer et prouver le lemme dont nous avons besoin.

Lemme 13 *On se place dans les hypothèses du théorème ci-dessus, et l'on note toujours* x_0 *l'enregistrement* $\overline{\theta(\rho)}$. *Alors, si la réduction de* $\theta.e$ *par* $\longmapsto$ *requiert* N *étapes, l'évaluation de* $(\overline{e}\ x_0)$ *requiert au moins* N *réductions élémentaires.*

Preuve. On ne décompte pas dans les réductions de $\theta.e$ les opérations manipulant l'état, à savoir (θ_{merge}) et (θ_{lift}), car il est clair qu'une réduction infinie implique une infinité de réductions en dehors de ces deux réductions-là.

La preuve se fait par récurrence sur N et par cas sur la structure de e.

- Si e est une valeur alors le résultat est immédiat car il n'y a aucune réduction de part et d'autre — la notion de valeur étant la même dans les langages source et cible.
- $e \equiv\ !x$: Dans ce cas, il y a exactement une réduction de chaque côté.
- $e \equiv x\ :=\ e_1$: L'évaluation de e est la suivante :

$$\theta.x\ :=\ e_1 \longmapsto^{N_1} \theta_1.x\ :=\ v_1 \longmapsto \theta_1 \uplus \{(x, v_1)\}.\mathsf{void}$$

avec

$$\theta.e_1 \longmapsto^{N_1} \theta_1.v_1$$

On a

$$(\overline{e}\ x_0) = \mathsf{let}\ (x_1, v) = (\overline{e_1}\ x_0)\ \mathsf{in}\ (x_1 \oplus \{x = v\}, \mathsf{void})$$

Or par hypothèse de récurrence appliquée à $\theta.e_1 \longmapsto^{N_1} \theta_1.v_1$, l'évaluation de $(\overline{e_1}\ x_0)$ requiert au moins N_1 étapes donc l'évaluation de $(\overline{e}\ x_0)$ requiert au moins $N_1 + 1 = N$ étapes.

- $e \equiv \mathsf{ref}\ e_1$: L'évaluation de e est de la forme :

  $$\theta.e \longmapsto^{N_1} \theta_1.\mathsf{ref}\ v_1 \longmapsto \theta_1 \uplus \{(x, v_1)\}.x$$

  et

  $$(\overline{e}\ x_0) = \mathsf{let}\ (x_1, v) = (\overline{e_1}\ x_0)\ \mathsf{in}\ (x_1, v)$$

  Par hypothèse de récurrence, l'évaluation de $(\overline{e_1}\ x_0)$ requiert au moins N_1 étapes donc l'évaluation de $(\overline{e}\ x_0)$ requiert au moins $N_1 + 1 = N$ étapes.
- $e \equiv \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$: La récurrence est immédiate, comme pour les deux cas précédents.
- $e \equiv \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$: On distingue deux cas suivant le type de e_1 :
  - $e_1 : \tau\ \mathsf{ref}$: L'évaluation de e est alors de la forme suivante

    $$\theta.e \longmapsto^{N_1} \theta_1.\mathsf{let}\ x = x_1\ \mathsf{in}\ e_2 \longmapsto \theta_1.e_2[x \leftarrow x_1] \longmapsto^{N_2} \theta_2.v$$

    et

    $$\begin{aligned}(\overline{e}\ x_0) = \ & \mathsf{let}\ (x_1, v_0) = (\overline{e_1}\ x_0)\ \mathsf{in}\\ & \mathsf{let}\ (x_2, v) = (\overline{e_2}\ (x_0 \oplus x_1 \oplus \{x = v_0\}))\ \mathsf{in}\\ & (x_1 \oplus x_2 \backslash x, v)\end{aligned}$$

    Par hypothèse de récurrence, l'évaluation de $(\overline{e_1}\ x_0)$ requiert au moins N_1 réductions, puis une réduction supplémentaire amène à évaluer $\overline{e_2}$ sur l'entrée $(x_0 \oplus x_1 \oplus \{x = v_0\})$. Or par hypothèse de récurrence l'évaluation de $\overline{e_2[x \leftarrow x_1]}$ requiert au moins N_2 réductions. Il est donc de même de $\overline{e_2}$ car $\overline{e_2[x \leftarrow x_1]} = \overline{e_2}[x \leftarrow x_1]$. L'évaluation de $\overline{e}\ x_0)$ requiert donc au moins $N_1 + N_2 + 2$ réductions, d'où le résultat.
  - $e_1 : \_ \neq \tau\ \mathsf{ref}$: L'évaluation de e est alors de la forme suivante

    $$\theta.e \longmapsto^{N_1} \theta_1.\mathsf{let}\ x = v_1\ \mathsf{in}\ e_2 \longmapsto \theta_1.e_2[x \leftarrow v_1] \longmapsto^{N_2} \theta_2.v$$

    et

    $$\begin{aligned}(\overline{e}\ x_0) = \ & \mathsf{let}\ (x_1, x) = (\overline{e_1}\ x_0)\ \mathsf{in}\\ & \mathsf{let}\ (x_2, v) = (\overline{e_2}\ (x_0 \oplus x_1))\ \mathsf{in}\\ & (x_1 \oplus x_2, v)\end{aligned}$$

    Par hypothèse de récurrence, l'évaluation de $(\overline{e_1}\ x_0)$ requiert au moins N_1 réductions, puis une réduction supplémentaire amène à évaluer $(\overline{e_2}[x \leftarrow \overline{v_1}]\ (x_0 \oplus x_1))$. Or $\overline{e_2}[x \leftarrow \overline{v_1}] = \overline{e_2[x \leftarrow v_1]}$ (Lemme 7 page 52) et par hypothèse de récurrence l'évaluation de $\overline{e_2[x \leftarrow v_1]}$ requiert au moins N_2 réductions, donc l'évaluation de $(\overline{e}\ x_0)$ requiert au moins $N_1 + N_2 + 2$ réductions, d'où le résultat.

- $e \equiv (e_2\ e_1)$: On distingue deux cas suivant le type de e_1 :
  - $e_1 = r : \tau\ \mathsf{ref}$: Alors e_2 s'évalue nécessairement en une abstraction $\lambda x.e_3$, et l'évaluation de e est donc de la forme suivante :

$$\begin{array}{rll} \theta.(e_2\ r) & \longmapsto^{N_1} & \theta_1.(\lambda x.e_3\ r) \\ & \longmapsto & \theta_1.e_3[x \leftarrow r] \qquad (\beta) \\ & \longmapsto^{N_2} & \theta'.v \end{array}$$

    et

$$\begin{array}{rl} (\overline{e}\ x_0) = & \mathsf{let}\ (x_1, f) = (\overline{e_2}\ x_0)\ \mathsf{in} \\ & \mathsf{let}\ (x_2, v) = (f\ r\ (x_0 \oplus x_1))\ \mathsf{in} \\ & (x_1 \oplus x_2, v) \end{array}$$

    Par hypothèse de récurrence, l'évaluation de $(\overline{e_2}\ x_0)$ requiert au moins N_1 réductions. Son résultat est $\overline{\lambda x.e_3} = \lambda x.\overline{e_3}$. Or par hypothèse de récurrence, on sait que l'évaluation de $\overline{e_3[x \leftarrow r]} = \overline{e_3}[x \leftarrow r]$ sur une entrée requiert au moins N_2 réductions. Il en est donc de même de $\lambda x.\overline{e_3}$ sur r et $(x_0 \oplus x_1)[r \leftarrow x]$. Donc l'évaluation de $(\overline{e}\ x_0)$ requiert au moins $N_1 + N_2 + 3$ réductions, d'où le résultat.
  - $e_1 : \_ \neq \tau\ \mathsf{ref}$: Il faut distinguer trois cas, suivant la valeur de e_2 :
    - e_2 s'évalue en une constante :

$$\theta.e \longmapsto^{N_1} \theta_1.(e_2\ v_1) \longmapsto^{N_2} \theta_2.(c\ v_1) \longmapsto \theta_2.\delta(c, v_1)$$

      Et la récurrence est immédiate, comme pour les cas ref et if.
    - e_2 s'évalue en Y : L'évaluation de e est de la forme

$$\begin{array}{rll} \theta.e & \longmapsto^{N_1} & \theta_1.(e_2\ v_1) \\ & \longmapsto^{N_2} & \theta_2.(\mathsf{Y}\ v_1) \\ & \longmapsto & \theta_2.(v_1\ \lambda x.(\mathsf{Y}\ v_1\ x)) \\ & \longmapsto^{N_3} & \theta_3.v \end{array}$$

      et

$$\begin{array}{rl} (\overline{e}\ x_0) = & \mathsf{let}\ (x_1, a) = (\overline{e_1}\ x_0)\ \mathsf{in} \\ & \mathsf{let}\ (x_2, f) = (\overline{e_2}\ (x_0 \oplus x_1))\ \mathsf{in} \\ & \mathsf{let}\ (x_3, v) = (f\ a\ (x_0 \oplus x_1 \oplus x_2))\ \mathsf{in} \\ & (x_1 \oplus x_2 \oplus x_3, v) \end{array}$$

      Par hypothèse de récurrence, l'évaluation de $(\overline{e_1}\ x_0)$ requiert au moins N_1 réductions. De même, l'évaluation de $(\overline{e_2}\ (x_0 \oplus x_1))$ requiert au moins N_2 réductions. Une fois ces deux évaluations faites, on est amené à évaluer $(\mathsf{Y}\ \overline{v_1})$ sur une certaine entrée, donc $(\overline{v_1}\ \lambda x.(\mathsf{Y}\ \overline{v_1}\ x))$ sur cette même entrée après une réduction. Or, par hypothèse de récurrence, on sait que l'évaluation de $\overline{(v_1\ \lambda x.(\mathsf{Y}\ v_1\ x))}$ sur une entrée requiert au moins N_3 réductions. D'autre part, v_1 et $\lambda x.(\mathsf{Y}\ v_1\ x)$ étant toutes deux des valeurs, on a

$$\overline{(v_1\ \lambda x.(\mathsf{Y}\ v_1\ x))} = \lambda x_0.\mathsf{let}\ (x_1, v) = (\overline{v_1}\ \lambda x.(\mathsf{Y}\ \overline{v_1}\ x)\ x_0)\ \mathsf{in}\ (x_1, v)$$

      Donc finalement l'évaluation du dernier let requiert au moins N_3 réductions, soit au total au moins $N_1 + N_2 + N_3 + 2$ réductions, d'où le résultat.

– e_2 s'évalue en une abstraction $\lambda x.e_3$: la récurrence est immédiate et similaire au premier cas du let in avec une étape supplémentaire (réduction de la fonction).

□

□

B Architecture de l'implantation

Nous présentons dans cette section l'architecture générale de l'implantation, et comment elle se rattache à la théorie. Chaque module de l'implantation contient des types et des fonctions qui correspondent à des définitions que nous avons introduites précédemment, et nous l'indiquerons à chaque fois par un renvoi à la définition correspondante. Les modules de l'implantation, et leur principales fonctions, sont les suivants, par ordre d'apparition :

`Effects` Ce module définit le type `Effects.t` des effets et les opérations associées. Un effet représente un couple de deux ensembles d'identificateurs, qui peuvent représenter soit des références soit des tableaux (et en fait même n'importe quelle variable mutable). Le type `Effects.t` implante la Définition 8 page 22.

`ProgAst` Ce module définit les types de valeurs et de calculs annotés ainsi que les programmes annotés. Plus précisément,

- `ProgAst.ml_type_v` et `ProgAst.ml_type_c` implantent respectivement les types τ et κ (Définition 28 page 63) ;
- `ProgAst.t` et `ProgAst.t_desc` implantent respectivement les programmes annotés e et les expressions de programmes s (Définition 34 page 67).

`Prog_env` Ce module définit les environnements de programmes, c'est-à-dire une table d'association entre des identificateurs et des types annotés. Il implante donc les structures dénotées par Γ dans le discours théorique (Définition 4 page 14). Si dans la théorie on utilise un unique environnement Γ pour écrire les jugements de typage, on utilise dans la pratique deux notions d'environnement : un environnement global, unique et impératif, et une notion d'environnement local fonctionnel (type `local_env`). L'environnement global stocke les introductions de variables globales, les programmes déjà prouvés, etc. alors qu'un environnement local sera utilisé par exemple par l'algorithme de typage lors de son parcours à l'intérieur des programmes.

`Monad` Ce module introduit la monade utilisée pour la traduction des programmes, c'est-à-dire d'une part la traduction des types et d'autre part les opérations de base permettant la traduction des programmes. En particulier :

- Les fonctions `trad_ml_type_v` et `trad_ml_type_c` implantent respectivement les traductions $\widehat{\tau}$ et $\widehat{\kappa}$ (Définition 30 page 64) ;
- La fonction `make_let_in` implante l'opération let in de la monade ;
- La fonction `result_tuple` implante l'opération unit de la monade.

`TradEnv` Ce module définit les traductions d'environnement. Il exporte principalement les fonctions suivantes :

- `cci_sign_of` qui implante la traduction $\widehat{\Gamma}$ (Définition 30 page 64) ;
- `before_sign_of` et `before_after_sign_of` qui implantent respectivement les traductions $Pre(\Gamma)$ et $Post(\Gamma, r : \tau)$ (Définition 31 page 65).

`Prog_typing` Ce module implante le jugement de typage $\Gamma \vdash_a e : \kappa$ (Définition 35 page 67) sous la forme d'une fonction prenant en argument Γ (environnement local) et e et retournant non pas seulement κ mais le programme e dans lequel chaque sous-expression, y compris le programme e lui-même, est annoté de son type de calcul κ.

`Prog_wp` Ce module implante la propagation des annotations à l'intérieur des programmes. En effet, afin d'éviter à l'utilisateur le recours à un trop grand nombre d'annotations, un certain nombre de propagations correctes est effectué automatiquement, telle que par exemple :

$$(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3)\ \{Q\} \quad \rightarrow \quad (\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \{Q\}\ \mathsf{else}\ e_3\ \{Q\})\ \{Q\}$$

`Mlise` Ce module implante l'interprétation dans le Calcul des Constructions $\widehat{e}$ (Définition 37 page 75), sous la forme d'une fonction `trans` prenant en argument un environnement local et un programme complètement typé *i.e.* où toutes les sous-expressions sont annotées par leur type de calcul (à la manière des programmes produits par le module précédent).

`Prog_tactic` Enfin, ce dernier module combine les trois modules précédents pour définir la commande `Correctness`. Étant donné un programme annoté e, il effectue les opérations suivantes :

1. le programme e est typé grâce au module `Prog_typing`. Appelons κ son type ;
2. certaines annotations sont propagées à l'intérieur de e grâce au module `Prog_wp` ;
3. le but Coq $\widehat{\kappa}$ est calculé grâce au module `Monad` et déclaré comme nouveau but ;
4. le terme de preuve partiel $\widehat{e}$ est calculé grâce au module `Mlise` ;
5. la tactique Refine prouve partiellement le but en utilisant $\widehat{e}$ et une tactique automatique passe derrière pour prouver les buts logiques triviaux et effectuer quelques réécritures. Seules subsistent alors les obligations de preuves ;
6. une fois la preuve terminée et sauvée par la commande Save, le programme et sa spécification sont sauvegardés dans l'environnement global (le programme lui-même est également conservé pour une possible future extraction). La commande Save de Coq a donc été surchargée à cet effet.

Bibliographie

Une version en ligne de cette bibliographie, contenant des pointeurs vers des versions électroniques d'un grand nombre d'articles, est disponible à l'adresse suivante : `http://www.lri.fr/~filliatr/biblio-these.html`

[1] The Coq Proof Assistant. `http://coq.inria.fr/`.

[2] The Objective Caml language. `http://pauillac.inria.fr/ocaml/`.

[3] *Proceedings of the Second B International Conference*, Montpellier, France, April 1998. Springer Verlag LNCS 1393.

[4] J. R. Abrial. *The B-Book. Assigning programs to meaning.* Cambridge University Press, 1996.

[5] R. J. R. Back. On Correct Refinement of Programs. *Journal of Computer and Systems Sciences*, 23(1) :49–68, August 1981.

[6] J. L. Bentley. *Writing Efficient Programs.* Prentice Hall, 1982.

[7] J. L. Bentley. *Programming Pearls.* Addison-Wesley, 1989.

[8] P. Chartier. Formalization of B in Isabelle/HOL. In *Proceedings of the Second B International Conference*, Montpellier, France, April 1998. Springer Verlag LNCS 1393.

[9] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3), 1988.

[10] P. Cousot. *Handbook of Theoretical Computer Science*, volume B, chapter Methods and Logics for Proving Programs, pages 841–993. Elsevier Science Publishers B. V., 1990.

[11] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[12] R. W. Floyd. Algorithm 113 : Treesort. *Communications of the ACM*, 5(8), August 1962.

[13] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science, Proceedings of Symposia in applied Mathematics 19*, pages 19–32, Providence, 1967. American Mathematical Society.

[14] T. Fuchß, W. Reif, G. Schellhorn, and K. Stenzel. Three Selected Case Studies in Verification. In M. Broy and S. Jähnichen, editors, *KORSO : Methods, Languages, and Tools for the Construction of Correct Software – Final Report.* Springer LNCS 1009, 1995.

[15] E. Giménez. Two Approaches to the Verification of Concurrent Programs in Coq. Personal communication, 1999.

[16] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Thèse de doctorat d'Etat, Univerité Paris 7, 1972.

[17] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge University Press, 1989.

[18] Mike Gordon. *Teaching and Learning Formal Methods*, chapter Teaching hardware and software verification in a uniform framework. Academic Press, 1996.

[19] D. Gries. *The Science of Programming.* Springer-Verlag, 1981.

[20] J. Guzmán and A. Suárez. An extended type system for exceptions. In *Record of the fifth ACM SIGPLAN workshop on ML and its Applications.* Research Report 2265, INRIA, BP 105 - 78153 Le Chesnay Cedex, France, June 1994.

[21] I. Hayes. *Specification Case Studies.* Oxford University Computing Laboratory, 1985. Technical monograph PRG-46.

[22] C. A. R. Hoare. Algorithm 65 : Find. *Communications of the ACM*, 4(7), July 1961.

[23] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1) :10–15, April 1962.

[24] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580,583, 1969. Also in [26] pages 45–58.

[25] C. A. R. Hoare. Proof of a program : *Find. Communications of the ACM*, 14(1) :39–45, January 1971. Also in [26] pages 59–74.

[26] C. A. R. Hoare and C. B. Jones. *Essays in Computing Science.* Prentice Hall, New York, 1989.

[27] C. B. Jones. *Software development. A rigorous Approach.* Prentice Hall, 1980.

[28] C. B. Jones. *Systematic Software Development using VDM.* C. A. R. Hoare Series Editor. Prentice Hall, 1989.

[29] C. B. Jones. *VDM : une méthode rigoureuse pour le développement du logiciel.* Masson, 1991. Traduction française de [28].

[30] D. J. King and P. Wadler. Combining monads. In *Glasgow Workshop on Functional Programming.* Springer-Verlag Workshops in Computing Series, July 1992.

[31] T. Kleymann. Metatheory of Verification Calculi in LEGO : To What Extent Does Syntax Matter ? Technical Report 98-393, LFCS, September 1998.

[32] D. E. Knuth. *The Art of Computer Programming. Volume 1 : Fundamental Algorithms.* Addison-Wesley, 1968.

[33] D. E. Knuth. *The Art of Computer Programming. Volume 2 : Seminumerical Algorithms.* Addison-Wesley, 1969.

[34] D. E. Knuth. *The Art of Computer Programming. Volume 3 : Sorting and Searching.* Addison-Wesley, 1973.

[35] D. E. Knuth. *Selected Papers on Computer Science.* CSLI Publications and Cambridge University Press, 1996.

[36] X. Leroy. *The Objective Caml system : Documentation and user's manual*, March 1997. `http://pauillac.inria.fr/ocaml/htmlman/`.

[37] X. Leroy and P. Weis. Polymorphic type inference and assignment. In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages*, pages 291–302, 1991.

[38] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Edinburgh University, Department of Computer Science, June 1989. Lecture Notes for course CS 359, Stanford University.

[39] E. Moggi. Computational lambda-calculus and monads. In *IEEE Symposium on Logic in Computer Science*, 1989.

[40] J.-F. Monin. *Comprendre les méthodes formelles*. Masson, 1996.

[41] C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.

[42] J. H. Morris and V. R. Pratt. A Linear Pattern Matching Algorithm. Technical Report 40, Computing Center, University of California, Berkeley, 1970.

[43] C. Muñoz. Supporting the B-method in PVS : An Approach to the Abstract Machine Notation in Type Theory. Submitted to FSE-98, 1998.

[44] P. W. O'Hearn and J. C. Reynolds. From Algol to Polymorphic Linear Lambda-calculus. Unpublished, April 1997.

[45] C. Parent. Developing certified programs in the system Coq – The Program tactic. Technical Report 93-29, Ecole Normale Supérieure de Lyon, October 1993. Also in Proceedings of the BRA Workshop Types for Proofs and Programs, May 93.

[46] C. Parent. *Synthèse de preuves de programmes dans le Calcul des Constructions Inductives*. Thèse de doctorat, Ecole Normale Supérieure de Lyon, Janvier 1995.

[47] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, January 1989. ACM.

[48] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. Thèse de doctorat, Université de Paris VII, Janvier 1989.

[49] C. Paulin-Mohring. Réalisabilité et extraction de programmes. In *Logique et informatique : une introduction*, pages 163–180. INRIA, 1991.

[50] C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in LNCS, 1993. Also LIP research report 92-49.

[51] W. Reif. The KIV-approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO : Methods, Languages, and Tools for the Construction of Correct Software – Final Report*. Springer LNCS 1009, 1995.

[52] T. Schreiber. Auxiliary Variables and Recursive Procedures. In *TAPSOFT'97 : Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 697–711. Springer-Verlag, April 1997.

[53] R. Sedgewick. *Quicksort*. Garland, New York, 1978. Also appeared as the author's Ph.D. dissertation, Stanford University, 1975.

[54] R. Sedgewick. *Algorithms*. Addison-Wesley, 1988.

[55] J.-P. Talpin and P. Jouvelot. The Type and Effect Discipline. *Information and Computation*, 111(2) :245–296, 1994.

[56] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, pages 461–493, 1992.

[57] P. Wadler. Monads for functional programming. In *Proceedings of the Marktoberdorf Summer School on Program Design Calculi*, August 1992.

[58] J. W. J. Williams. Algorithm 232 : Heapsort. *Communications of the ACM*, 7(6), June 1964.

[59] A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115 :38–94, 1994.

Index

U

V

W

Z

Résumé

Nous étudions le problème de la certification de programmes mêlant traits impératifs et fonctionnels dans le cadre de la théorie des types.

La théorie des types constitue un puissant langage de spécification, naturellement adapté à la preuve de programmes purement fonctionnels. Pour y certifier également des programmes impératifs, nous commençons par exprimer leur sémantique de manière purement fonctionnelle. Cette traduction repose sur une analyse statique des effets de bord des programmes, et sur l'utilisation de la notion de monade, notion que nous raffinons en l'associant à la notion d'effet de manière générale. Nous montrons que cette traduction est sémantiquement correcte.

Puis, à partir d'un programme annoté, nous construisons une preuve de sa spécification, traduite de manière fonctionnelle. Cette preuve est bâtie sur la traduction fonctionnelle précédemment introduite. Elle est presque toujours incomplète, les parties manquantes étant autant d'obligations de preuve qui seront laissées à la charge de l'utilisateur. Nous montrons que la validité de ces obligations entraîne la correction totale du programme.

Nous avons implanté notre travail dans l'assistant de preuve Coq, avec lequel il est dès à présent distribué. Cette implantation se présente sous la forme d'une tactique prenant en argument un programme annoté et engendrant les obligations de preuve. Plusieurs algorithmes non triviaux ont été certifiés à l'aide de cet outil (Find, Quicksort, Heapsort, algorithme de Knuth-Morris-Pratt).

Mots clés. Preuve de programmes. Théorie des Types. Programmation Impérative. Analyse statique des Effets. Monades. Calcul des Constructions Inductives. Système Coq.

Abstract

We study the problem of certifying programs mixing imperative and functional features in the general framework of type theory.

Type theory constitutes a powerful specification language, which is naturally suited for the proof of purely functional programs. To deal with imperative programs, we first express their semantics in a purely functional way. That translation is built on a static analysis of their effects, and on the use of monads, a notion that we refine in order to make it depend on the notion of effect. We show that this translation is semantically correct.

Then, given an annotated program, we build a proof of its specification, which is expressed as a purely functional proposition. That proof is built on the functional translation of the program. It is nearly always an incomplete term, in which the remaining parts are the proof obligations, left to the user. We show that the validity of those proof obligations implies the total correctness of the program.

We have implemented our work in the proof assistant Coq, with which it is already distributed. It appears as a tactic taking an annotated program as argument and generating the proof obligations. Several non-trivial algorithms have been certified with the help of that tool (Find, Quicksort, Heapsort, Knuth-Morris-Pratt's algorithm).

Keywords. Proof of programs. Type theory. Imperative programming. Static analysis of effects. Monads. Calculus of Inductive Constructions. Coq system.